Higher Order Mutation Testing

by

Yue Jia

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of UCL

Department of Compuer Science

University College London

University of London

July 2013

Declaration

I, Yue Jia, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

The work presented in this thesis is original work undertaken between September 2007 and August 2010 at King's College, University of London and between September 2010 and January 2013 at University College London, University of London. Some of the work presented in this thesis has previously been published in the following publications:

1. Yue Jia and Mark Harman, "An Analysis and Survey of the Development of Mutation Testing", *IEEE Transactions on Software Engineering*, vol. 37 no. 5, September 2011, pp. 649 – 678. [Google Scholar Citations: 165]

2. Mark Harman, Yue Jia and William Langdon, "Strong higher order mutation-based test data generation", *in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11)*, Szeged, Hungary, 05-09 September 2011, pp. 212–222. [Google Scholar Citations: 15]

3. Yue Jia and Mark Harman, "Higher Order Mutation Testing," *Journal of Information and Software Technology*, vol. 51, no. 10, October 2009, pp. 1379–1393. [Google Scholar Citations: 45]

4. Yue Jia and Mark Harman, "Constructing Subtle faults Using Higher Order Mutation Testing", *in Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, Beijing, China, 28–29 September 2008, pp.249–258. (Best Paper Award Winner) [Google Scholar Citations: 41]

5. Yue Jia and Mark Harman, "Milu: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language", *in Proceedings of the 3rd Testing Academia and Industry Conference - Practice and Research Techniques (TAIC PART'08)*, Windsor, UK, 29–31 August 2008, pp.94–98. [Google Scholar Citations: 30]

Additionally, the following work has also been published during the programme of study, however they do not feature in this thesis:

6. Mark Harman and Bill Langdon and Yue Jia and David White and Andrea Arcuri, and John Clark, "The GISMOE challenge: Constructing the Pareto Program Surface Using Genetic Programming to Find Better Programs" (keynote paper), *in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, Essen, Germany, September 3rd-7th, 2012, pp. 1–14.

7. Paolo Tonella, Alessandro Marchetto, Cu Duy Nguyen, Yue Jia, Kiran Lakhotia, Mark Harman, "Finding the Optimal Balance between Over and Under Approximation of Models Inferred from Execution Logs", *in Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST'12)*, Montreal, Canada, April 17-21 2012, pp. 21-30.

8. Mark Harman and Yue Jia and Yuanyuan Zhang, "App Store Mining and Analysis: MSR for App Stores" (short paper), *in Proceedings of the 9th Working Conference on Mining Software Repositories (MSR '12)*, Zurich, Switzerland, 2-3 June 2012, pp. 108–111

9. Bill Langdon, Mark Harman and Yue Jia, "Efficient Multi Objective Higher Order Mutation Testing with Genetic Programming", *Journal of Systems and Software*, vol. 83, no. 12, July 2010, pp.2416–2430.

10. Mark Harman, Yue Jia and William B. Langdon, "A Manifesto for Higher Order Mutation Testing", *in Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'10)*, Paris, France, 6th April 2010, pp.80–89.

11. Jens Krinke, Nicolas Gold, Yue Jia and Dave Binkley, "Cloning and Copying between GNOME Projects", *in Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR'10)*, Cape Town, South Africa, 2-3 May 2010, pp.98–101.

12. Jens Krinke, Nicolas Gold, Yue Jia and Dave Binkley, "Distinguishing Copies from Originals in Software Clones", *in Proceedings of the 4th International Workshop on Software Clones (IWSC'10)*, Cape Town, South Africa, 8 May 2010, pp.41–48.

13. Bill W. Langdon, Mark Harman and Yue Jia, "Multi Objective Mutation Testing with Genetic Programming", *in Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques 2009 (TAIC PART'09)*, Winsor, UK, 4–6 September 2009, pp.4–6.

14. Yue Jia, Dave Binkley, Mark Harman, Jens Krinke and Makoto Matsushita, " KClone: A Proposed Approach to Fast Precise Code Clone Detection", *in Proceedings of the 3rd International Workshop on Software Clones (IWSC'09)*, Kaiserslautern, Germany, 24 March 2009, pp.12–16.

Abstract

Mutation testing is a fault-based software testing technique that has been studied widely for over three decades. To date, work in this field has focused largely on first order mutants because it is believed that higher order mutation testing is too computationally expensive to be practical. This thesis argues that some higher order mutants are potentially better able to simulate real world faults and to reveal insights into programming bugs than the restricted class of first order mutants.

This thesis proposes a higher order mutation testing paradigm which combines valuable higher order mutants and non-trivial first order mutants together for mutation testing. To overcome the exponential increase in the number of higher order mutants a search process that seeks fit mutants (both first and higher order) from the space of all possible mutants is proposed.

A fault-based higher order mutant classification scheme is introduced. Based on different types of fault interactions, this approach classifies higher order mutants into four categories: expected, worsening, fault masking and fault shifting. A search-based approach is then proposed for locating subsuming and strongly subsuming higher order mutants. These mutants are a subset of fault mask and fault shift classes of higher order mutants that are more difficult to kill than their constituent first order mutants. Finally, a hybrid test data generation approach is introduced, which combines the dynamic symbolic execution and search based software testing approaches to generate strongly adequate test data to kill first and higher order mutants.

Acknowledgements

I would like to thank my supervisor Professor Mark Harman for his understanding, guidance, endless support and advice and for providing me the opportunity to undertake my PhD. I would like to thank my second supervisors, Nicolas Gold and Jens Krinke, for their support and comments. I would also like to thank Bill Langdon and Kiran Lakhotia for their helpful discussions.

I am very grateful to Dave Binkley for his friendly, insightful and interesting discussions. Thanks also go to Pedro Reales Mateo for thoughtful discussions on mutation testing. I am thankful to Matthew Patrick for his careful proofreading of this thesis and valuable comments.

I would like to thank all my colleagues at the Centre for Research in Evolution, Search and Testing for their general support and discussions. I would also like to thank the various anonymous referees for their comments on papers submitted for publication; their comments and feedback have been extremely beneficial.

I gratefully acknowledge the Overseas Research Students Awards Scheme program, EPSRC SEBASE project studentship and UCL CS department scholarship for the financial support I have received.

I express my particular appreciation to my parents, who have always stood by me in my best and worst times. Finally, to my fiancée Yuanyuan, thank you, I could not have achieved this without your continual support through these difficult and stressful times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software testing is an important yet expensive part of the software development life cycle. Early studies suggest that testing can comprise up to 50% of the software development budget [22], and a recent survey revealed that billions of dollars are routinely wasted on large software projects due to inadequate testing [47]. A fundamental limitation of software testing is that it is impossible to enumerate all the test inputs explicitly; this is because for many software systems there are effectively an infinite number of test inputs [11]. A means of overcoming this limitation is to propose testing criteria in order to assist the software tester in choosing which test inputs to use. Testing criteria are rules that specify properties that test data must satisfy. For example, statement coverage requires that test inputs cover every statement of the software system under testing, and branch coverage requires that test inputs cover each branch of the predicate points of the software system under testing. In addition to generating test data, test criteria have been used to assess software system quality as well as to determine when testing should cease [11]. As a result, testing criteria have become an effective means of increasing the confidence in the correctness of a software system [22].

Mutation testing is a fault-based testing technique which provides a testing criterion; it can be used to measure the effectiveness of a test set in terms of the ability of the test set to detect faults. A recent work by Li et al. [161] compared a mutation testing criterion with three other commonly used criteria: prime path coverage, edge-pair coverage and all-uses. The results suggest that the mutation testing criterion not only finds more faults than other criteria, but is also the most efficient criterion.

1.1 Mutation Testing with Examples

The history of mutation testing can be traced back to 1971 in a publication by the student Richard Lipton [162] as well as in publications from the late 1970s by DeMillo et al. [68] and Hamlet [114]. The general principle behind mutation testing is that artificial faults can be used to represent common programming mistakes. By choosing carefully the location of the program and the types of faults, it is possible to simulate any test adequacy criteria. Such faults are seeded deliberately into the original program by simple syntactic changes to create a set of faulty programs called mutants, each containing a different syntactic change.

In this introduction chapter, the `TCAS` program will be used as an example to illustrate the basic concepts and problems of mutation testing. The `TCAS` program is a traffic collision avoidance system which is designed to avoid aircraft collision. The version that will be used here is from the 'Siemens Suite' in the Software-artifact Infrastructure Repository (SIR) [78]. The 'universe' test pool for the `TCAS` program will also be taken from the SIR. It includes 1608 tests achieving adequate statement coverage, branch coverage and du-path coverage.

```c
int alt_sep_test()
{
  bool enabled, tcas_equipped, intent_not_known;
  bool need_upward_RA, need_downward_RA;
  int alt_sep;
  enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)
    && (Cur_Vertical_Sep > MAXALTDIFF);
  tcas_equipped = Other_Capability == TCAS_TA;
  intent_not_known = Two_of_Three_Reports_Valid
    && Other_RAC == NO_INTENT;
  alt_sep = UNRESOLVED;
  if (enabled && ((tcas_equipped
    && intent_not_known) || !tcas_equipped))
  {
    need_upward_RA = Non_Crossing_Biased_Climb()
      && Own_Below_Threat();
    need_downward_RA = Non_Crossing_Biased_Descend()
      && Own_Above_Threat();
    if (need_upward_RA && need_downward_RA)
      alt_sep = UNRESOLVED;
    else if (need_upward_RA)
      alt_sep = UPWARD_RA;
    else if (need_downward_RA)
      alt_sep = DOWNWARD_RA;
    else
      alt_sep = UNRESOLVED;
  }
  return alt_sep;
}
```

Listing 1.1: The main function of the TCAS program [78]

Listing 1.1 shows the original source code of the main function of the `TCAS` program. Listing 1.2 shows some example first order mutants generated for this function. In Listing 1.2, the mutants *FOM* 1, *FOM* 2 and *FOM* 4 are generated by negation of a logical expression, while mutant *FOM* 3 is generated by replacing the '==' operator with the '! =' operator.

```
FOM 1:
17     need_downward_RA = !(Non_Crossing_Biased_Descend()
18       && Own_Above_Threat());

FOM 2:
19     if (!(need_upward_RA && need_downward_RA))

FOM 3:
8 tcas_equipped = Other_Capability != TCAS_TA;

FOM 4:
13      && intent_not_known) || !(!tcas_equipped)))
```

Listing 1.2: Four examples of first order mutants

Based on the types of faults seeded, mutants can be classified as First Order Mutants (FOMs) and Higher Order Mutants (HOMs). First order mutants seed only simple faults, generated by a single syntactic change to the original program. For example, *FOM* 1, *FOM* 2, *FOM* 3 and *FOM* 4 in Listing 1.2 are first order mutants. Higher order mutants combine simple first order faults to simulate more complex faults. For example, an higher order mutant can be created by combining any two or more first order mutants in Listing 1.2. Historically, mutation testing was always concerned with first order mutants [68, 114], because of the exponential number of higher order mutants that can be generated from first order mutants.

To assess the quality of a given test set, the latter should be executed against the generated mutants. If the result of running a mutant is different from the result of running the original program for any test in the input test set, then the mutant is said to be 'killed', otherwise it is said to have 'survived'. One outcome of the mutation testing process is the mutation score, which is the ratio of the number of detected faults over the total number of the seeded faults and, therefore, indicates the quality of the input test set.

Mutation testing has a wide range of applications in software testing. It can be used for testing software at the unit level, the integration level and the specification level. Mutation testing has been applied to many programming languages as a white box unit test technique, for example it has been used in Fortran programs [43, 39, 163, 4, 202, 148], Ada programs [31, 214], C programs [6, 60, 246, 270, 245, 104, 268], Java programs [144, 143, 145, 146, 48, 49, 167, 168], C# programs [72, 73, 74, 75, 76], SQL code [46, 264, 265, 244] and AspectJ programs [12, 13, 18, 97]. Mutation testing has also been used for integration testing [60, 62, 58, 59]. Besides using mutation testing at the software implementation level, it has also been applied at the design level to test the specifications or models of a program. For example, at the design level, mutation testing has been applied to Finite State Machines [95, 125, 21, 29], State Charts [101, 262, 289], Estelle Specifications [253, 254], Petri Nets [93], Network Protocols [269, 140, 248, 230], Security Policies [157, 172, 187, 186, 229] and Web Services [159, 215, 288, 160, 158, 276].

1.1.1 Problems with Mutation Testing

Although mutation testing is able to effectively assess the quality of test sets, it still suffers from certain problems, such as the following detailed below.

1. **High computational cost**. A major factor inhibiting mutation testing from becoming a practical testing technique is the high computational cost of executing mutants against test sets. This is due to the large number of mutants generated from even simple programs. For example, it is easy to generate 266 first order mutants for the `TCAS` program, which is approximately 100 lines of code, and these mutants need to be executed against 1608 tests.

2. **Trivial mutants**. Traditional mutation testing has only applied first order mutants. However, many of the first order mutants generated by these simple syntactic fault insertions are readily killed by the simplest of test cases executed, leading to much wasted effort killing rather trivial mutants [11]. As a result, many mutation testers observe that even the most trivial, small and unimaginative test suite will kill a very large proportion of the first order mutants.

3. **Equivalent mutants (human effort)**. If a mutant and the original program are semantically identical, then the mutant is said to be 'equivalent'; no test case can kill it. Equivalent mutants are a problem for mutation testing, because equivalence is undecidable, making it difficult to ascertain whether a surviving mutant is killable, as demonstrated by Budd and Angluin [38]. Thus the detection of equivalent mutants typically involves additional human effort.

4. **Oracle (human effort)**. The human oracle problem [278] refers to the process of checking the original program's output with each test case. Strictly speaking, this is not a problem unique to mutation testing. In all forms of testing, once a set of inputs has been arrived at, there remains the problem of checking the output [278]. However, mutating testing is effective precisely because it is a demanding test, and this can lead to an increase in the number of test cases, thereby increasing the oracle cost. The oracle cost is often the most expensive part of the overall test activity.

Although it is impossible to solve these problems completely, with existing advances in mutation testing, the process of mutation testing can be automated, and the run-time can allow for reasonable scalability, as the next chapter will show. This thesis will focus on the investigation of higher order mutants and apply them to ease some of the problems discussed here.

1.2 Higher Order Mutation as Solution

The view of mutation testing as a process of inserting a single fault into a program under test is established firmly in the literature [68, 4, 196, 197]. This view also pervades the collective subconsciousness of the research community. It is widely believed that higher order mutants are far too numerous to be practical as a source of simulated faults [68, 4]. Furthermore, many might claim that the coupling effect means that higher order mutants are most likely to be unimportant because they are all coupled to first order mutants [196, 197].

However, recent empirical research on fault localisation [231, 88] suggests that many subtle faults are more like higher order mutants in real world programs. Purushothaman and Perry [231] found that 90% of post release faults are, in fact, complex faults in the large AT&T switch system. These complex faults can only be fixed by several changes to the syntax of the program at several different places, which can be represented easily by higher order mutants. Similarly, Eldh et al. [88] found that in the Ericsson middleware more than 50% of the faults are complex. Therefore, it can be hypothesised that some of higher order mutants actually represent subtle faults, and they may be potentially useful in mutation testing.

Table 1.1 shows two examples of such interesting higher order mutants. The higher order mutant HOM 1 is created by combining two first order mutants FOM 1 and

Table 1.1: Examples of interesting higher order mutants

Mutants	Changes	No. of Killing Tests	Why interesting?
FOM 1	see Listing 1.2	886	*N/A*
FOM 2	see Listing 1.2	269	*N/A*
HOM 1	*FOM* 1 + *FOM* 2	125	It is much harder to kill *HOM* 1 than *FOM* 1 and *FOM* 2, and any of the tests which kills *HOM* 1 can kill both *FOM* 1 and *FOM* 2, but not *vice versa*.
FOM 3	see Listing 1.2	224	*N/A*
FOM 4	see Listing 1.2	260	*N/A*
HOM 2	*FOM* 3 + *FOM* 4	40	It is much harder to kill *HOM* 2 than *FOM* 3 or *FOM* 4, and none of the tests which kill *FOM* 3 and *FOM* 4 can kill *HOM* 2.

FOM 2. As shown in Table 1.1, *FOM* 1 is killed by 886 tests, and *FOM* 2 is killed by 269 tests. However, *HOM* 1 is killed by 125 tests only, and, thus, it is much more difficult to kill than either *FOM* 1 or *FOM* 2. This is caused by the fact that the two faults represented by *FOM* 1 and *FOM* 2 mask each other. As shown in Listing 1.2, *FOM* 1 introduces a fault which negates the value of the variable 'need_downward_RA' on line 17. However, the effect of this fault is partially masked by the fault in *FOM* 2 which negates the logical expression 'need_upward_RA && need_downward_RA'. Clearly *HOM* 1 requires fewer tests to be killed than *FOM* 1 and *FOM* 2 and is, thus, more subtle. Such higher order mutants might have survived if only first order mutants were considered when generating test cases.

HOM 1 is also a strongly subsuming higher order mutant; a strongly subsuming higher order mutant is killed only by a subset of the intersection of test cases that kill each first order mutant from which it is constructed. On examination of all 125 tests that kill *HOM* 1, it was found that any of these tests are guaranteed to kill both *FOM* 1 and *FOM* 2. Therefore, *HOM* 1 can replace *FOM* 1 and *FOM* 2 without loss of test effectiveness. However, this is not true the other way round. There exist test sets that kill *FOM* 1 and *FOM* 2, but which fail to kill *HOM* 1. The mutants *FOM* 1 and *FOM* 2 cannot, even taken collectively, replace the higher order mutant without possible loss of test effectiveness.

In Table 1.1, the higher order mutant *HOM* 2 is created by combining the two first order mutants *FOM* 3 and *FOM* 4 in Listing 1.2. Similar to *HOM* 1, *HOM* 2 is also more difficult to kill than each first order mutant from which it is constructed. However, none of the tests which kill *FOM* 3 or *FOM* 4 can kill *HOM* 2. This means that after combining the faults represented by *FOM* 3 and *FOM* 4, the original faulty behaviour disappears, and new faulty behaviour is unveiled; we call this "fault-shifting". As shown in Listing 1.2, *FOM* 3 negates the value of the variable 'tcas_equipped', and *FOM* 4 negates 'tcas_equipped' again, which completely masks the fault effect introduced by *FOM* 3. However, as 'tcas_equipped' is also used in the same predicate on line 12. So the higher order mutant is not an equivalent mutant. Rather, such complex interaction introduces some new fault behaviour. Apparently, *HOM* 2 requires additional new test data to kill it. As a result, such higher order mutants should be also considered in mutation testing.

Combinations of faults such as those described above are relatively rare. As one might expect, adding more faults to a faulty program tends to make it more likely that the program will fail and, therefore, more likely that testing will reveal the presence of a fault. However, the rare exceptions to this rule are very interesting and, it can be argued, valuable. It is possible that applying these valuable higher order mutants can ease the four problems of mutation testing:

1. **High computational cost problem**. Higher order mutation addresses this by reducing the number of mutants by using strongly subsuming higher order mutants to replace all of the first order mutants from which they are constructed.

2. **Trivial mutant problem**. Higher order mutation addresses this by applying subtle fault-like higher order mutants in mutation testing, such as the fault-masking and fault-shifting higher order mutants.

3. **Equivalent mutant problem**. Higher order mutation cannot address this problem directly. However, there is some evidence to suggest that higher order mutants may be less likely to be equivalent than first order mutants [197]. Furthermore, using a co-evolutionary approach can also avoid generation of equivalent mutants, which is described in the future work section in Chapter 7.

4. **Oracle problem**. Higher order mutation addresses this by reducing the number of tests required by reducing the number of mutants with strongly subsuming higher order mutants.

1.3 Problems of the Thesis

To date, work in the field of mutation testing has largely focused on first order mutants [1]. There are two primary reasons for not considering higher order mutants. The first reason is the coupling-effect hypothesis [68, 197], which suggests that it is unlikely that higher order mutants will be found that are not coupled to first order mutants. Therefore, any increase in test effectiveness that accrues from higher order mutant testing will surely be minor. The second reason is that there are already a

[1]Not considering those paper published on higher order mutation after the work of this thesis started.

large number of first order mutants. This compounds the existing problem of high computational cost. There are exponentially more higher order mutants, so moving to higher order mutant testing will surely exacerbate an already difficult problem. These barriers present new challenges to the research on higher order mutant testing:

1. Are there any interesting higher order mutants that might be potentially useful to mutation testing?
2. If they exist, how can these interesting higher order mutants be found effectively?
3. How can test data be generated to kill these interesting higher order mutants?

In this thesis, higher order mutants will be classified from a fault interaction standpoint. This allows the identification of some categories of higher order mutant that represent real subtle faults, which should be also applied in mutation testing. In order to overcome the inherent computational cost that comes with the large number of higher order mutants, this thesis will introduce a search-based optimisation approach to identify valuable higher order mutants efficiently.

1.4 Aims and Objectives

Some valuable higher order mutants might be potentially better able to simulate real faults and to reveal insights into bugs than the restricted class of first order mutants. However, the mutation testing community has previously avoided working on higher order mutation testing, considering it to be too computationally expensive and, therefore, impractical. The general aim of the thesis is *to make higher order mutant testing applicable and practical using a search process that seeks fit mutants (both first and higher order) from the space of all possible mutants.*

The detailed aims and objectives of this thesis are as follows:

1. To investigate higher order mutants from a fault interaction standpoint.
2. To apply search-based optimisation approaches to locate very fit mutants (both first and higher order) within the search space of all possible mutants and to investigate empirically the higher order mutants found by the algorithms.
3. To extend the current state-of-the-art mutant-based test data generation techniques to handle higher order mutants and to evaluate this extended test data generation approach on both first order mutants and higher order mutants.

1.5 Contributions of the Thesis

The contributions of this thesis are:

1. A proposal of a practical higher order mutation testing paradigm, which applies valuable higher order mutants and non-trivial first order mutants together in mutation testing.
2. An investigation and classification of various kinds of higher order mutants.
3. A proposal of a search-based optimisation approach for finding optimal higher order mutants, which overcomes the exponential increase in the number of higher order mutants.
4. A proposal of a mutation-based test data generation approach, which combines dynamic symbolic execution and search-based software testing for strongly killing both first order mutants and higher order mutants.

5. An empirical study exploring higher order mutants based on the new categorisation. The results show that interesting higher order mutant categories, such as fault masking and fault shifting, are very common. They exist in all the subjects under study.

6. An empirical study applying the proposed search-based algorithm exploring the proportion of all higher order mutants that are subsuming and strongly subsuming. The results show that a small proportion of higher order mutants are subsuming, and only a small but useful proportion of them are strongly subsuming.

7. An empirical study which demonstrates that the hybrid approach achieved higher mutation adequacy than two recent mutation-based test data generation approaches.

8. Milu, an open-source fully-featured mutation testing tool, which handles both first and higher order mutation for C programs.

9. Several comprehensive trend analyses of the approaches, tools and developments, applications and empirical studies of mutation testing. These analyses provide evidence that mutation testing techniques and tools are reaching a state of maturity and applicability.

1.6 Organisation of the Thesis

Chapter 2 surveys the literature on mutation testing. It begins by introducing the two fundamental hypotheses of mutation testing followed by a discussion of the general process of mutation testing. The chapter then moves on to examine three main research topics in the field: cost reduction for mutation analysis, mutation-based test data generation and equivalent mutation detection.

Chapter 3 provides a comprehensive analysis of the approaches, tools and developments and empirical results of mutation testing. The chapter presents the results of several development trend analyses. These analyses provide evidence that mutation testing techniques and tools are reaching a state of maturity and applicability, while the area of mutation testing is the subject of increasing interest in its own right.

Chapter 4 introduces a new classification of higher order mutants from a fault interaction standpoint. The chapter also describes an open-source C mutation testing tool which can generate both first and higher order mutants. The chapter then investigates the proportion of different categories of all second order mutants and samples of third to fifth order mutants in six subject programs. In total, more than two million higher order mutants were generated with 9.2 billion test executions. The results demonstrate that a large proportion of the behaviour of higher order mutants is changed due to fault interaction.

Chapter 5 introduces the higher order mutation testing paradigm with the concept of subsuming higher order mutants. The chapter describes three search-based algorithms for searching subsuming higher order mutants. The chapter, then, explores the proportion of all higher order mutants that are subsuming and strongly subsuming. The results show that a small but useful proportion of higher order mutants are subsuming, and that a small proportion of these are strongly subsuming. Although the proportion of strongly subsuming mutants is small, the number of strongly subsuming mutants is large, because the number of higher order mutants increases exponentially. The search-based algorithms were able to find small but useful numbers of strongly subsuming higher order mutants in all of the ten programs studied.

Chapter 6 introduces SHOM, a mutation-based test data generation approach that combines dynamic symbolic execution and search-based software testing. SHOM targets strong mutation adequacy and is capable of killing both first and higher order mutants. The chapter reports the results of an empirical study using 17 programs. SHOM achieved higher strong mutation adequacy than two recent mutation-based test data generation approaches.

Chapter 7 concludes the thesis and provides possible directions for future research.

Chapter 2

Literature Survey

This chapter reviews work in the field of mutation testing. It begins with an introduction of the two fundamental hypotheses of mutation testing followed by a discussion of the general process of mutation testing. The chapter then describes two types of cost reduction techniques, mutants reduction and execution cost reduction. It then moves on to the description of equivalent mutation detection techniques. At the end, it introduces different applications of mutation testing and empirical experiments of the research work on mutation testing.

2.1 The Theory of Mutation Analysis

2.1.1 Fundamental Hypotheses

Mutation testing promises to be effective in identifying adequate test data which can be used to find real faults [103]. However, the number of such potential faults for a given program is enormous; it is impossible to generate mutants representing all of them. Therefore, traditional Mutation Testing targets only a subset of these faults, those which are close to the correct version of the program, with the hope that these will be sufficient to simulate all faults. This theory is based on two hypotheses: the Competent Programmer Hypothesis (CPH) [68, 4] and Coupling Effect [68].

The CPH was first introduced by DeMillo et al. in 1978 [68]. It states that programmers are competent, which implies that they tend to develop programs close to the correct version. As a result, although there may be faults in the program delivered by a competent programmer, we assume that these faults are merely a few simple faults which can be corrected by a few small syntactical changes. Therefore, in Mutation Testing, only faults constructed from several simple syntactical changes are applied, which represent the faults that are made by "competent programmers". An example of the CPH can be found in Acree et al.'s work [4]. A theoretical discussion using the concept of program neighbourhoods can also be found in Budd et al.'s work [40].

The Coupling Effect was also proposed by DeMillo et al. in 1978 [68]. Unlike the CPH concerning a programmer's behaviour, the Coupling Effect concerns the type of faults used in mutation analysis. It states that "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors". Offutt [196, 197] extended this into the Coupling Effect Hypothesis and the Mutation Coupling Effect Hypothesis with a precise definition of simple and complex faults (errors). In his definition, a simple fault is represented by a simple mutant which is created by making a single syntactical change, while a complex fault is represented as a complex mutant which is created by making more than one change.

According to Offutt, the Coupling Effect Hypothesis is that "complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults "[197]. The Mutation Coupling Effect Hypothesis now becomes "Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants [197]". As a result, the mutants used in traditional Mutation Testing are limited to simple mutants only.

There has been much research work on the validation of the coupling effect hypothesis [163, 196, 197, 183]. Lipton and Sayward [163] conducted an empirical study using a small program, FIND. In their experiment, a small sample of 2nd-order, 3rd-order and 4th-order mutants is investigated. The results suggested that an adequate test set generated from 1st-order mutants was also adequate for the samples of kth-order mutants ($k = 2, ..., 4$). Offutt [196, 197] extended this experiment using all possible 2nd-order mutants with two more programs, MID and TRITYP. The results suggested that test data developed to kill 1st-order mutants killed over 99% 2nd-order and 3rd-order mutants. This study implied that the mutation coupling effect hypothesis does, indeed manifest itself in practice. Similar results were found in the empirical study by Morell [183].

The validity of the mutation coupling effect has also been considered in the theoretical studies of Wah [273, 274, 275] and Kappoor [142]. In Wah's work [274, 275], a simple theoretical model, the q function model was proposed which considers a program to be a set of finite functions. Wah applied test sets to the 1st-order and the 2nd-order model. Empirical results indicated that the average survival ratio of 1st-order mutants and 2nd-order mutants is $1/n$ and $1/n^2$ respectively where n is the order of the domain [274]. This result is also similar to the estimated results of the empirical studies mentioned above. A formal proof of the coupling effect on the boolean logic faults can be also found in Kappoor's work [142].

2.1.2 The Process of Mutation Analysis

The traditional process of mutation analysis is illustrated in Figure 2.1. In mutation analysis, from a program p, a set of faulty programs p' called mutants, is generated by a few single syntactic changes to the original program p. As an illustration, Table 2.1 shows the mutant p', generated by changing the *and* operator (&&) of the original program p, into the *or* operator (||), thereby producing the mutant p'.

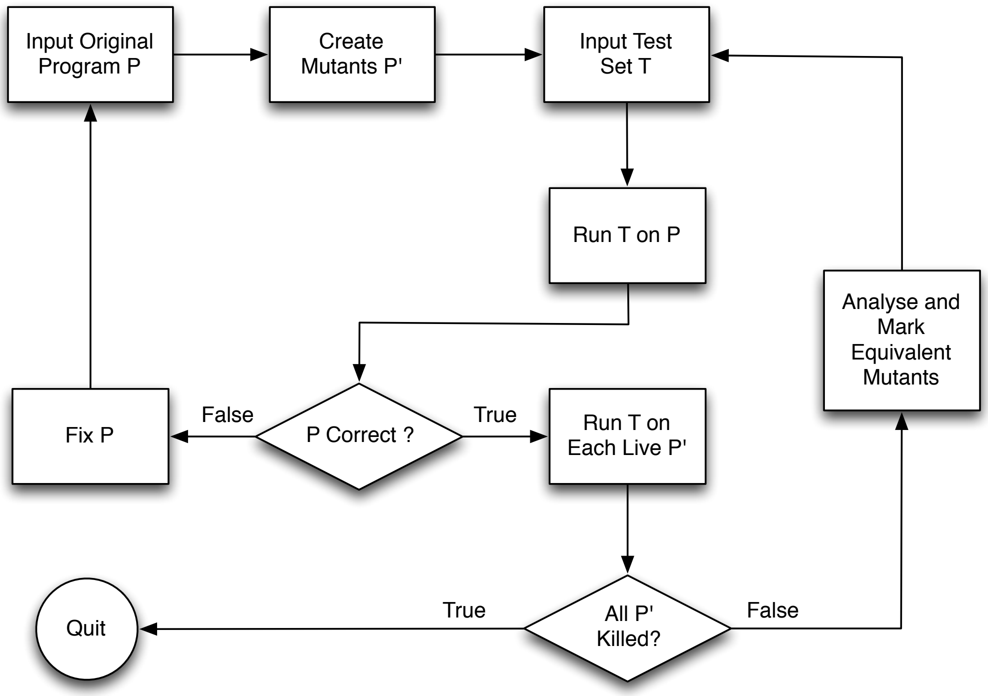

Figure 2.1: Generic Process of Mutation Analysis [213]

A transformation rule that generates a mutant from the original program is known as a mutation operator[1]. Table 2.1 contains only one example of a mutation operator; there are many others. Typical mutation operators are designed to modify variables and expressions by replacement, insertion or deletion operators. Table 2.2 lists the first set of formalised mutation operators for the Fortran programming language.

[1]In the literature of mutation testing, mutation operators are also known as mutant operators, mutagenic operators, mutagens and mutation rules [213].

Table 2.1: A Example of Mutation Operation

Program p	Mutant p'
...	...
if ($a > 0$ && $b > 0$)	if ($a > 0$ || $b > 0$)
return 1;	return 1;
...	...

These typical mutation operators were implemented in the Mothra mutation system [148].

To increase the flexibility of mutation testing in practical applications, Simao et al. [249] also proposed a transformation language, MuDel, used to specify the description of mutation operators. Besides modifying program source, mutation operators can also be defined as rules to modify the grammar used to capture the syntax of a software artefact. A much more detailed account of these grammar-based mutation operators can be found in the work of Offutt et al. [199].

In the next step, a test set T is supplied to the system. Before starting the mutation analysis, this test set needs to be successfully executed against the original program p to check its correctness for the test case. If p is incorrect, it has to be fixed before running other mutants, otherwise each mutant p' will then be run against this test set T. If the result of running p' is different from the result of running p for any test case in T, then the mutant p' is said to be 'killed', otherwise it is said to have 'survived'.

After all test cases have been executed, there may still be a few 'surviving' mutants. To improve the test set T, the program tester can provide additional test inputs to kill these surviving mutants. However, there are some mutants that can never be killed, because they always produce the same output as the original program.

Table 2.2: The first set of mutation operators: The 22 “Mothra” Fortran Mutation Operators (adapted from [148])

Mutation Operators	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

These mutants are called Equivalent Mutants. They are syntactically different but functionally equivalent to the original program. Automatically detecting all equivalent mutants is impossible [38, 208], because program equivalence is undecidable. The equivalent mutant problem has been a barrier that prevents mutation testing from being more widely used. Several proposed solutions to the equivalent mutant problem are discussed in Section 2.3.

Mutation Testing concludes with an adequacy score, known as the Mutation Score, which indicates the quality of the input test set. The mutation score (MS) is the ratio of the number of killed mutants over the total number of non-equivalent mutants. The goal of mutation analysis is to raise the mutation score to 1, indicating the test set T is sufficient to detect all the faults denoted by the mutants.

2.2 Cost Reduction Techniques

Mutation testing is widely believed to be a computationally expensive testing technique. However, this belief is partly based on the outdated assumption that all mutants in the traditional Mothra set need to be considered. In order to turn mutation testing into a practical testing technique, many cost reduction techniques have been proposed. In the survey work of Offutt and Untch [213], cost reduction techniques are divided into three types: 'do fewer', 'do faster' and 'do smarter'. In this paper, these techniques are classified into two types, reduction of the generated mutants (which corresponds to 'do fewer') and reduction of the execution cost (which combines do faster and do smarter). The rest of the section will introduce each cost reduction technique in detail. Section 2.2.1 will present work on mutant reduction techniques, while Section 2.2.2 will cover execution reduction techniques.

2.2.1 Mutant Reduction Techniques

One of the major sources of computational cost in mutation testing is the inherent running cost in executing the large number of mutants against the test set. As a result, reducing the number of generated mutants without significant loss of test effectiveness has become a popular research problem. For a given set of mutants, M, and a set of test data T, $MS_T(M)$ denotes the mutation score of the test set T applied to mutants M. The mutant reduction problem can be defined as the problem of finding a subset of mutants M' from M, where $MS_T(M') \approx MS_T(M)$. This section will introduce four techniques used to reduce the number of mutants, Mutant Sampling, Selective Mutation and Mutant Clustering.

Mutant Sampling

Mutant Sampling is a simple approach that randomly chooses a small subset of mutants from the entire set. This idea was first proposed by Acree [3] and Budd [37]. In this approach, all possible mutants are generated first as in traditional mutation testing. $x\%$ of these mutants are then selected randomly for mutation analysis and the remaining mutants are discarded. There were many empirical studies of this approach. The primary focus was on the choice of the random selection rate (x). In Wong and Mathur's studies [279, 177], the authors conducted an experiment using a random selection rate $x\%$ from 10% to 40% in steps of 5%. The results suggested that random selection of 10% of mutants is only 16% less effective than a full set of mutants in terms of mutation score. This study implied that Mutant Sampling is valid with a $x\%$ value higher than 10%. This finding also agreed with the empirical studies by DeMillo et al. [66] and King and Offutt[148]. Instead of fixing the sample rate, Sahinoglu and Spafford [237] proposed an alternative sampling approach based on the Bayesian sequential probability ratio test (SPRT). In their approach, the mutants are randomly selected until a statistically appropriate sample size has been reached. The result suggested that their model is more sensitive than the random selection because it is self-adjusting based on the available test set. A more recent empirical study by Zhang et al. [292] suggested that randomly sampling mutants can still achieve competitive results compared to other selective mutation techniques.

Selective Mutation

A reduction in the number of mutants can also be achieved by reducing the number of mutation operators applied. This is the basic idea, underpinning Selective Mutation, which seeks to find a small set of mutation operators that generate a subset of all possible mutants without significant loss of test effectiveness. This idea was first suggested as "constrained mutation" by Mathur [174]. Offutt et al. [212] subsequently extended this idea calling it Selective Mutation.

Mutation operators generate different numbers of mutants and some mutation operators generate far more mutants than others, many of which may turn out to be redundant. For example, two mutation operators of the 22 Mothra operators, ASR and SVR, were reported to generate approximately 30% to 40% of all mutants [148]. To effectively reduce the generated mutants, Mathur [174] suggested omitting two mutation operators ASR and SVR which generated most of the mutants. This idea was implemented as "2-selective mutation" by Offutt et al. [212].

Offutt et al. [212] have also extended Mathur and Wong's work by omitting four mutation operators (4-selective mutation) and omitting six mutation operators (6-selective mutation). In their studies, they reported that 2-selective mutation achieved a mean mutation score of 99.99% with a 24% reduction in the number of mutants reduced. 4-selective mutation achieved a mean mutation score of 99.84% with a 41% reduction in the number of mutants. 6-selective mutation achieved a mean mutation score of 88.71% with a 60% reduction in the number of mutants.

Wong and Mathur adopted another type of selection strategy, selection based on test effectiveness [279, 283], known as constraint mutation. Wong and Mathur suggested using only two mutation operators: ABS and RAR. The motivation for the ABS operator is that killing the mutants generated from ABS requires test cases from different parts of the input domain. The motivation for the ROR operator is that killing the mutants generated from ROR requires test cases which 'examine' the mutated predicate [279, 283]. Empirical results suggest that these two mutation operators achieve an 80% reduction in the number of mutants and only 5% reduction in the mutation score in practice.

Offutt et al. [203] extended their 6-selective mutation further using a similar selection strategy. Based on the type of the Mothra mutation operators, they divided them into three categories: statements, operands and expressions. They tried to omit operators from each class in turn. They discovered that 5 operators from the operands and expressions class became the key operators. These 5 operators are ABS, UOI, LCR, AOR and ROR. These key operators achieved 99.5% mutation score.

Mresa and Bottaci [188] proposed a different type of selective mutation. Instead of trying to achieve a small loss of test effectiveness, they also took the cost of detecting equivalent mutants into consideration. In their work, each mutation operator is assigned a score which is computed by its value and cost. Their results indicated that it was possible to reduce the number of equivalent mutants while maintaining effectiveness.

Based on previous experience, Barbosa et al. [20] defined a guideline for selecting a sufficient set of mutation operators from all possible mutation operators. They applied this guideline to Proteum's 77 C mutation operators [6] and obtained a set of

10 selected mutation operators, which achieved a mean mutation score of 99.6% with a 65.02% reduction in the number of mutants. They also compared their operators with Wong's and Offutt et al.'s set. The results showed their operator set achieved the highest mutation score.

The most recent research work on selective mutation was conducted by Namin et al. [189, 190, 191]. They formulated the selective mutation problem as a statistical problem: the variable selection or reduction problem. They applied linear statistical approaches to identify a subset of 28 mutation operators from 108 C mutation operators. The results suggested that these 28 operators are sufficient to predict the effectiveness of a test suite and it reduced 92% of all generated mutants. According to their results, this approach achieved the highest rate of reduction compared with other approaches.

Mutant Clustering

The idea of Mutant Clustering was first proposed in Hussain's masters thesis [130]. Instead of selecting mutants randomly, Mutant Clustering chooses a subset of mutants using clustering algorithms. The process of Mutation Clustering starts from generating all first order mutants. A clustering algorithm is then applied to classify the first order mutants into different clusters based on the killable test cases. Each mutant in the same cluster is guaranteed to be killed by a similar set of test cases. Only a small number of mutants are selected from each cluster to be used in mutation testing, the remaining mutants are discarded. In Hussain's experiment, two clustering algorithms, K-means and Agglomerative clustering were applied and the result was compared with random and greedy selection strategies. Empirical results suggest that Mutant Clustering is able to select fewer mutants but still maintain the mutation score. A development of the Mutant Clustering approach can be found in the work of Ji et al. [134]. Ji et al. use a domain reduction technique to avoid the need to execute all mutants.

2.2.2 Execution Cost Reduction Techniques

In addition to reducing the number of generated mutants, the computational cost can also be reduced by optimising the mutant execution process. This section will introduce the three types of techniques used to optimise the execution process that have been considered in the literature.

Strong, Weak and Firm Mutation

Based on the way in which we decide whether to analyse if a mutant is killed during the execution process, mutation testing techniques can be classified into three types, Strong Mutation, Weak Mutation and Firm Mutation.

Strong Mutation is often referred to as traditional Mutation Testing. That is, it is the formulation originally proposed by DeMillo et al. [68]. In Strong Mutation, for a given program p, a mutant m of program p is said to be killed only if mutant m gives a different output from the original program p.

To optimise the execution of the Strong Mutation, Howden [129] proposed Weak Mutation. In Weak Mutation, a program p is assumed to be constructed from a set of components $C = \{c_1, ..., c_n\}$. Suppose mutant m is made by changing component c_m, mutant m is said to be killed if any execution of component c_m is different from mutant m. As a result, in Weak Mutation, instead of checking mutants after the execution of the entire program, the mutants need only to be checked immediately after the execution point of the mutant or mutated component.

In Howden's work [129], the component C referred to one of the following five types: variable reference, variable assignment, arithmetic expression, relational expression and boolean expression. This definition of components was later refined by Offutt and Lee [205, 204]. Offutt and Lee defined four types of execution: evaluation after the first execution of an expression (Ex-Weak/1), the first execution of a statement (St-Weak/1), the first execution of a basic block (BB-Weak/1) and after N iterations of a basic block in a loop ((BB-Weak/N).

The advantage of weak mutation is that each mutant does not require a complete execution process; once the mutated component is executed we can check for survival. Moreover, it might not even be necessary to generate each mutant, as the constraints for the test data can sometimes be determined in advance [284]. However, as different components of the original program may give different outputs from the original execution, weak mutation test sets can be less effective than strong mutation test sets. In this way, weak mutation sacrifices test effectiveness for improvements in test effort. This raises the question as to what kind of trade-off can be achieved.

There were many empirical studies on the Weak Mutation trade off. Girgis and Woodward [110] implemented a weak mutation system for Fortran 77 programs. Their system is an analytical type of weak mutation system in which the mutants are killed by examining the program's internal state. In their experiment, four of Howden's five program components were considered. The results suggested that weak mutation is less computationally expensive than strong mutation. Marick [171] drew similar conclusions from his experiments.

A theoretical proof of Weak Mutation by Horgan and Mathur [127] showed that under certain conditions, test sets generated by weak mutation can also be expected to be as effective as strong mutation. Offutt and Lee [205, 204] presented a comprehensive empirical study using a weak mutation system named Leonardo. In their experiment, they used the 22 Mothra mutation operators as fault models instead of Howden's five component set. The results from their experiments indicated that Weak Mutation is an alternative to Strong Mutation in most common cases, agreeing with the probabilistic results of Horgan and Mathur [127] and experimental results of Girgis and Woodward [110] and Marick[171]. The most recent work on the weak mutation was conducted by Durelli et al. [86]. They extended a Java virtual machine to support weak mutation analysis. Experimental results show that the virtual machine-based implementation achieves speedups of more than 80% compared to traditional strong mutation.

Firm Mutation was first proposed by Woodward and Halewood [287]. The idea of Firm Mutation is to overcome the disadvantages of both weak and strong mutations by providing a continuum of intermediate possibilities. That is, the 'compare state' of Firm Mutation lies between the intermediate states after execution (Weak Mutation) and the final output (Strong Mutation). In 2001, Jackson and Woodward [133] an approach to Firm Mutation that executes Java mutants in parallel. Recently, Mateo et al. [173] proposed a improved Firm Mutation-based approach called Flexible Weak Mutation. Instead of using a fixed pre-defined intermediate state, Flexible Weak Mutation chooses the comparing state dynamically after the execution point. Moreover, to kill a mutant, Flexible Weak Mutation requires multiple differences in the intermediate states found. This approach has been implemented into a Java mutation testing tool, Bacterio [173].

Run-time Optimisation Techniques

The Interpreter-Based Technique is one of the optimisation techniques used in the first generation of Mutation Testing tools [202, 148]. In traditional Interpreter-Based Techniques, the result of a mutant is interpreted from its source code directly. The main cost of this technique is determined by the cost of interpretation. To optimise the traditional Interpreter-Based approach, Offutt and King [202, 148] translated the original program into an intermediate form. Mutation and interpretation are performed at this intermediate code level. Interpreter-Based tools provide additional flexibility and are sufficiently efficient for mutating small programs. However, due to the nature of interpretation, it becomes slower as the scale of programs under test increases.

The Compiler-Based Technique is the most common approach to achieve program mutation [56, 57]. In a Compiler-Based Technique, each mutant is first compiled into an executable program; the compiled mutant is then executed by a number of test cases. Compared to source code interpretation techniques, this approach is much faster because execution of compiled binary code takes less time than interpretation. However, there is also a speed limitation, known as compilation bottleneck, due to the high compilation cost for programs whose run-time is much longer than the compilation/link time. [50].

DeMillo et al. proposed the Compiler-Integrated Technique [67] to optimise the performance of the traditional Compiler-Based Techniques. Because there is only a minor syntactic difference between each mutant and the original program, compiling each mutant separately in the Compiler-Based technique will result in redundant compilation cost. In the Compiler-Integrated technique, an instrumented compiler is designed to generate and compile mutants.

The instrumented compiler generates two outputs from the original program: an executable object code for the original program and a set of patches for mutants. Each patch contains instructions which can be applied to convert the original executable object code image directly to executable code for a mutant. As a result, this technique can effectively reduce the redundant cost from individual compilation. A much more detailed account can be found in the Krauser's PhD thesis [150]. Recently, Just et al. [141] extended this technique and implemented it into the Java Standard Edition compiler.

The Mutant Schema Generation approach is also designed to reduce the overhead cost of the traditional interpreter-based techniques [266, 268, 267]. Instead of compiling each mutant separately, the mutant schema technique generates a metaprogram. Just like a 'super mutant' this metaprogram can be used to represent all possible mutants. Therefore, to run each mutant against the test set, only this metaprogram need be compiled. The cost of this technique is composed of a one-time compilation cost and the overall run-time cost. As this metaprogram is a compiled program, its running speed is faster than the interpreter-based technique. The results from Untch et al.'s work [268] suggest that the mutant schema prototype tool, TUMS, is significantly faster than Mothra using interpreter techniques. Much more extensive results are reported in detail in the Untch's PhD dissertation [267]. A similar idea of the Mutant Schemata technique, named the Mutant Container, was proposed by Mathur independently. The details can be found in a software engineering course 'handout' by Mathur [175].

The most recent work on reduction of the compilation cost is the Bytecode Translation Technique. This technique was first proposed by Ma et al. [206, 168]. In Bytecode Translation, mutants are generated from the compiled object code of the original program, instead of the source code. As a result, the generated 'bytecode mutants' can be executed directly without compilation. As well as saving compilation cost, Bytecode Translation can also handle off-the-shelf programs which do not have available source code. This technique has been adopted in the Java programming language [206, 168, 169, 238]. However, not all programming languages provide an easy way to manipulate intermediate object code. There are also some limitations for the application of Bytecode Translation in Java, such as not all the mutation operators can be represented at the Bytecode level [238].

Bogacki and Walter introduced an alternative approach to reduce compilation cost, called Aspect-Oriented Mutation [28, 27]. In their approach, an aspect patch is generated to capture the output of a method on the fly. Each aspect patch will run programs twice. The first execution obtains the results and context of the original program and mutants are generated and executed in the second execution. As a result, there is no need to compile each mutant. Empirical evaluation between a prototype tool and Jester can be found in the work of Bogacki and Walter [27].

Advanced Platforms Support for Mutation Testing

Mutation Testing has also been applied to many advanced computer architectures to distribute the overall computational cost among many processors. In 1988, Mathur and Krauser [176] were the first to perform Mutation Testing on a vector processor system. Krauser et al. [151, 152] proposed an approach for concurrent execution mutants under SIMD machines. Fleyshgakker and Weiss [98, 277] proposed an algorithm that significantly improved techniques for parallel mutation testing . Choi and Mathur [50] and Offutt et al. [211] have distributed the execution cost of mutation testing through MIMD machines. Zapf [290] extended this idea in a network environment, where each mutant is executed independently.

Higher order mutant testing is a "do fewer" but "smarter" approach. It can be considered as a form of selective mutation. However rather than selecting from first order mutants, higher order mutation testing specifically search and targets those HOMs, the strongly subsuming higher order mutants, each of which can be used to replace more than one first order mutant. Therefore fewer (but better) mutants are used in higher order mutation testing, which also leads to fewer (but better) test cases.

Table 2.3: A Example of Equivalent Mutation

Program p	Equivalent Mutant m
for $(int\ i = 0;\ i < 10;\ i++)$ $\{$ $...(the\ value\ of\ i$ $is\ not\ changed)$ $\}$	for $(int\ i = 0;\ i\ != 10; i++)$ $\{$ $...(the\ value\ of\ i$ $is\ not\ changed)$ $\}$

2.3 Equivalent Mutant Detection Techniques

To detect if a program and one of its mutants programs are equivalent is undecidable, as proved in the work of Budd and Angluin [38]. As a result, the detection of equivalent mutants alternatively may have to be carried out by humans. This has been a source of much theoretical interest. For a given program p, m denotes a mutant of program p. Recall that m is an equivalent mutant if m is syntactically different from p, but has the same behaviour with p. Table 2.3 shows an example of equivalent mutant generated by changing the operator $<$ of the original program into the operator $!=$. If the statements within the loop do not change the value of i, program p and mutant m will produce identical output.

An equivalent mutant is created when a mutation leads to no possible observable change in behaviour; the mutant is syntactically different but semantically identical to the original program from which it is created. Grün et al. [113] manually investigated eight equivalent mutants generated from the JAXEN XPATH query engine program. They pointed out four common equivalent mutant situations: the mutant is generated from dead code, the mutant improves speed, the mutant only alters the internal states and the mutant cannot be triggered (i.e. no input test data can change the program's behaviour at the mutation point). It is worth noticing that these four are not the only situations that lead to equivalent mutants. For example, none of it applies to the example in Table IV.

As the mutation score is counted based on non-equivalent mutants, without a complete detection of all equivalent mutants, the mutant score can never be 100%, which means the programmer will not have complete confidence in the adequacy of a potentially perfectly adequate test set. Empirical results indicate that there are 10% to 40% of mutants which are equivalent [208, 200]. Fortunately, there has been much research work on the detection of the equivalent mutants.

Baldwin and Sayward [19] proposed an approach that used compiler optimisation techniques to detect equivalent mutants. This approach is based on the idea that the optimisation procedure of source code will produce an equivalent program, so a mutant might be detected as equivalent mutants by either 'optimisation' or a 'de-optimisation process'. Baldwin and Sayward [19] proposed six types of compiler optimisation rules that can be used for the detection of equivalent mutants. These six were implemented and empirically studied by Offutt and Craft [200]. The empirical results showed that, generally, 10% of all mutants were equivalent mutants for 15 subject programs.

Based on the work of constraint test data generation, Offutt and Pan [208, 207, 221] introduced a new equivalent mutant detection approach using constraint solving. In their approach, the equivalent mutant problem is formulated as a constraint satisfaction problem by analysing the path condition of a mutant. A mutant is equivalent if and only if the input constraint is unsatisfiable. Empirical evaluation of a prototype has shown that this technique is able to detect a significant percentage of equivalent mutants (47.63% among 11 subject programs) for most of the programs. Their results suggest that the constraint satisfaction formulation is more powerful than the compiler optimisation technique [200].

The program slicing technique has also been proposed to assist in the detection of equivalent mutants [272, 124, 118]. Voas and McGraw [272] were the first to suggest the application of program slicing to Mutation Testing. Hierons et al. [124] demonstrated an approach using slicing to assist the human analysis of equivalent mutants. This is achieved by the generation of a sliced program that denotes the answer to an equivalent mutant. This work was later extended by Harman et al. [118] using dependence analysis.

Adamopoulos et al. [5] proposed a co-evolutionary approach to detect possible equivalent mutants. In their work, a fitness function was designed to set a poor fitness value to an equivalent mutant. Using this fitness function, equivalent mutants are wiped out during the co-evolution process and only mutants that are hard to kill and test cases that are good at detecting mutants are selected.

Ellims et al. [90] reported that mutants with syntactic difference and the same output can be also semantically different in terms of running profile. These mutants often have the same output as the original programs but have different execution time or memory usage. Ellims et al. suggested that 'resource-aware' might be used to kill the potential mutants.

A more recent work by Grün et al. [113, 239] investigated the impact of mutants. The impact of a mutant was defined as the different program behaviour between the original program and the mutant and it was measured through the code coverage in their experiment. The empirical results suggested that there was a strong correlation between mutant 'killability' and its impact on execution, which indicates that if a mutant has higher impact, it is less likely to be equivalent.

The most recent work on the equivalent mutants was conducted by Schuler and Zeller [240]. In [240], the authors studied the correlation between the impact of various mutations including impact on coverage, impact on return values and impact on invariants and their likelihood of producing equivalent mutants. The empirical results suggested that mutants with impact on coverage or return values are more likely to be non-equivalent, and more than 90% of highest coverage impact are non-equivalent mutants.

The thesis does not address the the equivalent mutants directly. However, higher order mutation testing can reduce the number of equivalent. Because there was comparatively low density of equivalent mutants found in the higher order paradigm, compared to that found in the first order paradigm [197].

2.4 Applications

Since mutation testing was proposed in the 1970s, it has been applied to test both program source code (Program Mutation) [64] and program specification (Specification Mutation) [112]. Program Mutation belongs to the category of white box based testing, in which faults are seeded into source code, while Specification Mutation belongs to black box based testing where faults are seeded into program specifications, but in which the source code may be unavailable during testing.

2.4.1 Program Mutation

Program Mutation has been applied to both the unit level [68] and the integration level [59] of testing. For unit level Program Mutation, mutants are generated to represent the faults that programmers might have made within a software unit, while for the integration level Program Mutation, mutants are designed to represent the integration faults caused by the connection or interaction between software units [271]. Applying Program Mutation at the integration level is also known as Interface Mutation which was first introduced by Delamaro et al. [59] in 1996. Interface

Mutation has been applied to C Programs by Delamaro et al. [59, 58, 60] and also to the CORBA Programs by Ghosh and Mathur [105, 108, 107, 109]. Empirical evaluations of Interface Mutation can be found in Vincenzei et al.'s work [271] and Delamaro et al.'s work [62, 61].

Mutation Testing for Fortran

In the earliest days of mutation testing, most of the experiments on mutation testing targeted Fortran. Budd et al. [43, 39] was the first to design mutation operators for Fortran IV in 1977. Based on these studies, a Mutation Testing tool named PIMS was developed for testing Fortran IV programs [39, 163, 4]. However, there were no formal definitions of mutation operators for Fortran until 1987. In 1987, Offutt and King [202, 148] summarized the results from previous work and proposed 22 mutation operators for Fortran 77. This set of mutation operators became the first set of formalized mutation operators and consequently had greater influence on later definitions of mutation operators for applying mutation testing to the other programming languages. These mutation operators are divided into three groups; the Statement analysis group, the Predicate analysis group and the Coincidental correctness group.

Mutation Testing for Ada

Ada mutation operators were first proposed by Bowser [31] in 1988. In 1997, based on previous work of Bowser's Ada mutation operators [31], Agrawal et al.'s C mutation operators [6] and the design of Fortran 77 mutation operators for MOTHRA [148], Offutt et al. [214] redesigned mutation operators for Ada programs to produce a proposed set of 65 Ada mutation operators. According to the semantics of Ada, this set of Ada mutation operators is divided into five groups: Operand Replacement Operators group, Statement Operators group, Expression Operators group, Coverage Operators group and Tasking Operators group.

Mutation Testing for C

In 1989, Agrawal et al. [6] proposed a comprehensive set of mutation operators for the ANSI C programming language. There were 77 mutation operators defined in this set, which was designed to follow the C language specification. These operators are classified into variable mutation, operator mutation, constant mutation and statement mutation. Delamaro et al. [60, 62, 58, 59] investigated the application of Mutation Testing at the integration level. They selected 10 mutation operators from Agrawal et al.'s 77 mutation operators to test interfaces of C programs. These mutation operators focus on injecting faults into the signature of public functions. More recently, higher order mutant testing has also been applied to C Programs by Jia and Harman [136].

There are also mutation operators that target specific C program defects or vulnerabilities. Shahriar and Zulkernine [246] proposed 8 mutation operators to generate mutants that represent Format String Bugs (FSBs). Vilela et al. [270] proposed 2 mutation operators representing faults associated with static and dynamic memory allocations, which were used to detect Buffer Overflows (BOFs). This work was subsequently extended by Shahriar and Zulkernine [245] who proposed 12 comprehensive mutation operators to support the testing of all BOF vulnerabilities, targeting vulnerable library functions, program statements and buffer size. Ghosh et al. [104] have applied Mutation Testing to an Adaptive Vulnerability Analysis (AVA) to detect BOFs.

Mutation Testing for Java

Traditional mutation operators are not sufficient for testing Object Oriented (OO) programming languages like Java [146, 168]. This is mainly because the faults represented by the traditional mutation operators are different to those in the OO environment, due to OO's different programming structure. Moreover, there are new faults, introduced by OO-specific features, such as inheritance and polymorphism.

As a result, the design of Java mutation operators was not strongly influenced by previous work. Kim et al. [144] were the first to design mutation operators for the Java programming language. They proposed 20 mutation operators for Java using HAZOP (Hazard and Operability Studies). HAZOP is a safety technique which investigates and records the result of system deviations. In Kim et al.'s work, HAZOP was applied to the Java syntax definition to identify the plausible faults of the Java programming language. Based on these plausible faults, 20 Java mutation operators were designed, falling into six groups: Types/Variables, Names, Classes/interface declarations, Blocks, Expressions and others.

Based on their previous work on Java mutation operators, Kim et al. [143] introduced Class Mutation, which applies mutation to OO (Java) programs targeting faults related to OO-specific features. In Class Mutation, three mutation operators representing Java OO-features were selected from the 20 Java mutation operators. In 2000, Kim et al. [145] added another 10 mutation operators for Class Mutation. Finally, in 2001, the number of the Class mutation operators was extended to 15 and these mutation operators were classified into four types: polymorphic types, method overloading types, information hiding and exception handling types [146]. A similar approach was also adopted by Chevalley and Thevenod-Fosse in their work [48, 49].

Ma et al. [167, 168] pointed out that the design of mutation operators should not start with the selected approach (Kim et al.'s approach [143]). They suggested that the selected mutation operators should be obtained from empirical results of the effectiveness of all mutation operators. Therefore, instead of continuing Kim et al.'s work [145], Ma et al. [167] proposed 24 comprehensive Java mutation operators based on previous studies of OO fault models. These are classified into six groups: Information Hiding group, Inheritance group, Polymorphism group, Overloading group, Java Specific Features group and Common Programming Mistakes group. Ma et al. conducted an experiment to evaluate the usefulness of the proposed class mutation operators [166]. The results suggested that some class mutation model faults can be detected by traditional Mutation Testing. However, the mutants generated by the EOA class mutation (Reference assignment and content assignment replacement) and the EOC class mutation (Reference comparison and content comparison replacement) can not be killed by a traditional mutation adequate test set.

There are also alternative approaches to the definition of the mutation operators for Java. For example, instead of applying mutation operators to the program source, Alexander et al. [25, 9] designed a set of mutation operators to inject faults into Java utility libraries, such as, the Java container library and the iterator library. Based on work on traditional mutation operators, Bradury et al. [33] introduced an extension to the concurrent Java environment.

Mutation Testing for C#

Based on previous proposed Java mutation operators, Derezińska introduced an extension to a set of C# specialised mutation operators [73, 74] and implemented them in a C# mutation tool named CREAM [75]. Empirical results for this set of C# mutation operators using the CREAM were reported by Derezińska and Szustek [74, 76].

Mutation Testing for SQL

Mutation testing has also been applied to SQL code to detect faults in database applications. The first attempt to the design of mutation operators for SQL was done by Chan et al.[46] in 2005. They proposed 7 SQL mutation operators based on the enhanced entity-relationship model. Tuya et al. [265] proposed another set of mutant operators for SQL query statements. This set of mutation operators is organized into four categories, including mutation of SQL clauses, mutation of operators in conditions and expressions, mutation handling NULL values and mutation of identifiers. They also developed a tool named SQLMutation that implements this set of SQL mutation operators and an empirical evaluation concerning results using SQLMutation [264]. A development of this work targeting Java database applications can be found in the work of Zhou and Frankl [295]. Shahriar and Zulkernine [244] have also proposed a set of mutation operators to handle the full set of SQL statements from connection to manipulation of the database. They introduced 9 mutation operators and implemented them in an SQL mutation tool called MUSIC.

Mutation Testing for Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm that aids programmers in separation of crosscutting concerns. Ferrari et al. [97] proposed 26 mutation operators based on a generalisation of faults for general Aspect-Oriented programs. These mutation operators are divided into three groups: pointcut expressions, aspect declarations and advice definitions and implementation. Empirical results from evaluation of this work using real world applications can also be found in their work [97]. A recent work from Delamare et al. introduced an approach to detect equivalent mutants in AOP programs using static analysis of aspects and base code [55].

AspectJ is a widely studied aspect-oriented extension of the Java language, which provides many special constructs such as aspects, advice, join points and pointcuts [13]. Baekken and Alexander [18] summarised previous research work on the fault model associated with AspectJ pointcuts. They proposed a complete AspectJ fault model based on the incorrect pointcut pattern, which was used as a set of mutation operators for AspectJ programs. Based on this work, Anbalagan and Xie [12, 13] proposed a framework to generate mutants for pointcuts and to detect equivalent mutants. To reduce the total number of mutants, a classification and ranking approach based on the strength of the pointcuts was also introduced in their framework.

Other Program Mutation Applications

Besides these programming languages, mutation testing has also been applied to Lustre programs [85, 84], PHP programs [247], Cobol programs [115], Matlab/Simulink [291] and spreadsheets [2]. There is also research work investigating the design of mutation operators for real-time systems [103, 258, 193, 194] and concurrent programs [44, 106, 165, 33, 8].

2.4.2 Specification Mutation

Although mutation testing was originally proposed as a white box testing technique at the implementation level, it has also been applied at the software design level. Mutation Testing at design level is often referred to as 'Specification Mutation', which was first introduced by Gopal and Budd in 1983 [112, 41]. In Specification Mutation, faults are typically seeded into a state machine or logic expressions to generate 'specification mutants'. A specification mutant is said to be killed if its output condition is falsified. Specification Mutation can be used to find faults related to missing functions in the implementation or specification misinterpretation [219].

Mutation Testing for Formal Specifications

The formal specifications can be presented in many forms, for example calculus expressions, Finite State Machines (FSM), Petri Nets and Statecharts. The earlier research work on Specification Mutation considered specifications of simple logical expressions. Gopal and Budd [112, 41] considered specifications in predicate calculus targeting the predicate structure of the program under test. A similar work applied to the refinement calculus specification can be found in the work of Aichernig [7]. Woodward [287, 285] investigated mutation operators for algebraic specifications. In their experiment, they applied an optimization approach to compile a specification mutant into executable code and evaluated the approach to provide empirical results [286].

More recently, many formal techniques have been proposed to specify the dynamic aspects of a software system, for example, Finite State Machines (FSM), Petri Nets and State charts. Fabbri et al. [95] applied Specification Mutation to validate specifications presented as FSMs. They proposed 9 mutation operators, representing faults related to the states, events and outputs of an FSM. This set of mutation operators was later implemented as an extension of the C mutation tool Proteum [92]. An empirical evaluation of these mutation operators was reported by them [92]. Hierons and Merayo [125, 126] investigated the application of mutation testing to Probabilistic Finite State Machines (PFSMs). They defined 7 mutation operators and provided an approach to avoid equivalent mutants. Other work on EFSM mutation can also be found in the work of Batth et al. [21], Bombieri et al. [29] and Belli et al. [23].

Statecharts are widely used for the formal specification of complex reactive systems. Statecharts can be considered as an extension of FSMs, so the first set of mutation operators for Statecharts was also proposed by Fabbri et al. [94], based on their previous work on FSM mutation operators. Using Fabbri et al.'s Statecharts mutation operators, Yoon et al. [289] introduced a new test criterion, the State-based Mutation Test Criterion (SMTC). In the work of Trakhtenbrot [262], the author proposed new mutations to assess the quality of tests for statecharts at the implementation level as well as the model level. Other work on Statechart mutation can be found in the work of Fraser et al. [101].

Besides FSMs and Statecharts, Specification Mutation has been also applied to a variety of specification languages. For example, Souza et al. [253, 254] investigated the application of mutation testing to the Estelle Specification language. Fabbri et al. [93] proposed mutation operators for Petri Nets. Srivatanakul et al. [256] performed an empirical study using Specification Mutation to CSP Specifications. Olsson and Runeson [220] and Sugeta et al. [257] proposed mutation operators for SDL. Definitions of mutation operators for formal specification language can be found in the work of Black et al. [26] and the work of Okun [219].

Mutation Testing for Web Services

Lee and Offutt [159] were the first to apply Mutation Testing to Web Services. In 2001, they introduced an Interaction Specification Model to formalize the interactions between web components [159]. Based on this specification model, a set of generic mutation operators was proposed to mutate the XML data model. This work was later extended by Xu et al. [215, 288] targeting the mutation of XML data and they renamed it XML perturbation. Instead of mutating XML data directly, they perturbed XML schemas to create invalid XML data using 7 XML schema mutation operators. A constraint-based test case generation approach was also proposed and the results of empirical studies were reported [288]. Another set of XML schema mutation operators was proposed by Li and Miller [160].

There is also Web Service mutation work targeting specific XML-based language features, for example, the OWL-S specification language [158, 276] and WS-BPEL specification language [91]. Unlike the traditional XML specification language, OWL-S introduces semantics to workflow specification using an ontology specification language. In the work of Lee et al. [158], the authors propose mutation operators for detection of semantic errors caused by the misuse of the ontology classes.

Mutation Testing for Networks

Protocol robustness is an important aspect of any network system. Sidhu and Leung [248] investigated fault coverage of network protocols. Based on this work, Probert and Guo proposed a set of mutation operators to test network protocols [230]. Vigna et al. [269] applied Mutation Testing to network-based intrusion detection signatures, which are used to identify malicious traffic. Jing et al. [140] built a NFSM model for protocol messages and applied mutation testing to this model using the TTCN-3 specification language. Other work on the application of mutation testing to State based protocols can be found in the work of Zhang et al. [294].

Mutation Testing for Security Policy

Mutation Testing has also been applied to security policies [157, 172, 187, 186, 229]. Much of this research work sought to designed mutation operators that inject common flaws into different types of security policies. For example, Xie et al. [172] applied mutation analysis to test XACML, an Oasis standard XML syntax for defining security policies. A similar approach has also been applied by Mouelhi et al. [187]. Le Traon et al. [157] introduced 8 mutation operators for the Organization Based Access Control OrBAC policy. Mouelhi et al. [186] proposed a generic meta-model for security policy formalisms. Based on this formalism, a set of mutation operators was introduced to apply to all rule-based formalisms. Hwang et al. proposed an approach that applies mutation testing to test firewall policies [131].

2.4.3 Other Testing Applications

In addition to assessing the quality of test sets, Mutation Testing has also been used to support other testing activities, for example test data generation and regression testing, including test data prioritization and test data minimization. In this section, we summarise the main work on mutation as a support to these testing activities.

Mutation-based Test Data Generation

The main idea of mutation based test data generation is to automatically generate test data that can effectively kill mutants. There has been much work on different techniques and tools for generating mutants, with over 250 publications on mutation testing. However, the literature contains only 10 publications (about 4% of the total) that address the problem of automatically generating test data to kill mutants [139]. While mutation generation remains important, it is also clearly desirable to be able to use mutation testing to generate test cases as well as to asses them.

Previous work on the generation of test data to kill mutants has used traditional structural-oriented test data generation techniques, for example, traditional symbolic execution [70, 164, 195, 201, 216], Dynamic Symbolic Execution (DSE) [222, 225, 293] and Search Based Software Testing (SBST) [17, 102]. However, all of the existing techniques are designed to achieve only weak mutation adequacy and only for first order mutants. There is neither existing work on killing higher order mutants, nor any work on generating strong mutation adequate test data.

In order to (strongly) kill a first order mutant the killing conditions are well studied in the literature: A test input needs to satisfy following three conditions: Reachability, Infection and Propagation (RIP), each of which subsumes the preceding condition(s):

1. **Reachability**: The location of the mutant in the program must be executed by the test case. We say the mutant is 'reached'. Reaching all mutants of a program can be achieved by any branch adequate test set, so reachability is an instance of branch coverage, which is widely studied in literature [10, 111, 123, 241].

2. **Infection**: Immediately after mutant execution, the original program state and that of the mutant must differ. We say, the mutant 'infects' the state. A test case that achieves infection for a mutant m is also said to 'weakly kill' the m [70, 139, 185].

3. **Propagation**: The infected state must propagate to some point in the program at which it can be observed, such as an output statement. A test case that achieves propagation for a mutant m is also said to 'strongly kill' the m [70, 139, 185].

Constraint Based Testing (CBT) was the first test data generation technique used for mutation testing. It was proposed by DeMillo and Offutt, based on the idea of control flow analysis and symbolic execution [70, 195]. Constraint based testing seeks to generate test data to kill mutants weakly by reaching and infecting mutants, thereby achieving the 'R' and 'I' of the 'RIP' framework. Offutt and DeMillo represent reachability as a set of path conditions, constructed using control flow analysis and symbolic execution and augment these path constraints with constraints that denote infection.

The initial approach to CBT suffered from several problems inherited from the state-of-the-art in symbolic evaluation available at the time and also from the static domain reduction technique used. It was unable to handle arrays, loops and nested expressions well. To overcome these restrictions, Offutt et al. proposed a dynamic domain reduction technique [201, 216]. The dynamic domain reduction technique uses a more sophisticated back-tracker to dynamically split domains.

Dynamic Symbolic Execution (DSE) [111, 241] is a more recent innovation that overcomes many of the limitations of traditional symbolic execution. Using DSE, non-linear path constraints are simplified by the instantiation of concrete runtime values, harvested from program execution. DSE has been used in several coverage based testing tools, such as DART [111], CUTE [241] and Pex [261].

DSE also provides a natural way to generate weakly adequate mutation-based test inputs. A simple testability transformation [119] can be used to augment the program with conditional statements, the predicates of which capture the infection constraints. By construction, covering the branches of the transformed program entails satisfying these infection constraints, thereby tranforming branch coverage into weak mutation coverage. This approach was first suggested by Liu et al. [164], and was implemented by Zhang et al. [293] and Papadakis et al. [224].

Search Based Software Testing (SBST) [10, 121] has also been applied to the generation of weakly adequate mutation-based test data. Bottaci was the first to suggest using SBSE to kill mutants [30]. However, Search Based Mutation Test Generation remained unimplemented and unevaluated until the subsequent for work of Ayari et al. [17] and Fraser and Zeller [102], both of whom target Java.

The SHOM approach introduced in Chapter 6 combines DSE and SBST. It uses DSE to achieve weak mutation adequacy and extends this with a constraint-aware search based approach that maintains weak adequacy, while seeking to propagate tests to achieve strong mutation adequacy. SHOM thus extends previous work by generating test data for strong mutation adequacy and by generating test data for higher order mutants.

Regression testing

Test case prioritisation techniques are one way to assist regression testing. Mutation Testing has been applied as a test case prioritisation technique by Do and Gregg [79, 80]. Do and Gregg measured how quickly a test suite detects the mutant in the testing process. Testing sequences are rescheduled based on the rate of mutant killing. Empirical studies suggested that this automated test case prioritisation can effectively improve the rate of fault detection of test suites [80].

Mutation testing has also been used to assist the test case minimisation process. Test case minimisation techniques aim to reduce the size of a test set without losing much test effectiveness. Offutt et al. [210] proposed an approach named Ping-Pong. The main idea is to generate mutants targeting a test criterion. A subset of test data with the highest mutation score is then selected. Empirical studies show that Ping-Pong can reduce a mutation adequacy test set by a mean of 33% without loss of test effectiveness.

In addition to the previous mentioned applications, mutation analysis has also been applied to other application domains. For example, Serrestou et al. proposed an approach to evaluate and improve the functional validation quality of RTL in a hardware environment [243, 242]. Mutation analysis has also been used to assist the evaluation of software clone detection tools [234, 235].

Mutation Testing for Running Environment

During the process of implementing specifications, bugs might be introduced by programmers due to insufficient knowledge of the final target environment. These bugs are called “environment bugs” and they can be hard to detect. Examples are the bugs caused by memory limitations, numeric limitations, value initialisation, constant value interpretation, exception handling and system errors [255]. Mutation testing was first applied to the detection of such bugs by Spafford [255] in 1990. In his work, environment mutants were generated to detect integer arithmetic environmental bugs.

The idea of environment bugs was extended in 1990s by Du and Mathur, as many empirical studies suggested that “the environment plays a significant role in triggering security flaws that lead to security violations”[82]. As a result, mutation testing was also applied to the validation of security vulnerabilities. Du and Mathur [82] defined an EAI fault mode for software vulnerability, and this model was applied to generate environmental mutants. Empirical results from the evaluation of their experiments are reported in [83].

2.5 Empirical Evaluation

Many researchers have conducted experiments to evaluate the effectiveness of Mutation Testing [99, 100, 178, 279, 209, 14, 69, 54]. These experiments can be divided into two types: comparing mutation criteria with data flow criteria such as “all-use” and comparing mutants with real faults. Table 2.4 summarises the evaluation type and the subject programs used in each of these experiments.

Mathur and Wong have conducted experiments to compare the “all-use” criterion with mutation criteria [178, 279, 282]. In their experiment, Mathur and Wong manually generated 30 sets of test cases satisfying each criterion for each subject program. Empirical results suggested that mutation adequate test sets more easily satisfy the "all-use" criteria than all use test sets satisfy mutation criteria. This result indicates mutation criteria "probsubsumes" [2] the "all-use" criteria in general.

Table 2.4: Empirical Evaluation of Mutation Testing

Research	Evaluation Type	Subject Programs
DeMillo and Mathur [69]	real faults vs mutants	Tex
Mathur and Wong [178, 279]	all-use vs mutation criteria	Find, Strmat1, Strmat2 and Textfmt
Offutt et al. [209]	all-use vs mutation criteria	Bub, Cal, Euclid, Find, Insert, Mid, Pat, Quad, Trityp and Warshall
Daran and Thévenod-Fosse [54]	real faults vs mutants	Nuclear Reactor Safety Shutdown System
Frankl et al. [99, 100]	all-use vs mutation criteria	Determinant, Find1, Find2, Matinv1, Matinv2, Strmatch1, Strmatch2, Textformat.r and Transpose
Andrews et al. [14]	hand seeded faults vs mutants	Space, Printtokens, Printtokens2, Replace, Schedule, Schedule2, Tcas and Totinfo
Do and Rothermel [79, 80]	hand seeded faults vs mutants	Ant, Xml-security, Jmeter, Jtopas, galileo and nanoxml
Li et ail. [161]	all-users, edge-pair and prime path coverage vs mutation criteria	Twenty nine anonymous Java classes

Offutt et al. conduced a similar experiment using ten different programs [209]. The 'cross scoring' result also provides evidence for Mathur and Wong's probsubsumes relationship [178, 279]. In addition to comparing the two criteria with each other,

[2]If a test criterion C_1 probsumes a test criterion C_2, a test set which is adequate to C_1 is likely to be adequate to C_2 [209]

Offutt et al. also compared the two criteria in terms of the fault detection rate. This result showed that 16% more faults can be detected using mutation adequate test sets than “all-use” test sets, indicating that mutation criteria is “probbetter” [3] than the “all-use” data flow. This conclusion also agreed with the results of the experiment of Frankl et al. [99, 100]. The most recent work on the comparison between mutation criteria with other criteria was conducted by Li et al. [161]. They compared the mutation testing criterion with three other commonly used criteria: prime path coverage, edge-pair coverage and all-uses. The results suggest that the mutation testing criterion not only finds more faults than other criteria, but is also the most efficient criterion.

In addition to comparing mutation analysis with other testing criteria, there have also been empirical studies comparing real faults and mutants. In the work of Daran and Thévenod-Fosse [54], the authors conducted an experiment comparing real software errors with 1st order mutants. The experiment used a safety-critical program from the civil nuclear field as the subject program with 12 real faults and 24 generated mutants. Empirical results suggested that 85% of the errors caused by mutants were also produced by real faults, thereby providing evidence for the Mutation Coupling Effect Hypothesis. This result also agreed with DeMillo and Mathur’s experiment [69]. DeMillo and Mathur carried out an extensive study of the errors in TeX reported by Knuth[69] and they demonstrated how simple mutants could detect real complex errors from TeX.

Andrews et al. [14] conducted an experiment comparing manually instrumented faults generated by experienced developers with mutants automatically generated by 4 carefully selected mutation operators. In the experiment, the Siemens suite

[3]If a test criterion C_1 probbetter than a test criterion C_2, then a randomly selected test set which satisfies C_1 is more likely to detect a fault than a randomly selected test set which satisfies C_2 [209]

(Printtokens, Printtokens2, Replace, Schedule, Schedule2, Tcas and Totinfo) and the Space program were used as subjects. Empirical results suggested that, after filtering out equivalent mutants, the remaining non-equivalent mutants generated from the selected mutation operators were a good indication of the fault detection ability of a test suite. The results also suggested that the human generated faults are different from the mutants; both human and auto-generated faults are needed for the detection of real faults.

Do and Rothermel [79, 80] studied the effect of both hand seeded faults and machine generated mutants on fault detection ability and the test prioritisation order. In the test data prioritisation study, Do and Rothermel considered several prioritisation techniques to improve the fault detection rate. Their analysis showed that for non-control test case prioritisation, the use of mutation can improve fault detection rates. However the results are affected by the number of mutation faults applied. In the fault detection ability studies, Do and Rothermel followed Andrews et al.'s experimental procedure [14]. Results from 4 out of the 6 subject programs revealed a similar data spread to the work of Andrews et al. The effect of test set minimisation using mutation can be found in the work of Wong et al. [280].

Despite evaluating mutation testing against other testing approaches, there are also experiments that use mutation analysis to evaluate different testing approaches. For example, Andrews et al. [15] conducted an experiment to compare test data generation using control flow and data flow. Thevenod et al. [260] applied mutation analysis to compare random and deterministic input generation techniques. Bradbury et al. [34] used mutation analysis to evaluate traditional testing and model checking approaches on concurrent programs.

Summary

This chapter has provided a detailed survey of mutation testing. The paper covers theories, optimisation techniques, equivalent mutant detection, applications and empirical studies. The next chapter will present a comprehensive analysis of the development trends of mutation testing.

Chapter 3

Analysis of the Development of Mutation Testing

The literature on mutation testing has contributed a set of approaches, tools, developments and empirical results as surveyed in Chapter 2. This chapter presents the results of several development trend analyses. These analyses provide evidence that mutation testing techniques and tools are reaching a state of maturity and applicability, while the topic of mutation testing itself is the subject of increasing interest.

In order to provide a complete trend analysis covering all the publications related to mutation testing since the 1970s, we constructed a mutation testing publication repository, which includes more than 390 papers from 1977 to 2009 [135] (This analysis was carried out in 2010). We took four steps to build this repository. First we searched the online repositories of the main technical publishers, including IEEE explore, ACM Portal, Springer Online Library, Wiley Inter Science and Elsevier Online Library, collecting papers which have either "mutation testing", "mutation analysis", "mutants + testing", "mutation operator + testing", "fault injection" and “fault based testing” keywords in their title or abstract. Then we went through the references for each paper in our repository, to find missing papers using the same keyword rules. In this way, we performed a ‘transitive closure’ on the literature. mutation testing work which was not concerned with software, for example, hardware and also filtered out papers not written in English. We have made the repository publicly available at http://crestweb.cs.ucl.ac.uk/resources/mutation_testing_repository/ [135].

The rest of the chapter is organised as follows. Section 3.1 presents the publication trend for mutation testing. Section 3.2 presents some development trends including mutation techniques, applications and subject programs used in empirical study. Section 3.3 describes the development work on mutation tools. Section 3.4 discusses the unresolved problems, barriers and the areas of success in mutation testing.

3.1 Publication Trends

To understand the general trend for the mutation testing research area, we analysed the number of publications by year from 1977 to 2009. Consider again the results in Figure 3.1; there are five apparent outliers in years 1994, 2001, 2006, 2007 and 2009. The reason for the last four years, is that there were four mutation testing workshops held in 2000 (with proceedings published in 2001), 2006, 2007 and 2009. However, there is no direct evidence to explain the spike in year 2004; this just appears to be an anomalous productive year for Mutation Testing. The reader will also notice that 1986 is unique as no publications were found. An interesting explanation was provided by Offutt [198]: “1986 was when we were maximally devoted to programming Mothra. ”

We performed a regression analysis on these data and found there is a strong positive correlation between year and the number of publications ($r = 0.7858$). In order to

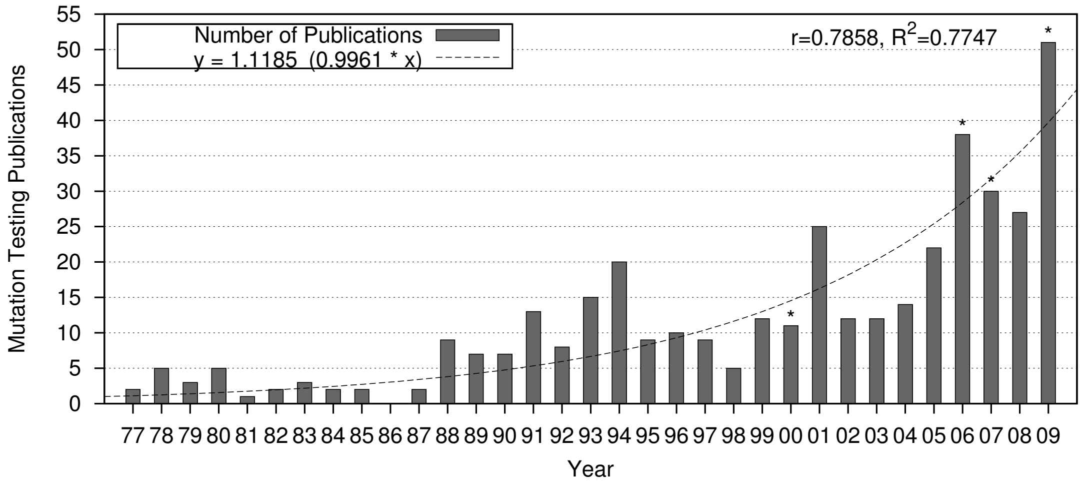

Figure 3.1: Mutation Testing Publications from 1978-2009 (* indicates years in which a mutation workshop was held.)

predict the trend of publications in the future, we have tried to find a trend line for this data using several common regression models: Linear, Logarithmic, Polynomial, Power, Exponential and Moving average. The dashed line in Figure 3.1 is the best fit line we found. It uses a quadratic model, which achieves the highest coefficient of determination ($R^2 = 0.7747$). To put the Mutation Testing growth trend into a wider context, we also collected and plotted the publication data from DBLP for the subject of computer science as a whole [263]. According to DBLP, the general growth in computer science is also exponential. From this analysis it is clear that mutation testing remains at least as healthy as computer science itself.

In order to take a closer look at the growing trend of the research work on Mutation Testing, we have classified this work into theoretical work and practical work. The theoretical category includes the publications concerning the hypotheses supporting mutation testing, optimisation techniques, techniques for reducing computational cost and techniques for the detection of equivalent mutants and surveys. The practical category includes publications on applications of mutation testing, development work on mutation testing tools and related empirical studies.

The goal of this separation of papers into theoretical and practical work is to allow us to analyse the temporal relationship between the development of theoretical and practical research effort by the community. Figure 3.2 shows the overall cumulative result. It is clear that both theoretical and practical work is increasing. In 2006 for the first time, the total number of practical publications surpasses the number of theoretical publications. To take a closer look at this relationship, Figure 3.3 shows the number of publications per year. From 1977 to 2000, there were fewer practical publications than theoretical. From 2000 to 2009, most of the research work appears to shift to the application area. This provides some evidence to suggest that the field is starting to move from foundational theory to practical application, possibly a sign of increasing maturity.

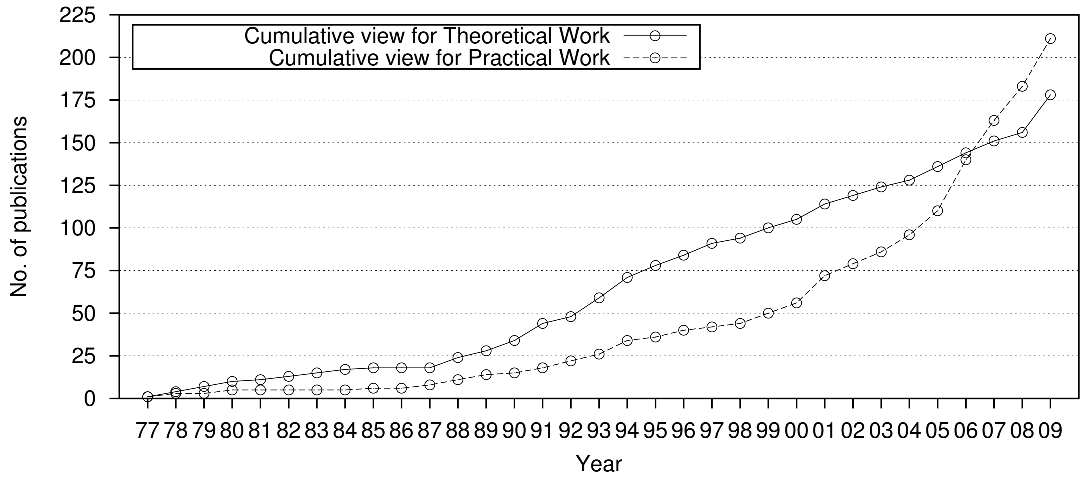

Figure 3.2: Theoretical Publications vs. Practical Publications (Cumulative view)

3.2 Development Trends

Mutation Techniques

Section 2.2 introduced a number of cost reduction techniques for mutation testing. Figure 3.4 provides an overview of the chronological development of this techniques.

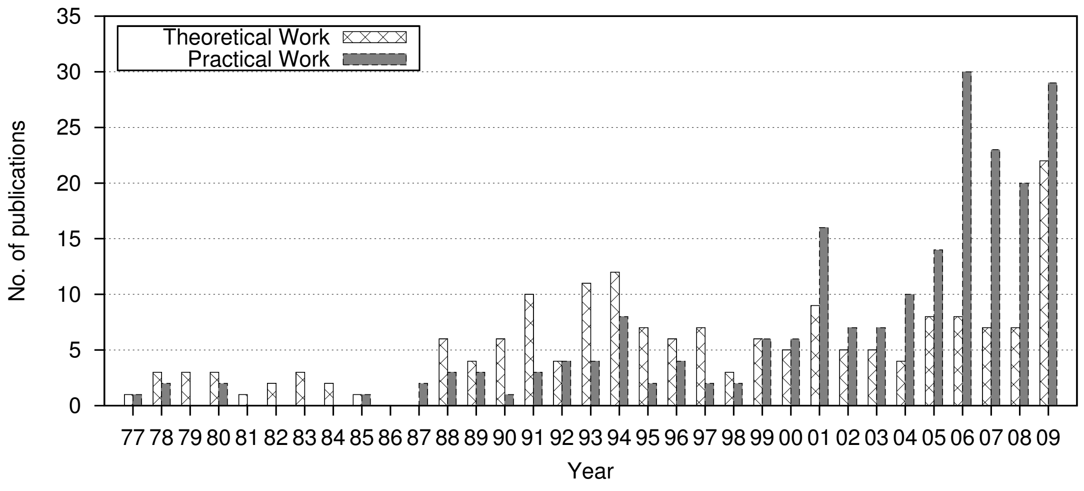

Figure 3.3: Theoretical Publications vs. Practical Publications

To take a closer look at the cost reduction research work, we counted the number of publications for each technique (see Figure 3.5). From this figure, it is clear that Selective Mutation and Weak Mutation are the most widely studied cost reduction techniques. Each of the other techniques is studied in no more than five papers, to date.

Applications

Section 2.2 introduced different applications of mutation testing. Figure 3.6 shows the chronological development of research work on Program Mutation and Specification Mutation. Figure 3.7 shows the percentage of the publications addressing each language to which mutation testing has been applied. As Figure 3.6 shows, there has been more work on Program Mutation than Specification Mutation. Notably more than 50% of the work has been applied to Java, Fortran and C. Fortran features highly because a lot of the earlier work on mutation testing was carried out on Fortran programs. In the following section, the applications of Program Mutation and Specification Mutation are summarised by the programming language targeted.

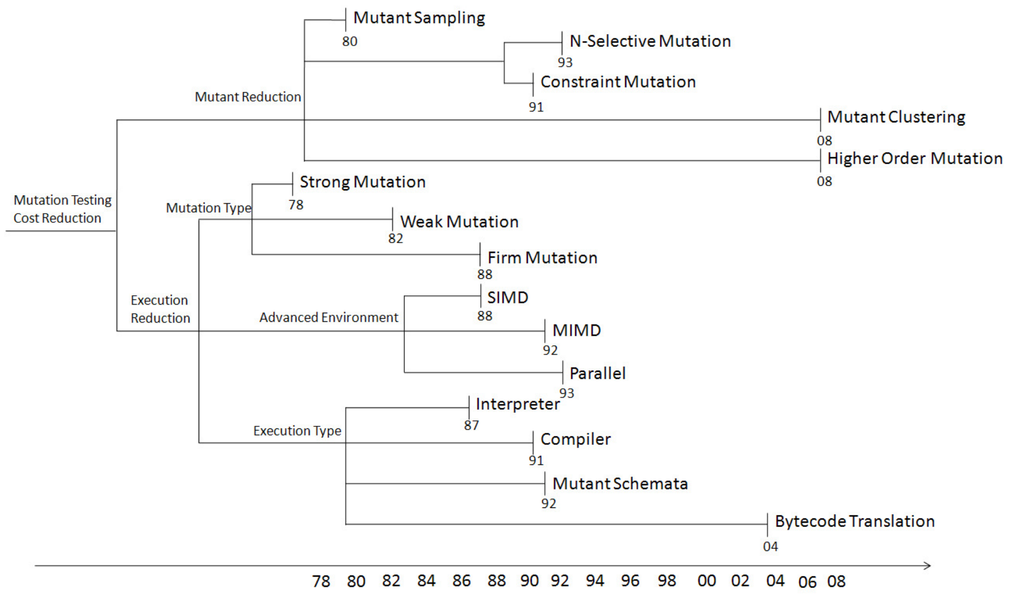

Figure 3.4: Overview of the Chronological Development of Mutant Reduction Techniques

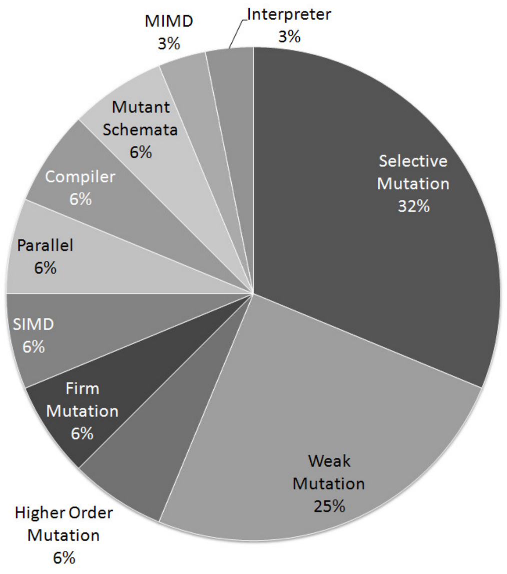

Figure 3.5: Percentage of publications using each Mutant Reduction Technique

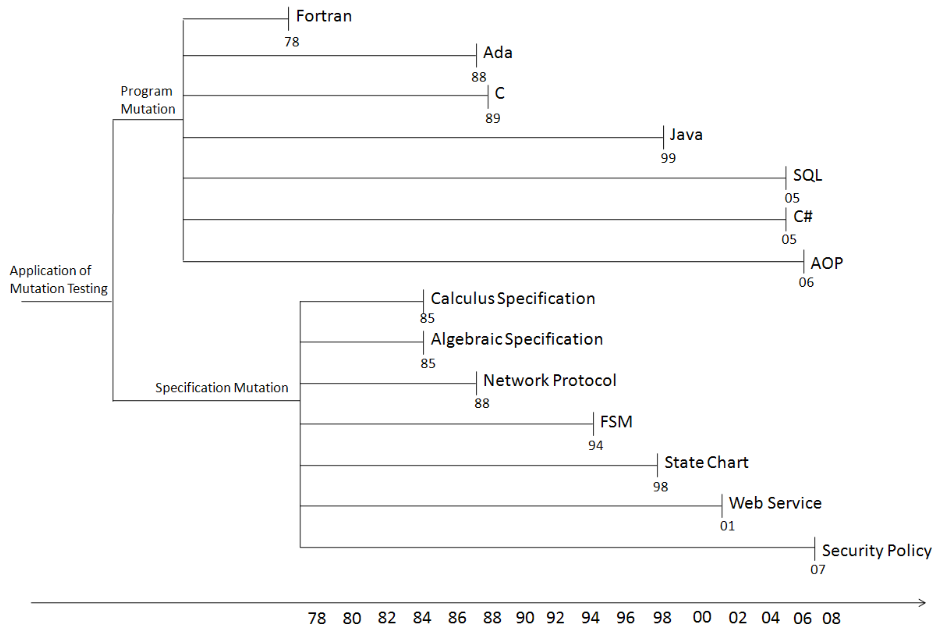

Figure 3.6: Publications of the Applications of Mutation Testing

Subject Programs

We have collected all the subject programs for each empirical experiment work from our repository, as shown in Table A.1 (Table A.1 is located in the Appendix A). Table A.1 shows the name, size, description, the year when the subject program was first applied and the overall number of research papers that report results for this subject program. The table entry for some sizes and descriptions of the subject programs are shown as “not reported”. This occurs where the information is unavailable in the literature. Table A.1 is sorted by the number of papers that use the subject program, so the first ten programs are the most studied subject programs in the literature on mutation testing. These wildly studied programs are all laboratory programs under 50 LoC but we also noticed that the 11th program is SPACE, a non-trivial real program.

To provide an overview of the trend of empirical studies on mutation testing to attack more challenging programs, we calculated the size of the largest subject program for

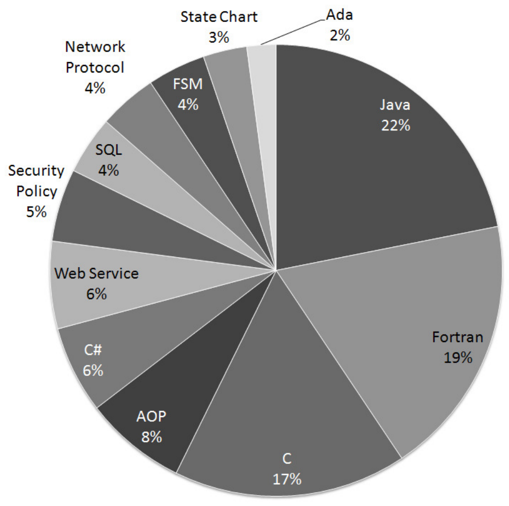

Figure 3.7: Percentage of publications addressing each language to which Mutation Testing has been applied

each year. For each year on the horizontal axis, the data point in Figure 3.8 shows the size of the largest program considered in a mutation study up to that point in time. Clearly the definition of “program size” can be problematic, so the figure is merely intended to be used as a rough indicator. There is evidence to indicate that the size of the subject programs that can be handled by mutation testing is increasing. However, caution is required. We found that although some empirical experiments were reported to handle large programs, some studies applied only a few mutation operators. We also counted the number of newly introduced subject programs for each year. The results are shown in Figure 3.9. The dashed line in the figure is the cumulative view of the results. The number of newly used subject programs is gradually increasing, which suggests a growth in practical work.

In the empirical studies, it may be more indicative to use a real world program rather than laboratory program. To understand the relationship between the use of laboratory programs and real world programs in mutation experiments, we have counted each type by year. The results are shown in Figure 3.10. In this study, we consider a real world program to be either an open source or an industry program.

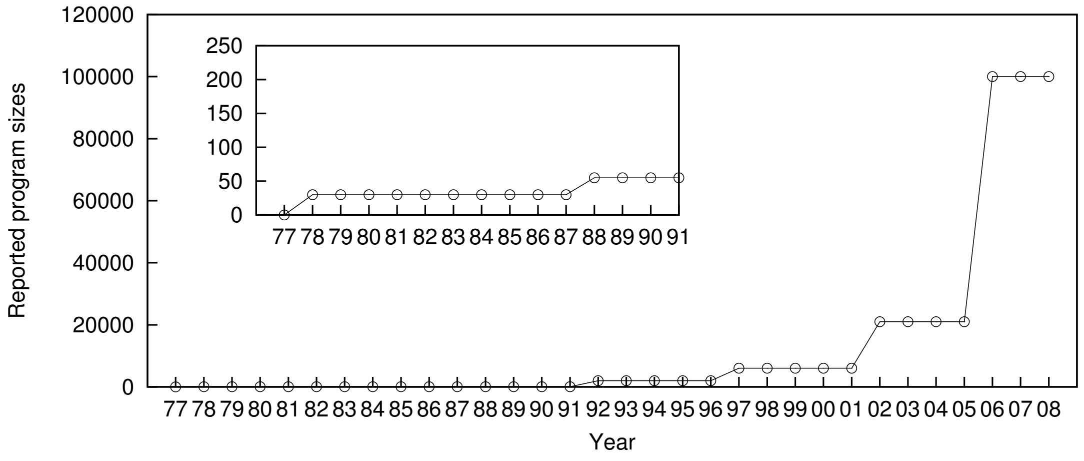

Figure 3.8: The largest program applied for each year

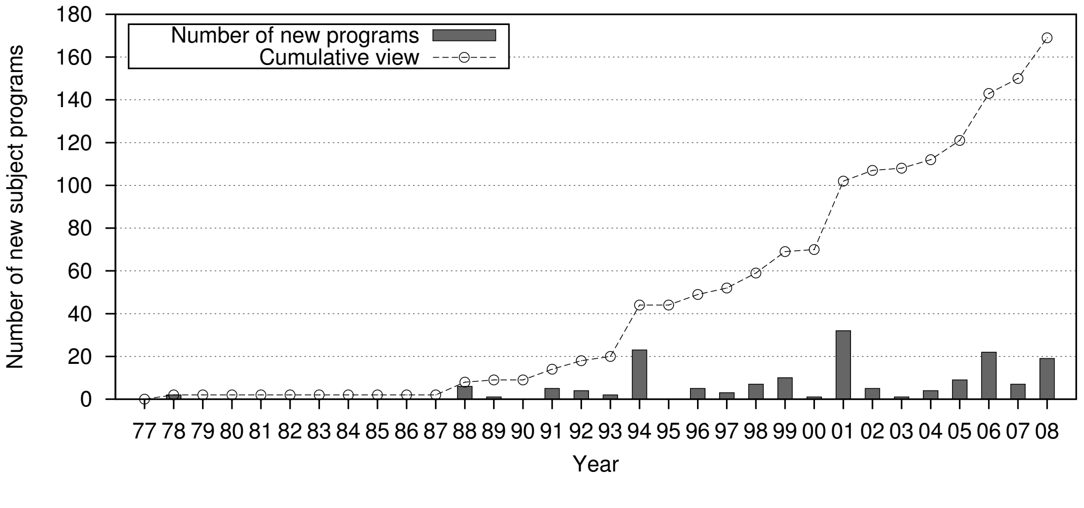

Figure 3.9: New programs applied for each year.

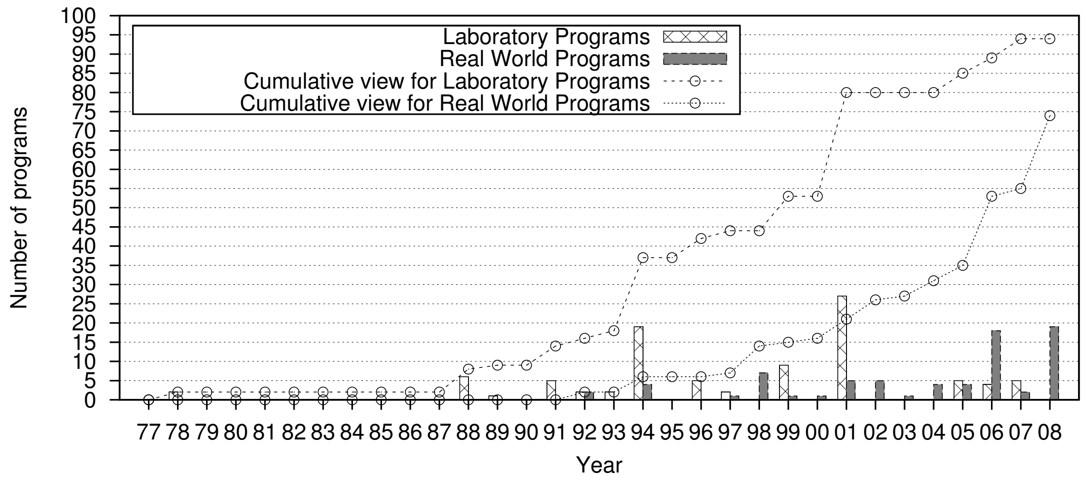

Figure 3.10: Laboratory programs vs. Real Programs

In Figure 3.10, the cumulative view shows that the number of real world programs started increasing in 1992, while the number of laboratory programs had already started increasing by 1988. Figure 3.10 also shows the number of laboratory and real programs introduced into studies each year as bars. This clearly indicates that, while there are correctly more laboratory programs overall, since 2002, far more new real programs than laboratory programs have been introduced. This finding provides some evidence to support the claim that the development of mutation testing is maturing.

In our study, we found that for each research area of mutation testing there is a different set of subject programs used as benchmarks. In Table 3.1 we have summarised these benchmark programs. We chose five active research areas based on our studies: Coupling effect, Selective Mutation, Weak, Strong and Firm Mutation, Equivalent Mutant Detection and experiments supporting testing, including the use of mutation analysis to select, minimise, prioritise and generate test data.

Table 3.1: Subject Programs by Application

Application	Subject Programs	Reference
Coupling Effect	Triangle, Find, MID	[196, 197]
Selective Mutation	Triangle, Find, Bubble, MID, Calendar, Euclid, Quad, Insert, Warshall, Pat, Totinfo, Schedule1, Schedule2, TCAS, Printtok1, Printtok2, Space, Replace, Banker, Sort, Areasg, Minv, Rpcalc, Seqstr, Streql, Tretrvi, Append, Archive, Change, Ckglob, Cmp, Command, Compare, Compress, Dodash, Edit, Entab, Expand, Getcmd, Getdef, Getfn, Getfns, Getlist, Getnum, Getone, Gtext, Makepat, Omatch, Optpat, Spread, Subst, Translit, Unrotate	[212, 203, 188, 20, 189, 191]
Weak, Strong, Firm Mutation	Triangle, Find, Bubble, MID, Calendar, Euclid, Quad, Insert, Warshall, Pat, Gcd, Sort, Max_index	[287, 205, 204]
Equivalent Mutant	Triangle, Find, Bubble, MID, Calendar, Euclid, Quad, Insert, Warshall, Pat, Bsearch, Max, Banker, Deadlock, Count, Dead	[200, 207, 208]
Testing (test case generation, prioritization, selection and reduction)	Triangle, Find, Bubble, MID, Calendar, Euclid, Quad, Insert, Warshall, Pat, Space, Bsearch, Totinfo, Schedule1, Schedule2, TCAS, Printtok1, Printtok2, Replace, Gcd, Binom, Ant, Stats Twenty-four, Conversions, Operators, XmlSecurity, Jmeter, JTopas, ATM, BOOK, VirtualMeeting, MinMax, NextDate, Finance	[70, 71, 201, 210, 281, 217, 79, 80, 164, 17, 128]

3.3 Tools for Mutation Testing

The development of Mutation Testing tools is an important enabler for the transformation of Mutation Testing from the laboratory into a practical and widely used testing technique. Without a fully automated mutation tool, Mutation Testing is unlikely to be accepted by industry. In this section, we summarise development work on Mutation Testing tools.

Since the idea of Mutation Testing was first proposed in the 1970s, many mutation tools have been built to support automated mutation analysis. In our study, we have collected information concerning 36 implemented mutation tools, including the academic tools reported in our repository as well as the tools from the open source and the industrial domains. Table 3.3 summarises the application, publication time and any notable characteristics for each tool. The detailed description of the tools can be found in the references cited in the final column of the table.

Figure 3.11 shows the growth in the number of tools introduced. In Figure 3.11, the development work can be classified into three stages. The first stage was from 1977 to 1981. In this early stage, in which the idea of Mutation Testing was first proposed, four prototype experimental mutation tools were built and used to support the establishment of the fundamental theory of mutation analysis, such as the Competent Programmer Hypothesis [4] and the Coupling Effect Hypothesis [68]. The second stage was from 1982 to 1999. There were four tools built in this period, three academic tools, Mothra for Fortran [65, 66], Proteum, TUMS for C [56, 57, 267] and one industry tool called Insure++. Engineering effort had been put into Mothra and Proteum so that they were able to handle small real programs not just laboratory programs. As a result, these two academic tools were widely used. Most of the advanced mutation techniques were experimented on using these two tools, for example, Weak Mutation [205, 204], Selective Mutation [203, 212], Mutant

Table 3.2: Summary of Published Mutation Testing Tools

Name	Application	Year	Character	Available	Reference
PIMS	Fortran	1977	General	No	[43, 39, 163]
EXPER	Fortran	1979	General	No	[4, 37, 42]
CMS.1	Cobol	1980	General	No	[3, 115]
FMS.3	Fortran	1981	General	No	[259]
Mothra	Fortran	1987	General	Yes	[65, 66]
Proteum 1.4	C	1993	Interface Mutation, Finite State Machines	No	[56, 57]
TUMS	C	1995	Mutant Schemata Generation	No	[266, 268, 267]
Insure++	C/C++	1998	Source Code Instrumentation (Commercial)	Commercially	[226]
Proteum/IM 2.0	C	2001	Interface Mutation, Finite State Machines	Yes	[63]
Jester	Java	2001	General (Open Source)	Yes	[182]
Pester	Python	2001	General (Open Source)	Yes	[182]
TDS	CORBA IDL	2001	Interface Mutation	No	[107]
Nester	C#	2002	General (Open Source)	Yes	[251]
JavaMut	Java	2002	General	Yes	[49]
MuJava	Java	2004	Mutant Schemata, Reflection Technique	Yes	[206, 168, 169]
Plextest	C/C++	2005	General (Commercial)	Commercially	[132]
SQLMutation	SQL	2006	General	Yes	[264]
Certitude	C/C++	2006	General (Commercial)	Commercially	[45]
SESAME	C, Lustre, Pascal	2006	Assembler Injection	No	[53]
ExMAn	C, Java	2006	TXL	Yes	[32]
MUGAMMA	Java	2006	Remote Monitoring	Yes	[147]
MuClipse	Java	2007	Weak Mutation, Mutant Schemata, Eclipse plug-in	Yes	[250]
CSAW	C	2007	Variable type optimization	Yes	[90, 89]
Heckle	Ruby	2007	General (Open Source)	Yes	[236]
Jumble	Java	2007	General (Open Source)	Yes	[252]
Testooj	Java	2007	General	Yes	[228]
ESPT	C/C++	2008	Tabular	Yes	[96]
MUFORMAT	C	2008	Format String Bugs	No	[246]
CREAM	C#	2008	General	No	[76]
MUSIC	SQL(JSP)	2008	Weak Mutation, SQL Vulnerabilities	No	[244]
MILU	C	2008	Higher Order Mutation, Search-based technique, Test harness embedding	Yes	[137]
Javalanche	Java	2009	Invariant and Impact analysis	Yes	[238, 113]
GAmera	WS-BPEL	2009	Genetic algorithm	Yes	[81]
MutateMe	PHP	2009	General (Open Source)	Yes	[35]
AjMutator	AspectJ	2009	General	Yes	[55]
JDAMA	SQL(JDBC)	2009	Byte code translation	Yes	[295]

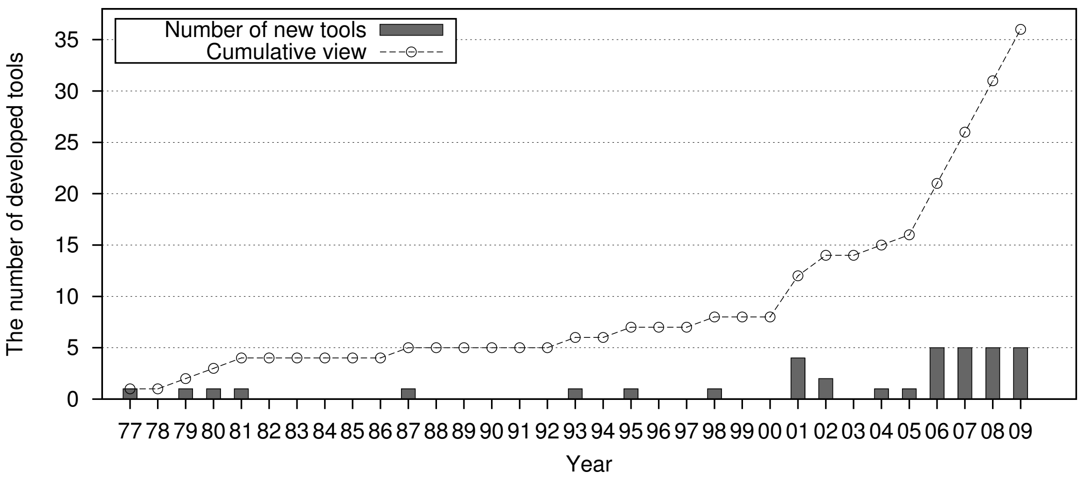

Figure 3.11: The number of tools introduced for each year

Sampling [279, 177] and Interface Mutation [59, 58]. The third stage of mutation testing development appears to have started from the turn of the new millennium, when the first mutation workshop was held. There have been 28 tools implemented since this time. In Figure 3.11, the dashed line shows a cumulative view of this development work. We can see that the tool development trend is rapidly increasing since year 2000, indicating that research work on Mutation Testing remains active and increasingly practical.

In order to explore the impact of mutation testing within the open source and industrial domains, we have classified tools into three classes: academic, open sources and industrial. Table 3.3 shows the number of each class over two periods; one is before the year 2000, the other is from the year 2000 to the present. As can be seen, there are more open source and industrial tools implemented recently, indicating that mutation testing has gradually become a practical testing technique, embraced by both the open source and industrial communities.

Table 3.3: Classification of Mutation Testing Tools

Stage	Overall Tools	Academic Tools	Open Source Tools	Commercial Tools
1975-1999	8	7	0	1
2000-present	28	19	7	2

3.3.1 Academic Tools

The four tools: PIMS, EXPER, CMS.1 and FMS.3 were prototype tools in the very early stages of the development of mutation testing. Although they could only handle small toy programs, they all implemented the basic structure of mutation analysis. Unfortunately non of these tools remain available. Mothra was the most widely studied mutation testing tool. It was initially designed as an extension of PIMS with more 'friendly' functions. Later on it was redesigned as the integration of a set of tools. Mothra provides a formal set of Fortran 77 mutation operators and it also provides several advanced optimisation techniques, such as Mutant Schema Generation [268], constraint-based test data generation [70], Selective Mutation [203] and Weak Mutation [204].

Proteum was the first C mutation tool. It applied the compiler based technique, and implemented Agrawal et al.'s 77 C mutation operators [6]. Beside traditional mutation analysis functions, it was the first tool provided the Interface Mutation function [59]. There were many extended versions of the Proteum tool which supported Specification Mutation technique and application domains, such as Proteum/FSM [92]. CSAW is a lightweight Mutation Testing tool for C programs, developed by Ellims et al. [90]. CSAW adopts an approach to reduce the number of non-killed mutants which is generated by mutation of variable types. MUTFOR-

MAT is a small application designed to find Format String Bugs in C programs by injecting faults into the string format functions. It was developed by Shahriar et al. [246] using TCL script and provided 8 mutation operators representing String Format faults. However, it only supports the basic mutation processes without the detection of equivalent mutant.

Kim et al. was the first to implement Java mutation tool [145]. They extended Mothra tool set with Java mutation operators to generate mutants for Java programs. Chevalley and Thevenod-Fosse developed another mutation testing for Java called JavaMut which implements 26 traditional and object-oriented mutation operators. A comprehensive Java mutation testing, MuJava, was developed by the Korean Advanced Institute of Science and Technology (KAIST) and George Mason University [168, 206, 169]. MuJava was designed as a general Mutation Testing tool for Java programs, supporting the entire mutation process. It employed the Weak Mutation and Mutant Schemata techniques, and provided Behavioural Class mutation operators and Structural Class mutation operators using Selective Mutation. A novel Bytecode Translation technique was also adopted by MuJava to reduce the computational cost. Instead of making changes on source code, this technique generates mutant by changing Java Bytecode directly [168].

MUGAMMA is a Java mutation extension of the GAMMA framework, developed by Kim et al. [147]. Unlike traditional mutation testing, it is designed to perform post deployment mutation analysis. It generates mutants in the field dynamically, and uses user's real time inputs as test data to determine if the mutants can be killed. It captures the results of each execution to provide additional confidence in the deployed system.

TDS is a mutation tool for testing Distributed Component-Based applications, developed by Ghosh et al. [107]. TDS applies Interface Mutation techniques to test the interfaces between two components which defined using CORBA IDL. SESAME is multi-language mutation tool developed by LAAS-CNRS group [53]. It supports to inject faults into assembly languages, procedural languages, such as Pascal, C and data-flow, such as Lustre. SESAME has now been used by a commercial testing tool, IBM Rational Test RealTime, to asses the quality of test sets. ExMAn is another multi-language mutation tool developed by Bradury et al. [32]. ExMan supports to generate mutants for C and Java programs using source transformation language TXL. It can also work as a mutation framework to support plugin of other quality assurance tool, such as model checkers and static analysis tools.

SQLMutation was the first mutation testing tool for SQL statements. It supports four type of mutation operators and provides a web application interface as well as a web service interface. MUSIC was developed by Shahriar et al. [244]. It was designed to detect SQL Vulnerabilities by injecting faults into SQL statements in JSP applications. It applied 9 SQL mutation operators using Selective Mutation. Weak Mutation was also adopted to reduce the computational cost by checking the internal results returned after query executions immediately.

The CREAM system was the first mutation testing tool for C# programs. It was proposed by Derezinska and Szustek [76]. The CREAM system implemented five object-oriented mutation operators (EOC, IHD, IPC, IOP, and JID). ESTP was designed by Feng et al. [96] to measure the effectiveness of existing testing strategies. In ESTP, a testing strategy is transformed into tabular specifications which are used to generate test cases automatically. To evaluate this test strategy, the generated test cases are evaluated by mutation testing using 20 of Agrawal et al's 77 C mutation operators [6].

Milu is another mutation testing tool for C programs, developed by Jia and Harman [137]. Milu was the first mutation tool which supports Higher Order Mutation Testing [136, 138]. To reduce the inherited computational cost of the traditional Mutation Testing, Milu provides a set of search based optimisation algorithms to search the subtle higher order mutants effectively. A novel test harness embedding technique was proposed to reduce the cost of the execution of mutants [137]. MILU is also the first tool to use Search Based Software Engineering (SBSE) to optimise mutation testing, although SBSE has been widely used to optimise other aspects of testing [116, 179]. The implementation of MiLu is presented in Section 4.2.

JAVALANCH is a mutation testing tool for Java, developed by Schuler et al. [113, 238]. The design of JAVALANCH focused on the automation and scalability. To reduce computational cost, the mutants are created in the Java binary code level using Selective Mutation and Mutant Schemata techniques. Moreover, JAVALANCH applies a new form of Weak Mutation approach which analyses the mutants using local invariants [238]. This approach has also been used to prioritize the execution of the non-equivalent mutants [113].

3.3.2 Industry and Open Source Tools

JESTER is the first open source tool for Java [182]. It only provides two mutation operators. One changes 0 to 1 and the other is to replace predicates with TRUE and FALSE [181]. A Python version of Jester, PESTER is available from the JESTER's website [182], and a C# version, NESTER is also available [251].

HECKLE is an open source unit mutation tool for Ruby at the RubyForge [236]. It currently supports mutation of booleans, number, strings, symbols, ranges, regexes and branches for entire classes, or individual methods. After running the mutation analysis automatically, it provides a simple report summarising statistical results. A detailed implementation document on the HECKLE can be found on its website [236].

JUMBLE is a class level Java Mutation Testing tool, developed by a commercial company called ReelTwo [233] from 2003 to 2006, and it has been released as an open source project under the GPL licence since 2007 [252]. JUMBLE supports seven types of mutation, including mutation on predicate conditions, arithmetic operators, increments, inline constants, class poll constants, return values and switch statements. It also provides a visualized feedback using a web based interface.

GAMERA is mutation testing framework for WS-BPEL. It was first designed as an academic tool by Domingues et al. [81] and released as an open source tool recently. It uses 26 WS-BPEL mutation operator and adopted genetic algorithm to reduced the number of mutants. MUTATEME is a mutation framework for PHP5 web applications. It is the most recent open source mutation tool and is still in alpha version, under testing. The detailed implementation and source code is available from its website [35].

INSURE++ was the earliest commercial automatic testing tool for C and C++ using mutation analysis technique [226]. Instead of generating all possible mutants, INSURE++ targets on the 'potential equivalent mutants' which have same behaviours as the original program. The idea is to then tries to generate the test cases to kill these mutants, and if any test case is able to killed the 'potential equivalent mutant', it might also finds the bug in the original program. INSURE++ applies a Source Code Instrumentation technique to optimise the performance [226].

PLEXTEST is a commercial Mutation Testing tool for C/C++ programs [132]. It implements the traditional mutation testing engine with a unit testing framework. It supports the entire mutation process as well as Selective Mutation, Mutant Schemata, and Weak Mutation to reduce computational cost.

CERTITUDE is the most recent commercial tool, developed by Certess Inc [45]. CERTITUDE is primarily designed for Electronic Design Automation (EDA) and provides a functional qualification as a verification criteria. It combines mutation testing along with static analysis to measure and improve the functional qualification for HDL functional verification.

3.4 Discussion

In the Redwine-Riddle maturation model [232], there is a trend that indicates that a technology takes about 15 to 20 years to reach a level of maturity at which time industrial uptake takes place. Suppose we cast our attention back by 15 years to the mid 1990s. We reach a point where only approximately 25% of the current volume of output had then been published in the literature. (see Figure 3.3). The ideas found in this early Mutation Testing literature have now been implemented in practical commercial Mutation Testing tools, as shown in Table 3.3. This observation suggests that the development of mutation testing is in line with Redwine and Riddle's findings.

Furthermore, the set of mutation testing systems developed in the laboratory now provides tooling for a great many different programming language paradigms (as shown in Table 3.3). This provides further evidence of maturity and offers hope that, as these tools mature, following the Redwine and Riddle model, we can expect a future state–of–practice in which a wide coverage of popular programming paradigms will be covered by real world mutation testing tools.

Finally, an increasing level of maturity can also be seen in the development of the empirical studies reported on mutation testing. For example, there is a noticeable trend for empirical studies to involve more programs and to also involve bigger and more realistic programs, as can be seen in the chronological data on empirical studies presented in Figure 3.8 and 3.9. However, it should also be noted that more work is required on real world programs and that many of our empirical evidence still rests on studies of what would now be regarded as 'toy programs'. There also appears to be an increasing degree of corroboration and replication of the results reported (see Table 2.4).

Summary

This chapter has provided a detailed analysis of trends and results on mutation testing. There has been much optimisation to reduce the cost of the mutation testing process. From the data we collected from and about the mutation testing literature, our analysis reveals an increasingly practical trend in the subject. We also found evidence that there is an increasing number of new applications. There are more, larger and more realistic programs that can be handled by mutation testing. Recent trends also include the provision of new open source and industrial tools. These findings provide evidence to support the claim that the field of mutation testing is now reaching a mature state.

Recent work has tended to focus on more elaborate forms of mutation than on the relatively simple faults that have been previously considered. There is an interest in the semantic effects of mutation, rather than the syntactic achievement of a mutation. This migration from the syntactic achievement of mutation to the desired semantic effect has raised interest in higher order mutation to generate subtle faults and to find those mutations that denote real faults. Next chapter will present a fault-based higher order mutant classification, which can be used to located higher order mutants that denote real faults.

Chapter 4

Higher Order Mutants Classification

It is widely believed that higher order mutants are too numerous to be practical as a source of simulated faults. Furthermore, many authors claim that the coupling of higher order mutants to first order mutants renders higher order mutants unimportant. This chapter will investigate higher order mutants from the perspective of fault interactions. Each first order mutant is considered to be a single fault, and higher order mutants are considered to be a combination of single faults. To capture the behaviour of each fault, both first and higher order mutants are executed with a test suite. Higher order mutants are then classified according to the changes in the behaviour of the faults caused by interactions between the faults.

This chapter will introduce an open-source mutation testing tool, MiLu, available on the MiLu website [1]. MiLu is specifically designed for higher order mutation testing of C programs, and, although designed to support higher order mutation testing, it is also an efficient and flexible tool for first order mutation testing. MiLu adopts the 77 C mutation operators of Agrawal et al. [6], and it provides customised mutation operators. The main contributions of the chapter are as follows:

1. A new classification of higher order mutants is introduced from a fault interaction perspective. A theoretical model for second order mutants is developed, and second order mutants are then classified systematically in a tree hierarchy structure.

2. The proportion of different categories of all second order mutants and samples of third to fifth order mutants are explored in six subject programs. In total, more than two million higher order mutants were generated with 9.2 billion test executions. The results demonstrate that a large proportion of the behaviour of higher order mutants is changed due to fault interaction.

3. MiLu, a higher order mutation testing tool, is presented. The efficacy of MiLu in mutant generation and execution is investigated and the performance of MiLu in both single and multi processing mode is reported.

The remainder of this chapter is organised as follows. Section 4.1 formally introduces the classification of higher order mutants. Section 4.2 presents a higher order mutation testing tool. Section 4.3 describes the experimental setting while the results are discussed in Section 4.4.

4.1 Higher Order Mutant Classification

There exist a very large number of higher order mutants, and it is therefore impractical to use all possible higher order mutants in mutation testing. Classification of higher order mutants can assist in the identification of those higher order mutants which can potentially benefit from higher order mutation testing. For the sake of simplicity, in this chapter, a second order mutant case is used to illustrate the proposed HOM classification. However, this approach can also be used to classify n^{th} order mutants, where $n > 1$. This section will first introduce this second order mutant case and then describe the classification approach in detail.

4.1.1 Second Order Mutant Case

The particular second order mutant case considered in this chapter is a simple but typical example of higher order mutants. Assume there are two first order mutants, f_1 and f_2, then h denotes the higher order mutant constructed from the first order mutants f_1 and f_2. Assume the existence of a test set T_u; T_u denotes the universal set of all possible test data. Test sets T_{f1}, T_{f2} and T_h denote the set of test data that kill the first order mutants f_1, f_2 and the higher order mutant h, respectively. Figure 4.1 shows a Venn diagram of such a typical second order mutant case.

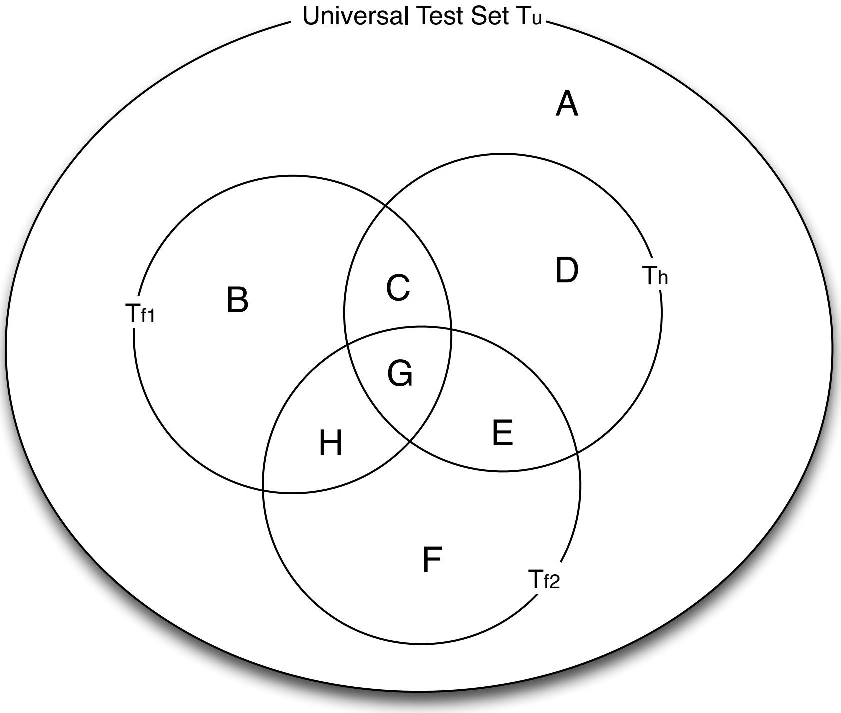

Figure 4.1: Second order mutant case Venn diagram

In Figure 4.1, the rectangle T_u depicts the set of all possible test inputs. Three circles, T_{f1}, T_{f2} and T_h, depict the possible regions of test cases that can kill mutants f_1, f_2 and h, respectively. The regions $A-F$ represent unique sets of test data in the universal test data domain. For example, the test set C includes all test cases that kill both mutants f_1 and h.

These unique test sets are used in the classification proposed in this chapter to illustrate each type of higher order mutant. For example, if the test sets A, B, D, F and H are empty, and the test sets C, G and E are non-empty, the higher order mutant h is a commonly occurring higher order mutant which is predicted by the coupling effect hypothesis. The formal mathematical notation and the equivalent textual descriptions of these regions are provided in Table 4.1.

Table 4.1: Description of the unique test sets

Test Set	**Mathematical Notation**	**Textual Description**
A	$T_u \backslash (T_{f1} \cup T_{f2} \cup T_h)$	Denotes the set of test data that do not kill any of mutants f_1, f_2 and h.
B	$T_{f1} \backslash (T_{f2} \cup T_h)$	Denotes the set of test data that kill only FOM f_1.
C	$(T_{f1} \cap T_h) \backslash T_{f2}$	Denotes the set of test data that kill only FOM f_1 and HOM h.
D	$T_h \backslash (T_{f1} \cup T_{f2})$	Denotes the set of test data that kill only HOM h.
E	$(T_{f2} \cap T_h) \backslash T_{f1}$	Denotes the set of test data that kill only FOM f_2 and HOM h.
F	$T_{f2} \backslash (T_{f1} \cup T_h)$	Denotes the set of test data that kill only FOM f_2.
G	$T_{f1} \cap T_{f2} \cap T_h$	Denotes the set of test data that kill FOMs f_1, f_2 and HOM h simultaneously.
H	$(T_{f1} \cap T_{f2}) \backslash T_h$	Denotes the set of test data that kills only FOMs f_1 and f_2 but not HOM h.

A bar notation is used in the proposed classification of higher order mutants in order to show whether a unique test set is empty or not. Let us assume that the test set T contains test data that kill the mutant M. In the bar notation used here, $\overline{T}$ means that the test set T is not empty and $\underline{T}$ means that the test set T is empty. For example, the bar notation of the common higher order mutant following the coupling effect hypothesis is $\underline{A}\underline{B}\overline{C}\underline{D}\overline{E}\underline{F}\overline{G}\underline{H}$ and the Venn diagram is shown in Figure 4.2.

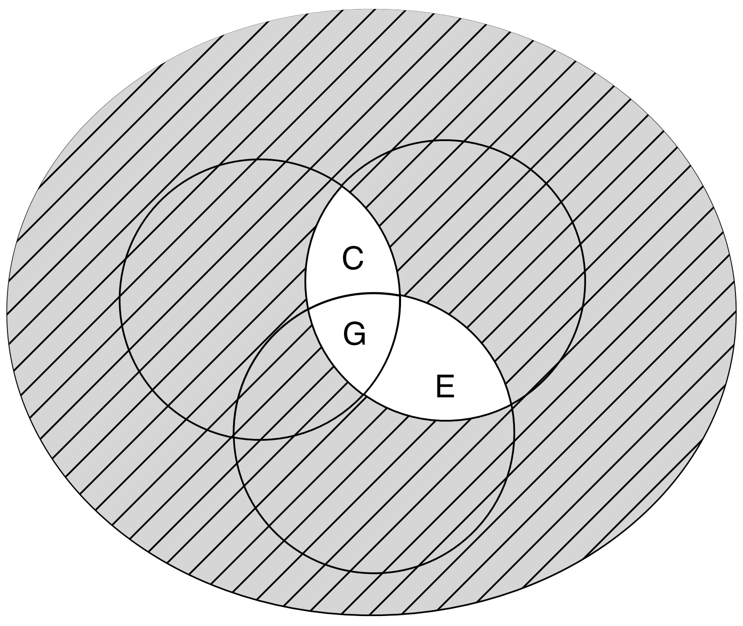

Figure 4.2: Example of the bar notation $\underline{\mathrm{A}}\underline{\mathrm{B}}\overline{C}\underline{\mathrm{D}}\overline{E}\underline{\mathrm{F}}\overline{G}\underline{\mathrm{H}}$. The shaded area depicts empty test sets. The diagram shows that the higher order mutant h is killed by the union of test sets that kill first order mutants f_1 and f_2. (See Table 4.1 for the explanations of test sets A - H)

4.1.2 Higher Order Mutant Classification

From a testing perspective, some types of higher order mutants are more interesting then others. Higher order mutants can be divided into two groups, interesting and uninteresting, as shown in Figure 4.3. The uninteresting group contains higher order mutants that we believe are of no assistance in fault-based testing. For example, if two faults are combined and the program (higher order mutants) is predicted by the coupling effect hypotheses or get even worse (i.e. it becomes more faulty than we expected), these higher order mutants are said to be uninteresting. This is because they cannot help the programmer find any new faults for most of the time. The interesting group contains types of higher order mutants that are potentially able to assist the programmer in fault-based testing. This interesting group can be further subdivided into two groups: fault Masking and fault Shifting. The formal definition of these classes is given in Table 4.2.

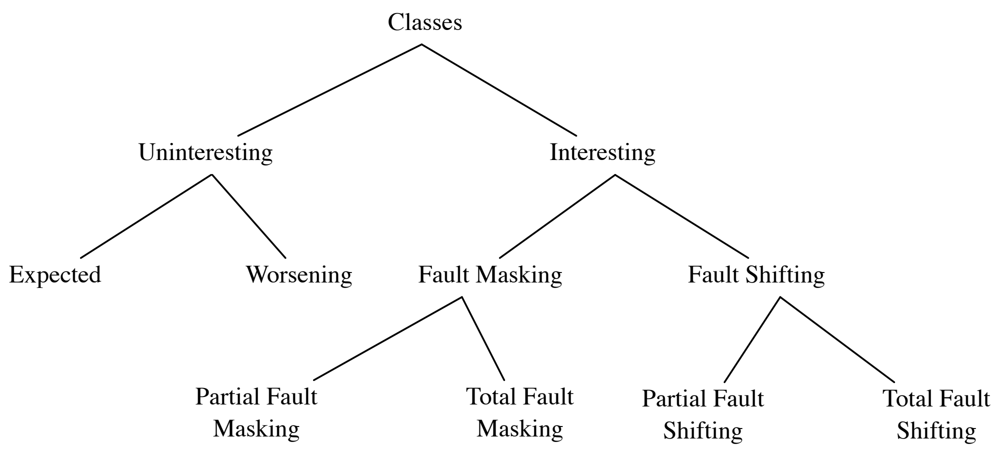

Figure 4.3: Tree of Classes

Second order mutants will now be used to illustrate this classification theoretically. All types of second order mutants were first enumerated systematically in a hierarchy tree structure. The working out of the full tree is shown in Figures 4.4. At the first level of the tree, three combinations of the first order mutants were considered: (i)

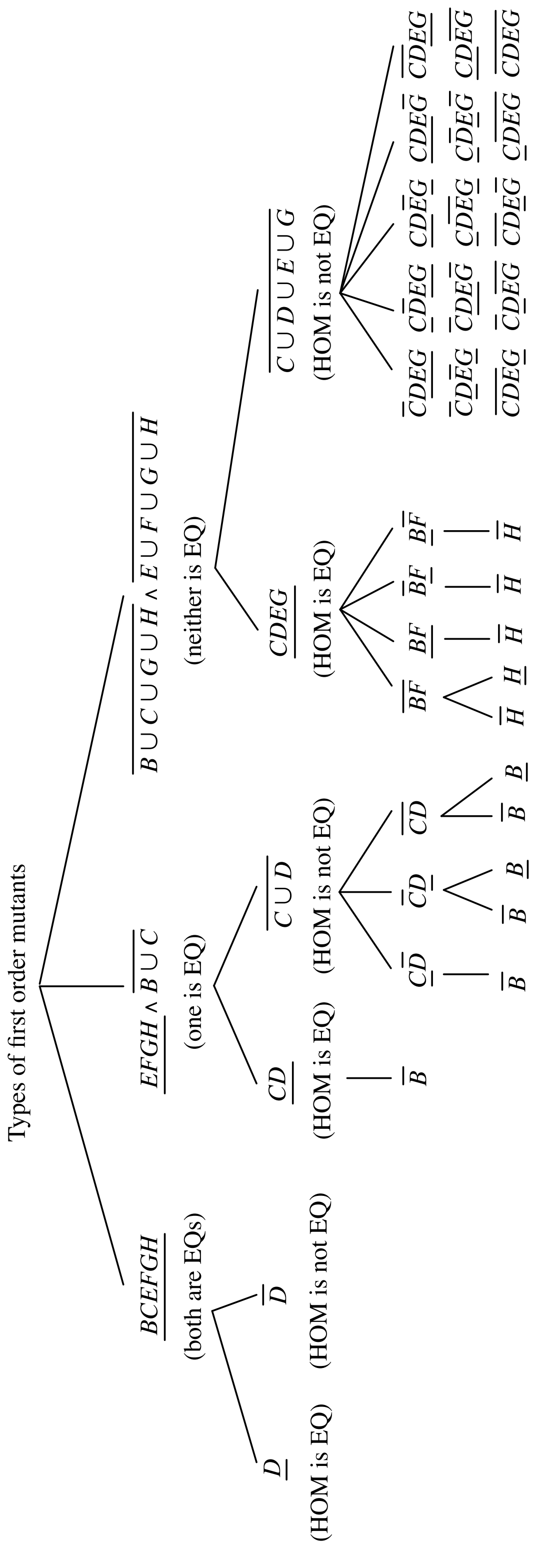

Figure 4.4: Second order classification tree

Table 4.2: Description of the HOM classes

Type	**Description**
Expected	As predicted by the coupling effect hypothesis. $T_{f1} \cup T_{f2} = T_h$
Worsening	From a bug perspective, two faults are combined so that the program gets even worse, i.e. it becomes more 'buggy' than expected. $T_{f1} \cup T_{f2} \subset T_h$
Partial Fault Masking (PFM)	Two mutants are combined so that they mask each other, i.e. no new test cases cause the program to fail, and some old test cases pass the test. $T_{f1} \neq \emptyset \wedge T_{f2} \neq \emptyset \wedge T_h \subset T_{f1} \cup T_{f2}$
Total Fault Masking (TFM)	A special case of PFM, where the faults completely mask each other. $T_{f1} \neq \emptyset \wedge T_{f2} \neq \emptyset \wedge T_h = \emptyset$
Partial Fault Shifting (PFS)	Two mutants are combined so that there exist new test cases that cause the program to fail, and some old test cases pass the test. $T_{f1} \neq \emptyset \wedge T_{f2} \neq \emptyset \wedge T_h \neq \emptyset \wedge T_h \setminus (T_{f1} \cup T_{f2}) \neq \emptyset \wedge T_h \cap (T_{f1} \cup T_{f2}) \neq \emptyset$
Total Fault Shifting (TFS)	A special case of PFS, where all old test cases pass the test. $T_{f1} \neq \emptyset \wedge T_{f2} \neq \emptyset \wedge T_h \neq \emptyset \wedge T_h \setminus (T_{f1} \cup T_{f2}) \neq \emptyset \wedge T_h \cap (T_{f1} \cup T_{f2}) = \emptyset$

both first order mutants f_1 and f_2 are equivalent, (ii) one of the first order mutants is equivalent (here assuming first order mutant f_2 is equivalent) and (iii) neither first order mutant f_1 or f_2 is equivalent. At the next level of the tree, each of these situations was further subdivided into two branches: (i) higher order mutant h is equivalent and (ii) higher order mutant h is not equivalent, Finally, all types of higher order mutant were enumerated by all possible combinations of the test sets regions identified in Figure 4.1. The classification based on the tree is shown below.

Both of the FOMs are equivalent, $\underline{BCEFGH}$

1. $\underline{D}$ (HOM h is equivalent) : Expected
2. $\overline{D}$ (HOM h is not equivalent) : Worsening

One of the FOMs (f_2) is equivalent, $\underline{EFGH} \wedge \overline{B \cup C}$

1. $\underline{CD}$ (HOM h is equivalent) :-
    (a) $\overline{B}$: Total Fault Masking
2. $\overline{C \cup D}$ (HOM h is not equivalent) :-
    (a) $\underline{C}\overline{D}$:-
        i. $\overline{B}$: Total Fault Shifting
    (b) $\overline{C}\underline{D}$:-
        i. $\overline{B}$: Partial Fault Masking
        ii. $\underline{B}$: Expected
    (c) $\overline{CD}$:-
        i. $\overline{B}$: Partial Fault Shifting
        ii. $\underline{B}$: Worsening

Neither of the FOMs is equivalent ($\overline{B \cup C \cup G \cup H} \;\wedge$ $\overline{E \cup F \cup G \cup H}$)

1. $\underline{CDEG}$ (HOM h is equivalent) : Total Fault Masking
2. $\overline{C \cup D \cup E \cup G}$ (HOM h is not equivalent) :-
   (a) $\overline{C}\underline{DEG}$: Partial Fault Masking
   (b) $\underline{C}\overline{D}\underline{EG}$: Total Fault Shifting
   (c) $\underline{CD}\overline{E}\underline{G}$: Partial Fault Masking
   (d) $\overline{C}\underline{DE}\overline{G}$: Partial Fault Masking
   (e) $\overline{CD}\underline{EG}$: Partial Fault Shifting
   (f) $\overline{C}\underline{D}\overline{E}\underline{G}$: Partial Fault Masking
   (g) $\overline{C}\underline{DE}\overline{G}$: Partial Fault Masking
   (h) $\underline{C}\overline{DE}\underline{G}$: Partial Fault Shifting
   (i) $\underline{C}\overline{D}\underline{E}\overline{G}$: Partial Fault Shifting
   (j) $\underline{CD}\overline{EG}$: Partial Fault Masking
   (k) $\overline{CDE}\underline{G}$: Partial Fault Shifting
   (l) $\overline{CD}\underline{E}\overline{G}$: Partial Fault Shifting
   (m) $\overline{C}\underline{D}\overline{EG}$: Partial Fault Masking
   (n) $\underline{C}\overline{DEG}$: Partial Fault Shifting
   (o) $\overline{CDEG}$: Partial Fault Shifting

In the second order mutant case, fault Masking and fault Shifting can be further classified based on mutant equivalence. Because instead of considering an equivalent mutant as a fault, it can also be considered as a meaning preserving patch. A fault Masking second order mutant with one equivalent first order mutant and one non-equivalent first order mutant is known as a 'fault Fixing' while a fault Shifting second order mutant with one equivalent first order mutant and one non-equivalent first order mutant is known as a 'fault Transforming', as shown in Table 4.3. It is important to note that this concept only applies to second order mutants.

Table 4.3: Description of special second order classes

Type	**Description**
Partial fault Fixing (PFF)	A patch (equivalent mutant) is applied to a fault (non equivalent mutant) resulting in an improvement, i.e. no new test cases cause the program to fail, and some old test cases pass the test. $T_{f1} \neq \emptyset \ \wedge \ T_{f2} = \emptyset \ \wedge \ T_h \subset T_{f1}$
Total fault Fixing (TFF)	A special case of PFM, where all old test cases pass the test. $T_{f1} \neq \emptyset \ \wedge \ T_{f2} = \emptyset \ \wedge \ T_h = \emptyset$
Partial fault Transforming (PFT)	A patch (equivalent mutant) is applied to a fault (non equivalent mutant) so that new test cases exist that cause the program to fail, and some old test cases pass the test. $T_{f1} \neq \emptyset \ \wedge \ T_{f2} = \emptyset \ \wedge \ T_h \neq \emptyset \ \wedge \ T_h \setminus T_{f1} \neq \emptyset \ \wedge \ T_h \cap T_{f1} \neq \emptyset$
Total fault Transforming (TFT)	A special case of PFS, where all old test cases pass the test. $T_{f1} \neq \emptyset \ \wedge \ T_{f2} = \emptyset \ \wedge \ T_h \neq \emptyset \ \wedge \ T_h \setminus T_{f1} \neq \emptyset \ \wedge \ T_h \cap T_{f1} = \emptyset$

4.2 Milu: Higher Order Mutation Tool

In the literature of mutation testing, there was only one mutation testing tool, Mothra [197] which supports to generate higher order mutants. This section introduces a new higher order mutation testing tool, MiLu, which is specifically designed for the study of higher order mutants in C programs and supports mutation testing in general. MiLu currently supports a subset of mutation operators for the C language [6] by default, and provides a set of APIs to implement user-defined mutation operators. The tool provides a source code analysis and program testing environment to support full mutation testing of either first order mutants, higher order mutants or both, and it is fully open source and available from the MiLu website [1].

Milu (in Chinese characters: 麋鹿) is the name of a deer that is, according to common folklore, composed of four other animal parts: a horse's head, a deer's antlers, a donkey's body and a cow's hooves. This real life animal is sometimes also known as Père David's Deer (Elaphurus davidianus) [36]. The construction of the Chinese name Milu illustrates a higher order mutant where the mutation operators of nature have been applied four times. Furthermore, the Milu deer is currently a critically endangered species, so the program name MiLu also signifies the characteristics of an interesting class of higher order mutants; rare but valuable.

MiLu provides two modes for mutation testing: traditional mode and higher order mode. Traditional mode is designed to support first order mutation testing. In this mode, one fault is seeded in each mutant. In higher order mode, multiple faults are seeded in each mutant. MiLu allow users to use either predefined mutation operators or their own implemented mutation operators. To automate the testing process, the user also needs to specify a comparison method, known as a driver, distinguishing the results between the mutants and the original program. MiLu then takes care of the rest of the work; it generates the mutants, executes each of them with the given test set and reports the mutation score and other information that may be of use to an experimenter. MiLu provides a set of APIs with detailed documentation for researchers programming for their own needs for the generation and evaluation of mutants. The mutation process adopted by MiLu is illustrated in Figure 4.5.

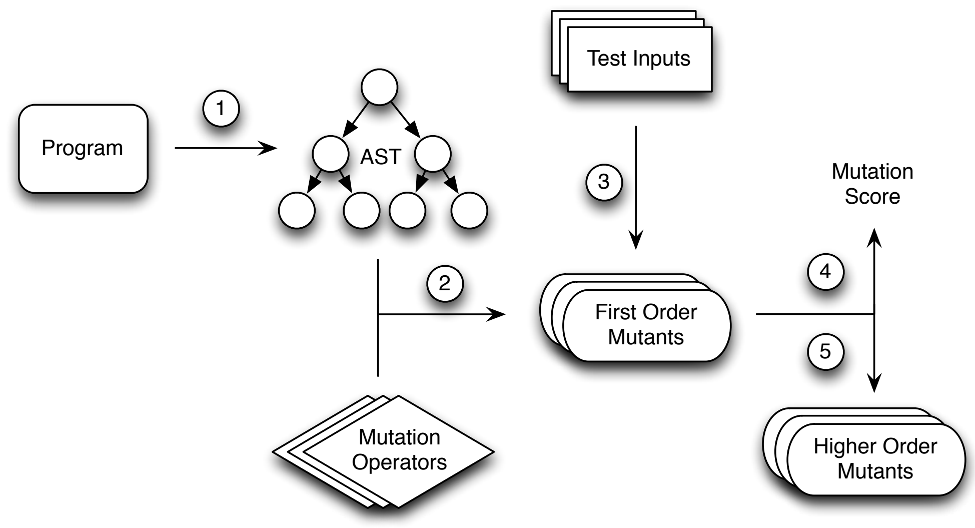

Figure 4.5: MiLu mutation processes

Mutation Process

The first step in Figure 4.5 is to parse source code into an abstract syntax tree (AST). MiLu uses the C library of Clang, the C front-end for the LLVM compiler [156], to parse the source code, thus it can mutate any C program which can be compiled by Clang. In the second step, mutants are generated by modifying nodes of the AST. By default, MiLu supports a subset of C mutation operators [6]. Users are able to implement new mutation operators by creating the mutator objects. Each mutator specifies how to modify nodes of the AST to create mutants and how to clean up the mutation process. A set of AST modification APIs are provided for users to implement mutators; detailed documentation is available on the MiLu website [1].

Step 3 of Figure 4.5 is to run mutants against a set of test data. By default, MiLu supports two types of mutation execution strategies: practical execution strategy and research execution strategy. The practical execution strategy is designed for the general mutation process. This strategy will stop executing a mutant if a test case kills the mutant. The research execution strategy requires execution of a mutant with all given test cases completely even the mutant is killed. The research execution strategy is much more time consuming but it provides additional attributes for each mutant.

To utilise the power of a multicore system, MiLu supports running multiple mutants simultaneously. There are two common ways to run mutants in parallel on a single machine: multi-threading and multi-processing. The multi-threading approach runs mutated programs within a process, while the multi-processing approach runs mutated programs as multiple operating system processes. Multi-threading is more lightweight than multi-processing, as threads have less context switching cost than processes. However, running mutants using processes is more stable, as any mutant crashes a thread may bring down the main mutation tool process. MiLu is designed to be a stable mutation system, thus the mutation execution process is implemented in multi-processing, and it allows users to choose the number of additional processes to run according to users' computer settings. The results of comparing single processing and multi-processing are reported in Section 4.4.3.

4.3 Empirical Study

This section describes a set of experiments designed to explore the properties of higher order mutants. Section 5.4 discusses the research questions that the study will address. Section 5.4 describes the subject programs used in this study. Section 4.3.3 briefly overviews the selected mutation operators.

4.3.1 Research Questions

This section sets out the research questions addressed in the empirical study and for which the next section provides answers.

RQ1: What is the distribution of different classes of second order mutants?

The main aim of this chapter is to classify higher order mutants from a testing perspective. Such classification can be used to identify the interesting higher order mutants which exhibited unusual behaviour due to fault interactions. Therefore, the natural first research question is to investigate the distribution of all mutants over the class of second order mutants. In particular, the proportion of Fault Masking and Fault Shifting higher order mutants will be reported since these higher order mutants might be able to assist the programmer in finding new faults.

RQ2: What is the distribution of different classes of third to fifth order mutants?

Due to the large number of third to fifth order mutants, it is impossible to enumerate all of them. We therefore sampled ten subsets of third to fifth order higher order mutants. Algorithm 1 sets out the steps involved in the experimental procedure. The second research question investigates the distribution of each class of third to fifth order mutants.

RQ3: How efficient is MiLu for mutant generation and mutant execution?

In the experiment, we compare the running time of MiLu using the research execution strategy and the standard execution strategy. We also studied the execution cost in single processing mode and multiple processing mode.

4.3.2 Subject Programs

The experiment described above was performed with six programs: `Mid`, `Find`, `Triangle`, `TCAS`, `Totinfo` and `Replace`. Because of the high computational cost of executing all second order mutants and sampling the higher order mutants, only small and medium-sized programs were used. In total 2,014,699 mutants were generated in the experiment.

```
for each subject program do
    generate all possible first order mutants, $F$
    generate all possible second order mutants, $H_2$
    execute $F$ and $H_2$ on all available tests
    classify mutants $H_2$
    for repeat 10 times do
        randomly generate $n$ 3-5th order $H_{3-5}$, n = $\frac{|H_2|}{10}$
        execute $H_{3-5}$ on all available tests
        classify mutants $H_{3-5}$
    end
end
```

Algorithm 1: Experimental procedure

`Mid`, `Find` and `Triangle` are three small programs used in previous studies on the coupling effect hypothesis by Offutt [197]. The `Mid` program takes three integers as input and outputs the middle value. The `Triangle` program is used to determine the type of a triangle from the length of its sides. The `Find` program takes an integer array A and an index value i and sorts the array so that any element on the left of A(i) is less than or equal to A(i) and any element on the right of A(i) is greater than or equal to A(F).

`TCAS`, `Totinfo` and `Replace` are three larger programs from the 'Siemens Suite' which can be downloaded from the Software-artifact Infrastructure Repository (SIR) [78]. These programs are widely used as a benchmark for software testing techniques. `TCAS` is a program used to avoid an aircraft collision. `Totinfo` is a program that computes statistics from input data, and `Replace` performs pattern matching and substitution.

In order to capture the fault behaviour, a test suite is needed for each subject program. For programs `TCAS`, `Totinfo` and `Replace`, the 'universe' test pool from SIR which includes 1,608 tests achieving adequate statement coverage, branch coverage and du-path coverage, is used. The programs `Mid` and `Triangle` take three integers as inputs; in this experiment all tuple combination of integers from the domain [-5,5] are enumerated as test inputs. `Find` takes an array with length of 10 as input, and pairwise coverage tests are generated from the domain [-5,5] as test inputs. The characteristics of these programs are shown in Table 4.4.

Table 4.4: Selected Subject Programs

Programs	Scale	No. of Test Cases	No. of FOMs	No. of 2nd HOMs	No. of Sampled 3-5 HOMs	FOM Mutation Score
`Mid`	27 LoC	1,331	30	360	360	83.33%
`Find`	49 LoC	1,671	180	15,712	15,712	69.44%
`Triangle`	55 LoC	1,331	274	36,810	36,810	92.70%
`TCAS`	95 LoC	1,608	266	34,697	34,697	77.44%
`Totinfo`	247 LoC	1,052	516	131,815	131,815	81.39%
`Replace`	492 LoC	5,542	1,257	786,694	786,694	78.52%

4.3.3 Mutation Operators

The study of Agrawal et al. describes the mutation operators for the C language into 77 sets. However, not all of the mutation operators increase the effectiveness of mutation testing. Offutt [212, 203] shows that 5 of 22 Fortran mutation operators used by Mothra are sufficient for effective mutation testing. Andrews et al. applied these operators to generate mutants for C programs [14, 15]. They found that the generated mutants are very good at predicting the detection effectiveness of real faults. In the experiment presented in this chapter, the subset of the C mutation operators that fall into Offutt's five categories [203] will be used, and they are listed in Table 4.5. As the total number of higher order mutants are related to the number of first order mutants, this selective mutation approach will reduce the experiment runtime cost.

Table 4.5: Selected C mutation operators

Mutation Operators	Description
CRCR	Required constant replacement
OAAN	Arithmetic operator mutation
OAAA	Arithmetic assignment mutation
OCNG	Logical context negation
OIDO	Increment/decrement mutation
OLLN	Logical operator mutation
OLNG	Logical negation
ORRN	Relational operator mutation
OBBA	Bitwise assignment mutation
OBBN	Bitwise operator mutation

4.4 Results and Analysis

This section will present the answer to each research question in turn, indicating how the results answer each question.

4.4.1 Answer to RQ1

RQ1 is designed to investigate the quantity of each class of second order mutants. To begin the analysis, each column of Table 4.6 presents the number and percentage of second order mutants found in each category. In general, the expected and worsening categories are of little interest, since the higher order mutant faults that fall into these two categories are not able to assist the programmer in finding new faults.

A total of 67.43% of second order mutants fall into the expected category, and only 4.24% of mutants fall in the worsening category. The fault masking category has 19.2% of second order mutants; within this category, partial fault masking mutants comprise approximately 99.5%, and total fault masking mutants comprise only approximately 0.5%. The number of total fault masking mutants is very small; they are potentially equivalent second order mutants which require additional human effort to detect.

A total of 9.11% of second order mutants fall into the fault shifting category, and within this category, 99.8% of mutants belong to the partial fault shifting class and only 0.02% are total fault shifting mutants. The total fault shifting higher order mutants are decoupled higher order mutants; their very small number further confirms the results of the coupling effect hypothesis as stated by Offutt [197].

4.4.2 Answer to RQ2

RQ2 investigates the quantity of each class of third to fifth order mutants and results are summarised in Table 4.7. Because it is impossible to enumerate all third to fifth order mutants, we randomly sampled ten sets of third to fifth order mutants. The total number of sampled higher order mutants was chosen to be equal to the number of 2nd order mutants for each program (see Algorithm 1 for details). Table 4.7 reports the average percentage of third to fifth order mutants found in each category of the ten samples and the standard deviation in brackets. The comparison between second order mutants and third to fifth order mutants is shown in Figure 4.6. Figure 4.6 suggests that as the order increases, the number of partial fault masking category and partial fault shifting category increases.

A total of 29.63% of third to fifth order mutants fall into the expected category which is less than half of the second order mutants in this category. A total of

Table 4.6: Distribution of different classes for second order mutants

Program	Expected (%)	Worsening (%)	PFM (%)	TFM (%)	PFS (%)	TFS (%)
`Mid`	213 (59.17)	16 (4.44)	115 (31.94)	0 (0.00)	16 (4.44)	0 (0.00)
`Find`	10,179 (64.78)	1,062 (6.76)	2,855 (18.17)	27 (0.17)	1,581 (10.06)	8 (0.05)
`Triangle`	25017 (67.96)	1,514 (4.11)	7,572 (20.57)	43 (0.12)	2,655 (7.21)	9 (0.02)
`TCAS`	19,733 (56.87)	1,480 (4.27)	7,400 (21.33)	62 (0.18)	6,003 (17.30)	19 (0.05)
`Totinfo`	114,977 (87.23)	4329 (3.28)	10,332 (7.84)	100 (0.08)	2,076 (1.57)	1 (0.00)
`Replace`	539,503 (68.58)	21,223 (2.70)	115,513 (14.68)	746 (0.09)	109,657 (13.94)	18 (0.00)
Average	(67.43)	(4.24)	(19.09)	(0.11)	(9.09)	(0.02)

4.83% of higher order mutants become worse which is similar to the second order case. The fault masking category has 38.76% of third to fifth order mutants, which becomes the biggest category. The percentage of Partial Fault Masking third to fifth mutants is about twice as much as the second order case while the total fault masking mutants remain same.

The fault shifting category has 26.78% of third to fifth order mutants. The percentage of partial fault shifting third to fifth mutants is about three times as much as the second order case while the total fault masking mutants remain same. Again the number of total fault shifting higher order mutants is very small, which confirms the results of the coupling effect hypothesis[197].

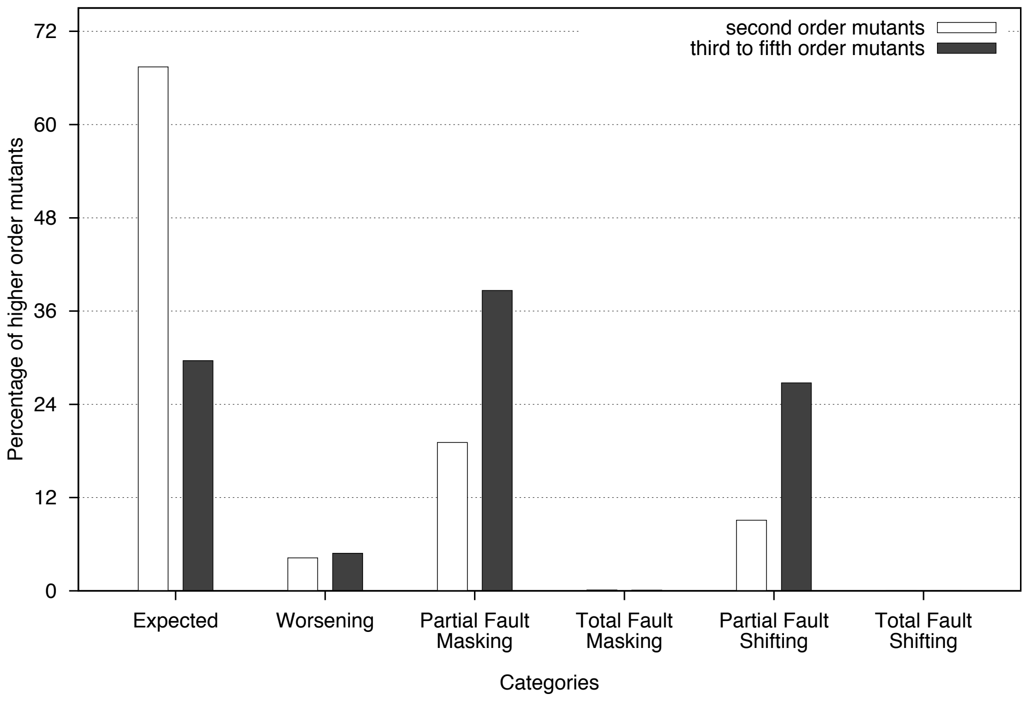

Figure 4.6: A Comparison of categories of 2nd order mutants and 3rd to 5th order mutants.

Table 4.7: Distribution of different classes for the ten samples of third to fifth order mutants. The STD number is the standard divination of ten samples

Program	Expected % (STD)	Worsening % (STD)	PFM % (STD)	TFM % (STD)	PFS % (STD)	TFS % (STD)
`Mid`	12.50 (4.94)	2.78 (2.93)	71.11 (8.20)	0.28 (0.88)	13.33 (5.82)	0.00 (0.00)
`Find`	20.90 (0.64)	9.54 (0.73)	36.80 (1.31)	0.13 (0.14)	32.60 (1.16)	0.02 (0.03)
`Triangle`	22.97 (0.60)	5.78 (0.34)	43.09 (0.80)	0.01 (0.02)	28.13 (1.03)	0.01 (0.01)
`TCAS`	11.91 (0.68)	2.78 (0.24)	34.56 (0.93)	0.11 (0.03)	50.60 (1.38)	0.04 (0.03)
`Totinfo`	74.63 (0.23)	4.74 (0.11)	16.26 (0.37)	0.02 (0.02)	4.35 (0.20)	0.00 (0.00)
`Replace`	34.87 (0.27)	3.375 (0.01)	30.13 (0.03)	0.04 (0.00)	31.58 (0.24)	0.00 (0.00)
Average	29.63	4.83	38.66	0.10	26.77	0.01
Average STD	(1.23)	(0.73)	(1.94)	(0.18)	(1.64)	(0.01)

4.4.3 Answer to RQ3

RQ3 investigates the efficiency of MiLu. Table 4.8 reports the average time for mutant generation and execution for each program. In Table 4.8 the columns labelled 'Gen.' report the average mutant generation time for 100 mutants taken by MiLu (in seconds). The columns labelled 'Exe.' report the average mutant execution time for 100 mutants taken by MiLu (in seconds). The columns labelled 'PRA.' indicate that MiLu runs using the practical execution strategy, that is, MiLu stops running any mutants which are killed by one test case. The columns labelled 'RES.' indicate that MiLu runs using the research execution strategy, that is, MiLu runs the complete test suite on all mutants. The two labels, 'SP' and 'MP', indicate that MiLu runs in the traditional single processing and multi-processing modes, respectively. In this experiment, MiLu executes mutants with 12 processes in the multi-processing mode.

Table 4.8: Efficiency of running MiLu

Program	Gen. (PRA,SP)	Exe. (PRA,SP)	Exe. (RES,SP)	Gen. (PRA,MP)	Exe. (PRA,MP)	Exe. (RES,MP)
`Mid`	0.49	16.25	308.25	0.06	4.66	25.29
`Find`	0.50	48.64	424.42	0.06	5.77	29.25
`Triangle`	0.49	171.47	306.98	0.05	11.58	26.51
`TCAS`	0.54	27.10	372.38	0.09	6.43	31.93
`Totinfo`	0.67	17.36	247.23	0.10	3.92	25.77
`Replace`	0.69	79.95	1419.95	0.09	19.18	103.91
Average	0.56	60.13	513.20	0.07	8.59	40.44

In MiLu, the mutant generation process involves parsing source code, applying syntactic transformation to the source code and outputting mutants. The mutant execution process involves compiling mutants and executing test suites on mutants. As shown in Table 4.8, in general, it is much faster to generate mutants (average times range from 0.07 to 0.56 seconds per 100 mutants) than to execute mutants (average times range from 8.59 to 513.2 seconds per 100 mutants). The practical execution strategy is much faster than the research execution strategy; specifically, compared to the research execution strategy, the practical execution strategy is 8.53 times faster in single processing mode and 4.7 times faster in multi-processing mode, on average. Using 12 processes, compared to the single processing mode, the multi-processing mode achieves 7 times speed increase in the standard execution strategy and 12.68 times speed increase in the research execution strategy on average.

Summary

This chapter introduced a new higher order mutant classification. Based on different types of fault interactions, this approach classifies higher order mutants into four categories: expected, worsening, fault masking and fault shifting. This chapter also presents MiLu, a C mutation testing tool that can handle both first and higher order mutants. The new classification approach was studied empirically using six programs. The results show that interesting fault masking and fault shifting classes of higher order mutants can be found in all of the subject programs.

In this chapter, all possible second order mutants were enumerated in order to identify the interesting mutants. However, this is impossible when running mutation testing due to the high computational cost. The next chapter will introduce a search-based optimisation approach for finding optimal higher order mutants which potentially represent subtle faults in real world programs. This approach has the potential to overcome the exponential explosion in the number of higher order mutants.

Chapter 5

Searching for Higher Order Mutants

The chapter introduces the concept of subsuming higher order mutants. A subsuming higher order mutant is more difficult to kill than the first order mutants from which it is constructed. As such, it may be preferable to replace the first order mutants with the single higher order mutant. In particular, this chapter will introduce the concept of a strongly subsuming higher order mutant. A subsuming higher order mutant is only killed by a subset of the intersection of test cases that kill each first order mutant from which it is constructed. Both subsuming and strongly subsuming higher order mutants belong to the subsets of fault masking and fault shifting higher order mutants, which were introduced in Chapter 4.

Consider a subsuming higher order mutant, h, constructed from the FOMs $f_1, ..., f_n$. The set of test cases that kill h also kill each and every first order mutant $f_1, ..., f_n$. Therefore, h can replace all of the mutants $f_1, ..., f_n$ without loss of test effectiveness. The converse does not hold; there exist test sets that kill all FOMs $f_1, ..., f_n$ but fail to kill h. The first order mutants cannot, even taken collectively, replace the higher order mutant without possible loss of test effort. This is the sense in which h can be said to 'strongly subsume' $f_1, ..., f_n$.

In order to overcome the inherent computational cost that comes with the large number of HOMs, this chapter introduces a search-based approach to identify these subsuming higher order mutants efficiently. The main contributions of the chapter are as follows:

1. A novel higher order mutation testing paradigm is introduced. The concepts of subsuming higher order mutants and a search-based approach to overcome the exponential explosion in the number of higher order mutants are also introduced. The work presented takes advantage of higher order mutation testing which clarifies the differences between the higher order mutation testing paradigm and the first order mutation testing paradigm, as previously studied and practiced.

2. The proportion of all higher order mutants that are subsuming and strongly subsuming is explored. The results show that a large proportion of higher order mutants are subsuming and that a small proportion of these are strongly subsuming. Although the proportion of strongly subsuming mutants is small, the number of strongly subsuming mutants is large because the number of higher order mutants increases exponentially. The search-based algorithms were able to find small but useful numbers of strongly subsuming higher order mutants in all of the ten programs studied.

3. The relationship between mutant killing set intersection and mutant order is investigated. The results demonstrate the degree to which higher order mutants contain first order mutants that are completely decoupled.

4. Three algorithms for finding optimal higher order mutants are introduced. The results indicate that the genetic algorithm performs best overall. However, it

is also demonstrated that each of the algorithms targets a different kind of higher order mutant; therefore, all three algorithms are useful.

The rest of this chapter is organised as follows. Section 5.1 formally introduces the concept of a subsuming higher order mutant. Section 5.2 discussed the advantage of higher order mutant testing. Section 5.3 presents a search-based approach and explains three meta-heuristic algorithms used to find higher order mutants. Section 5.4 details the experimental setting, while the results are discussed in Section 5.5. Section 5.6 discusses threats to the validity of the experiments and the related work.

5.1 Subsuming Higher Order Mutants

Chapter 4 introduced two classes of interesting higher order mutants, fault masking and fault shifting; however, not all higher order mutants that fall into these two classes are suitable for practical use in mutation testing. For example, total fault masking higher order mutants are equivalent mutants should be avoided. To identify a subset of fault masking and fault shifting higher order mutants representing potential real subtle faults, higher order mutants are further classified in terms of the way that they are 'coupled' and 'subsuming', as shown in Figure 5.1. In Figure 5.1, the region area in the central Venn diagram represents the domain of all higher order mutants. The sub-diagrams surrounding the central region illustrate each category. For the sake of simplicity of exposition, these examples illustrate the second order mutant case; it is assumed that there are two first order mutants, f_1 and f_2, and h denotes the higher order mutant constructed from the first order mutants f_1 and f_2. The two regions depicted in each sub-diagram represent the test sets containing all the test cases that kill the first order mutants f_1 and f_2. The shaded area represents the test set that contains all test cases that kill the higher order mutant h. The areas of the regions indicate the proportion of the domain of higher order mutants for each category.

Following the coupling effect hypothesis [197], if a test set that kills the first order mutants also contains cases that kill the higher order mutant, it can be said that the higher order mutant is a ‘coupled higher order mutant’, otherwise it is said to be a ‘de-coupled higher order mutant’. Therefore, in Figure 5.1, the sub-diagram is a coupled higher order mutant if it contains an area where the shaded region overlaps with the unshaded regions, for example as in the sub-diagrams (a), (b) and (f). Since the shaded region from the sub-diagrams (c) and (d) do not overlap with the unshaded regions, (c) and (d) are de-coupled higher order mutants. Sub-diagram (e) is a special case of a de-coupled higher order mutant because there is no test case that can kill the higher order mutant; there is no overlap and thus the higher order mutant is an equivalent mutant.

Subsuming higher order mutants, by definition, are more difficult to kill than their constituent first order mutants. Therefore, in Figure 5.1, the subsuming higher order mutants can be represented as those where the shaded area is smaller than the area of the union of the two unshaded regions, such as in sub-diagrams (a), (b) and (c); in contrast, the higher order mutants represented in (d), (e) and (f) are non-subsuming. Furthermore, the subsuming higher order mutants can be classified into strongly subsuming higher order mutants and weakly subsuming higher order mutants. By definition, if a test case kills a strongly subsuming higher order mutant, it guarantees that its constituent first order mutants are killed as well. Therefore, if the shaded region lies only inside the intersection of the two unshaded regions, it is a strongly subsuming higher order mutant, as depicted in (a), which is a subset of fault masking higher order mutants. Otherwise, it is a weakly subsuming higher order mutant, as depicted in (b) and (c), which are a subset of fault shifting higher order mutants.

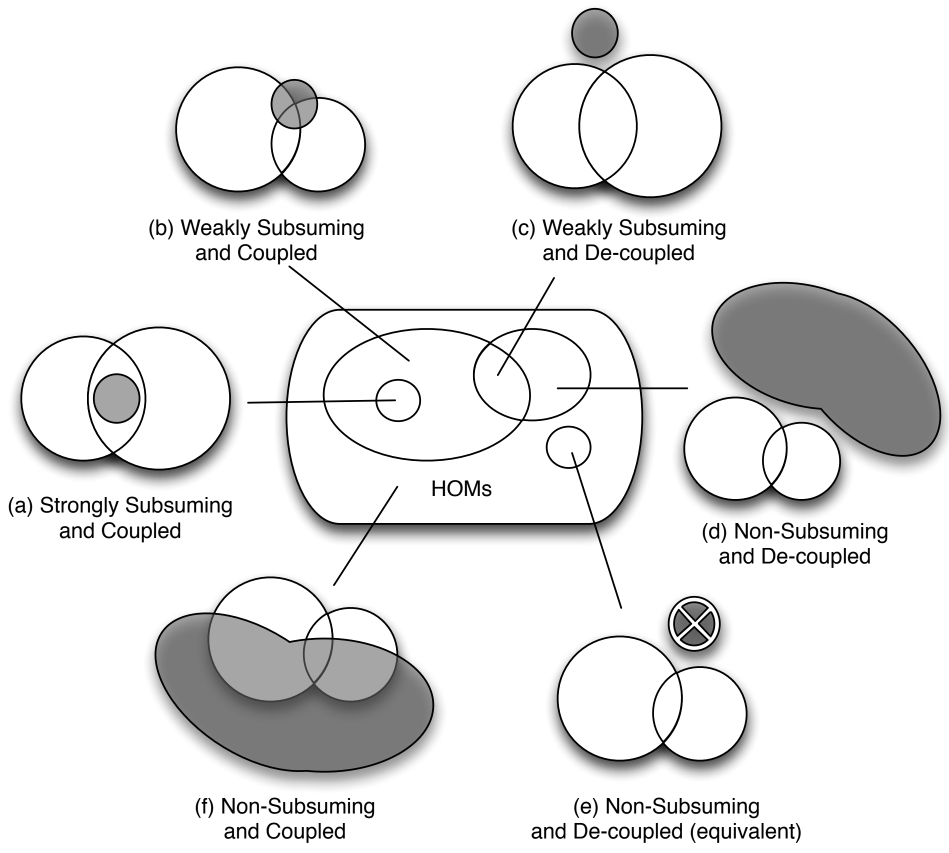
(b) Weakly Subsuming and Coupled
(c) Weakly Subsuming and De-coupled
(a) Strongly Subsuming and Coupled
HOMs
(d) Non-Subsuming and De-coupled
(f) Non-Subsuming and Coupled
(e) Non-Subsuming and De-coupled (equivalent)

Figure 5.1: Subsuming higher order mutant classification. The central Venn diagram depicts important subclasses into which higher order mutants fall, while the outer diagrams depict killing test sets for the higher order mutants (shaded) and their constituent first order mutants (unshaded). For ease of exposition, the diagrams illustrate only the second order case, whereas the definitions cover any order. Higher order mutants of type (a), (b) and (c) are more difficult to kill than their constituent first order mutants, thereby capturing more subtle faults. In particular, type (a) are both subtle and useful; they can replace their constituent first order mutants because they are killed by a subset of the intersection of test cases that kill their constituents.

According to the combination of subsuming and de-coupled higher order mutant types, the six possibilities to be considered are: (a) strongly subsuming and coupled, (b) weakly subsuming and coupled, (c) weakly subsuming and de-coupled, (d) non-subsuming and de-coupled, (e) non-subsuming, de-coupled which is equivalent and (f) non-subsuming and coupled, which is of no use, as shown in Figure 5.1. The formal definitions of these higher order mutants are now given. Let h be a higher order mutant, constructed from first order mutants $f_1, ..., f_n$. The existence of a test set T is assumed; T is the set of all test cases under consideration. T_h is the subset of T that kills the higher order mutant h, while $T_1, ..., T_n$ are the subsets of T that kill the constituent first order mutants $f_1, ..., f_n$, respectively.

Definition 1 (Strongly Subsuming and Coupled)**.**

$$T_h \subset \bigcap_i T_i \ \ and \ \ T_h \neq \emptyset$$

Definition 2 (Weakly Subsuming and Coupled)**.**

$$|T_h| < |\bigcup_i T_i| \ , \ \ T_h \neq \emptyset \ \ and \ \ T_h \cap \bigcup_i T_i \neq \emptyset$$

Definition 3 (Weakly Subsuming and De-coupled)**.**

$$|T_h| < |\bigcup_i T_i| \ , \ \ T_h \neq \emptyset \ \ and \ \ T_h \cap \bigcup_i T_i = \emptyset$$

Definition 4 (Non-Subsuming and De-coupled)**.**

$$|T_h| \geq |\bigcup_i T_i| \ , \ \ T_h \neq \emptyset \ \ and \ \ T_h \cap \bigcup_i T_i = \emptyset$$

Definition 5 (Non-Subsuming and De-coupled)**.**

$$T_h = \emptyset \qquad (Equivalent)$$

Definition 6 (Non-Subsuming and Coupled)**.**

$$|T_h| \geq |\bigcup_i T_i| \ \ and \ \ T_h \cap \bigcup_i T_i \neq \emptyset \qquad (Useless)$$

5.2 Advantages of Higher Order Mutant Testing

At first sight, any move from first order mutants to higher order mutants brings with it an exponential explosion. Since a higher order mutant is constructed by combining different first order mutants, the number of higher order mutants can be computed from the number of first order mutants. For such higher order mutants, let n be the number of places in the program that can be mutated, and $m_{1\ldots n}$ be the number of changes that can be applied at location n. The number of the first order mutants is given by $\sum_{i=0}^{n} m_i$. The number of the ith order mutants is given by $\frac{n! \prod_{x=2}^{n} m_x}{i!}$. Because of this exponential explosion, using higher order mutants has previously been considered to be too computationally expensive to be practical. Furthermore, the coupling hypothesis [68, 196, 197] suggests that the vast majority of higher order mutants will be coupled to first order mutants, such that test sets that kill all first order mutants will also kill almost all higher order mutants.

However, the few higher order mutants that are not coupled to their constituent first order mutants may be very important; they are killed by a different set of test cases than their constituent first order mutants. For decoupled mutants, the act of combining first order mutants *shifts* the fault-revealing test set. Suppose that the act of combining first order mutants to form a decoupled higher order mutant not only shifts the fault-revealing set, but also reduces its size so that the higher order mutant is more difficult to kill than its constituent first order mutants. It is very likely that such a higher order mutant would potentially be valuable in testing. Using the nomenclature introduced in this chapter, it would be termed a ‘subsuming decoupled higher order mutant’.

De-coupling is not the only way to produce a subsuming higher order mutant. Strongly subsuming higher order mutants are, by definition, coupled since the test sets that kill them are subsets of those that kill each of their constituent first order mutants. Therefore, both coupled and decoupled higher order mutants may turn out to be more difficult to kill than the first order mutants from which they are constructed, making them potentially valuable to the mutation testing process. This chapter will focus on the subsuming higher order mutants in general and the strongly subsuming higher order mutants in particular since a strongly subsuming higher order mutant can always be used as a substitute for its constituent first order mutants. It is reasonable to state that higher order mutation testing can reduce test effort.

It might be assumed that, since there are exponentially more higher order mutants than first order mutants, higher order mutation testing would be much more computationally expensive than first order mutant testing. However, it is possible for it to be *less* expensive. This apparent paradox is resolved by targeting specifically those higher order mutants, the strongly subsuming higher order mutants, each of which can be used to replace more than one first order mutant. Fewer (but better) mutants mean fewer (but better) test cases. This higher order mutant testing approach avoids dumb mutants in favour of subtle ones. Of course, in order to find the subtle higher order mutants, it is necessary to first construct *all* of their constituent first order mutants. However, this process is entirely automated by the search-based optimisation approach.

In contrast, the process of checking the original program's output for each mutant-killing test cases often requires a (human) oracle. This oracle cost is often the most expensive part of the overall test activity. The oracle cost can be reduced by reducing the size of the test suite. By moving from the first order to the higher order paradigm, one seeks to reduce the number of mutants considered, simultaneously increasing their quality. This has the potential to reduce test effort while improving effectiveness.

Figure 5.2 illustrates a simple example of using strongly subsuming higher order mutant to reduce test effort and to increase test effectiveness at the same time. Suppose there is a strongly subsuming higher order mutant h which is constructed from the first order mutants f_a and f_b. The two regions T_a and T_b in Figure 5.2 represent the test sets containing all the test cases that kill the first order mutants f_a and f_b, respectively, while the region T_h represents the test set containing all test cases that kill the strongly subsuming higher order mutant h. In traditional mutation testing it is easy to find test cases like t_a and t_b which kill both first order mutants f_a and f_b. However, the test case t_h that kills the strongly subsuming higher order mutant h is a better choice because it kills the first order mutants f_a and f_b both separately and in combination, so a human oracle need only check one test output. Reduction of test effort can also be achieved by some 'smart' techniques with slightly more effort; for example, clustering test cases to identify the intersection of T_a and T_b. Although any test case selected from this intersection can achieve the same test effort as the test cases that kill the strongly subsuming higher order mutant h, such a test case like t_{ab} might not able to find the subtle fault represented by the strongly subsuming higher order mutant h, thereby losing test effectiveness.

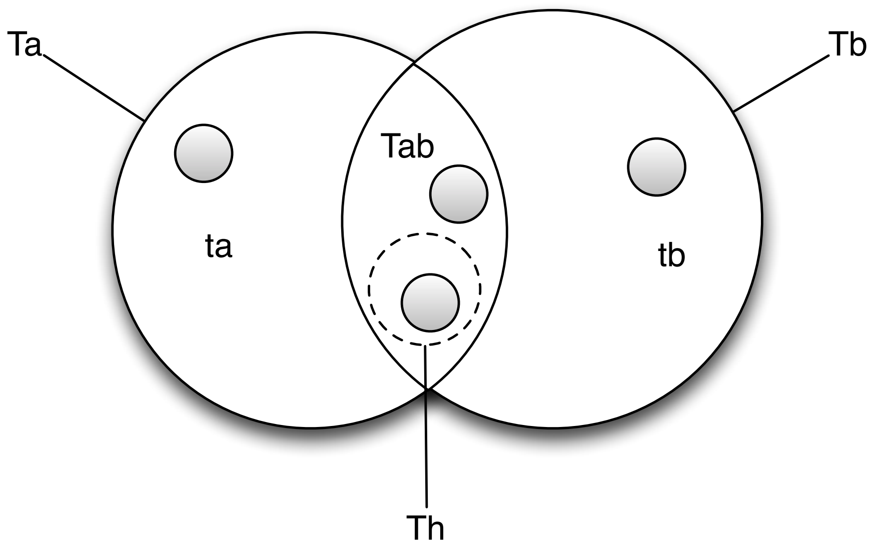

Figure 5.2: Test Effort Reduction Example

5.3 Algorithm

Due to the large number of higher order mutants, it is possible for the computation cost in finding valuable higher order mutants to turn out to be extremely high. Therefore, using a normal undirected search is not sufficiently efficient to find subsuming higher order mutants. In order to find the subsuming higher order mutants more effectively, the proposed approach uses three meta-heuristic algorithms (GR, GA, HC). This section will introduce the representation and fitness function first and, then, explain the three meta-heuristic algorithms in detail.

Representation

To identify a higher order mutant uniquely, two parameters must be specified: the position at which to mutate and the mutation operator to be applied. In the proposed approach, higher order mutants are represented as a vector of *MutationId* data type. Each *MutationId* contains two integers representing the location of the mutant and the type of the mutant, respectively. An example of the data representation for a second order mutant is shown in Figure 5.3.

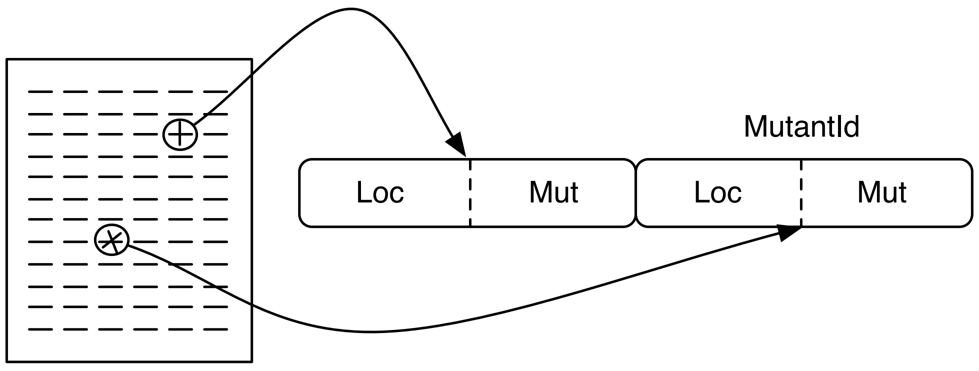

Figure 5.3: Data representation for a second order mutant.

Fitness Function

In order to measure the fitness of the higher order mutant, a value is needed that measures the ease with which a first or higher order mutant can be killed. Let T be a set of test cases, $\{M_1,...,M_n\}$ be a set of mutants, and the $kill(\{M_1, ..., M_n\})$ function returns the set of test cases which kill the mutants $M_1, ..., M_n$. Fragility will be defined for a set of mutants so that a single definition caters for individual mutants (which may be either first order *or* higher order), but also for sets of individual mutants, such that the fragility of a mutant shall be defined as follows:

Definition 7 (Fragility)**.**

$$fragility(\{M_1, ..., M_n\}) = \frac{|\bigcap_{i=1}^{n} kill(M_i)|}{|T|}$$

The value of fragility lies between 0 and 1. When fragility takes the value 0 there is no test case that can kill this mutant, indicating that this mutant is potentially an equivalent mutant. As the value of fragility increases from 0 to 1, the mutant is assessed to be weaker, until the value equals 1, when the mutant is so weak that it can be killed by any of the test cases. In the following, $M_{1...n}$ is used to denote a higher order mutant consisting of the first order mutants F_1 to F_n. The fitness function for a higher order mutant is defined as follows.

Definition 8 (Fitness Function)**.**

$$fitness(M_{1...n}) = \frac{fragility(\{M_{1...n}\})}{fragility(\{F_1, ..., F_n\})}$$

That is, the fitness of a higher order mutant is defined to be the ratio of the fragility of this higher order mutant to the fragility of its constituent first order mutants. From the definition, if the fitness is greater than 1, then the higher order mutant is weaker than its constituent first order mutants (i.e. it is useless). As the fitness decreases from 1 to 0, the higher order mutant becomes gradually stronger than its constituent first order mutants. However, when the fitness value reaches 0, the higher order mutant is considered as a potential equivalent higher order mutant, and so all such zero-valued higher order mutants are discarded. All of the following algorithms use this fitness function to evaluate the fitness of higher order mutants.

Greedy Algorithms

A greedy algorithm (GR) is an algorithm that makes local optimised choices at each stage with the hope of achieving a near global optimum [52]. The general procedure of a greedy algorithm starts by solving the first sub-problem by selecting the solution with maximum current fitness. The action is then repeated to solve the rest of the problem. Therefore, greedy algorithms can only be used to solve problems that can be divided into sub-problems and can only provide a single solution. In order to apply the greedy approach to finding more than one subsuming higher order mutant, several optimised changes have been made. An initial first order mutant is chosen at random as a starting point. Subsequently, the normal greedy algorithm process is performed to incrementally augment with additional first order mutants. An archive operation is used to store the subsuming higher order mutants found. The overall algorithm is iterated with repeated randomised initial position, much like a random-restart hill climbing algorithm. The pseudo-code is shown in Algorithm 2.

Hill Climbing Algorithm

A hill climbing algorithm (HC) is a local search algorithm in which the next solution considered will depend on both the fitness value and distance to the current solution. The process starts from a random initial solution. By comparing the fitness of the current solution with that of its neighbour solution, the most fit solution becomes

```
Input   : Fitness evaluation limit: limit
Output: Mutation vector: hom_list
set counter = 0
while counter < limit do
    set available_foms = getAllFOMs()
    set hom = generateRandFOM()
    set best_hom = hom
    foreach FOM m in available_foms do
        temp_hom = combine(hom,m)
        if fitness(temp_hom) < fitness(best_hom) then
            best_hom = temp_hom
            best_fom = $m$
        end
        RemoveFOM(available_foms, best_fom)
        archvie(hom_list,best_hom)
    end
    counter ++
end
return hom_list
```

Algorithm 2: Optimised Greedy Algorithm

the new current solution, until fitness cannot be improved further.

Here, a neighbourhood operator considers two types of moves to generate a new mutant, location change neighbour move and mutation change neighbour move. The first type of move keeps all of the original mutation, but tries to apply them to different locations. The second type of move keeps all the positions of the original mutation, but tries to explore different types of mutation at thesis locations. Figure 5.4 illustrates the concept of these two neighbourhood moves. The proposed optimised algorithm is based on a random-restart hill climbing algorithm, which chooses a random starting solution for each run. The pseudo-code is shown in Algorithm 3.

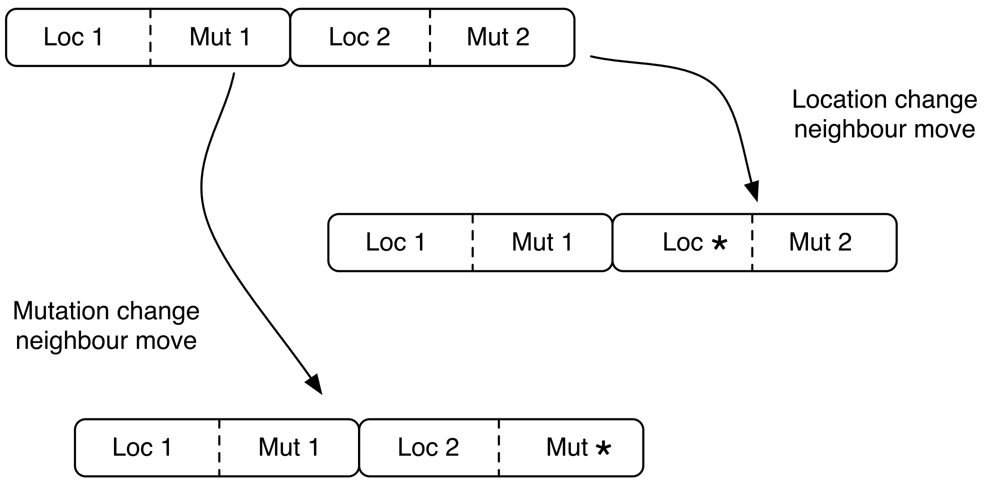
Loc 1
Mut 1
Loc 2
Mut 2
Location change neighbour move
Loc 1
Mut 1
Loc *
Mut 2
Mutation change neighbour move
Loc 1
Mut 1
Loc 2
Mut *

Figure 5.4: Two types of neighbour moves for hill climbing algorithm.

Genetic Algorithm

A genetic algorithm (GA) is a population-based evolutionary algorithm that simulates the process of natural genetic selection according to the Darwinian theory of biological evolution [180]. In a genetic algorithm, every possible solution within the solution domain is represented as a chromosome, and crossover and mutation operations are repeatedly performed on chromosomes to produce new solutions until a

```
Input  : Running Time Limit: limit
Input  : Local move limit: local_move_limit
Output: Mutation vector: hom_list
set counter = 0
set hom = generateRandFOM()
set no_improve = 0
while counter < limit do
    temp_hom = getRandomNeighbour(hom)
    if fitness(temp_hom) < fitness(hom) then
        hom = temp_hom
    else
        no_improve = no_improve + 1
        if no_improve == local_move_limit then
            archvie(hom_list, hom)
            hom = generateRandFOM()
        end
    end
    counter ++
end
return hom_list
```

Algorithm 3: Optimised Hill Climbing Algorithm

member of the evolving population is deemed to represent a suitably 'good' solution. In the proposed genetic algorithm, each gene within the chromosome represents the position and possible type of mutation (see Section 5.3). The algorithm uses a single point crossover operator to generate new mutants, as shown in Figure 5.5. The mutation operators include three possible changes: add a mutant, delete a mutant, and change mutation type, as shown in Figure 5.6. In additional to crossover and mutation operators, an archive operator is used to store the subsuming higher order mutants found. The pseudo-code is shown in Algorithm 4.

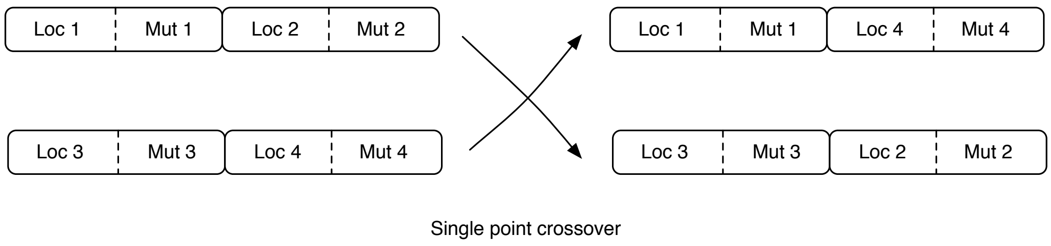

Figure 5.5: Single point crossover for the genetic algorithm

5.4 Empirical Study

This section describes the set of experiments designed to explore properties of subsuming higher order mutants. Section 5.4 discusses the research questions that the study will address. Section 5.4 describes the subject programs used in this study. Section 5.4 explains the experimental procedure.

Research Questions

This section sets out the research questions addressed in the empirical study and for which the next section provides answers.

```
Input  : Fitness evaluation limit: limit
Output: Mutation vector: hom_list
set counter = 0
while len(population) < $pop\_size$ do
    set $m$ = generateRandHOM()
    evaluateFitness($m$)
    append(population,$m$)
end
while counter < limit do
    createMatingPool(population)
    crossover(population)
    mutate(population)
    evaluateFitness(population)
    archvie(hom_list, population)
    counter ++
end
return hom_list
```

Algorithm 4: Optimised Genetic Algorithm

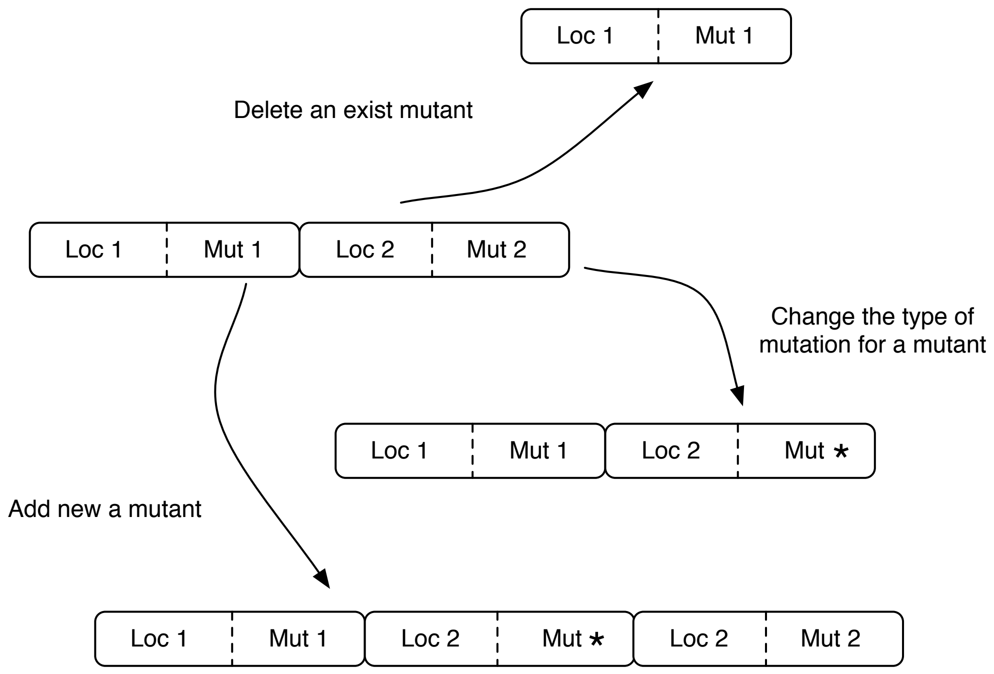

Figure 5.6: Three types of mutation operators for the genetic algorithm

RQ1: How numerous are subsuming higher order mutants?
The main goal of this chapter is to introduce and study subsuming higher order mutants. Therefore, the natural first research question is how prevalent subsuming higher order mutants are.

RQ2: What proportion of subsuming higher order mutants have entirely decoupled constituent first order mutants?
Since the work presented in this chapter seeks ways in which first order mutants combine to make valuable higher order mutants that partially mask each other, it is also interesting to ascertain what proportion of higher order mutants contain first order mutants whose killing sets do not overlap. Where there is no intersection between the killing sets of the first order mutants, these first order mutants cannot combine in ways that partially mask one another. This issue is explored in RQ2 by repeated sampling of higher order mutants to determine the relative proportion (for

each program studied) of the higher order mutants that consist of entirely decoupled first order mutants. This allows the approximation of the overall proportion of 'decoupled higher order mutants' and the degree to which this proportion varies per program studied.

RQ3: What proportion of subsuming higher order mutants are strongly subsuming? As introduced in Section 5.1, strongly subsuming higher order mutants are the most valuable higher order mutants that can be applied in higher order mutation testing, directly. RQ3 studies the proportion of the strongly subsuming higher order mutants found in all subsuming higher order mutants.

RQ4: What do strongly subsuming higher order mutants look like?
In order to understand higher order mutants better, several of those strongly subsuming higher order mutants found by the proposed algorithms were examined in order to find the simplest example of a strongly subsuming higher order mutant. This illustrates the way in which faults may partially mask one another so that the set of test cases that kill all first order mutants is a subset of the intersection of the test sets that kill the first order mutants. Surprisingly, the proposed algorithms even managed to find such an example in the familiar `Triangle` program; it was initially believed that such a program would have been too small and simple to allow for the construction of a strongly subsuming higher order mutant.

RQ5: Which algorithms perform best at finding subsuming higher order mutants? Three algorithms for finding subsuming higher order mutants are introduced. RQ5 explores how these algorithms perform in relative terms.

Subject Programs

The experiments use ten benchmark C programs with branch adequate test sets from the Software-artifact Infrastructure Repository (SIR) [78], as described in the first two columns of Table 5.1. The `Triangle` program is a small program that is used to determine the type of triangle from the length of its sides. This version is the one used by Offutt in the coupling effect study [197].

The seven programs `Replace`, `TCAS`, `Schedule2`, `Schedule`, `Totinfo`, `Printtokens` and `Printtokens2` are collectively known as the 'Siemens Suite', which is widely used as a benchmark for software testing techniques. `TCAS` is a program used to avoid an aircraft collision. `Schedule2` and `Schedule` are programs that prioritise schedulers. `Totinfo` is a program that computes statistics from input data. `Printtokens` and `Printtokens2` are lexical analysers. `Replace` performs pattern matching and substitution.

Besides the `Triangle` program and the Siemens Suite, there are two other 'real world' programs: `Gzip` and `Space`. `Gzip` is a widely used compression program and `Space` is an interpreter for an array definition language.

There are two reasons for choosing these programs. Firstly, previous studies of higher order mutants are limited to programs on a small scale. In contrast, this study is able to consider programs from 50 to 6,000 lines of code. Secondly, in order to measure the fitness of higher order mutants precisely, the higher order mutants have to be executed against a set of reasonably high quality test cases. The SIR provides branch adequate test sets, thereby achieving this aim.

Experimental Procedure

Algorithm 5 sets out the steps involved in the experimental procedure. Trivial mutants are first filtered out from the set of all first order mutants to remove from consideration those killed by all test cases and those killed by none of the test cases. The remaining 'non-trivial mutants' are used to generate subsuming higher order mutants. The set of all possible subsuming higher order mutants is unfeasibly large,

```
for each subject program do
    generate all possible first order mutants
    filter out the first order mutants that are killed by all test cases
    filter out the first order mutants that are killed by non-test cases
    store rest first order mutants as the set: ‘non-trivial first order mutants’
    apply search based optimisation to generate subsuming higher order
    mutants from non-trivial first order mutants
    for 100 trials, from all non-trivial first order mutants, allow the algorithm
    to consider 10,000 higher order mutants from which its optimisation
    procedure finds as many subsuming higher order mutants as possible,
    guided by the fitness function do
        count the percentage of subsuming higher order mutants within the
        higher order mutants
        count the percentage of strongly subsuming higher order mutants
        within the subsuming higher order mutants
        count the percentage of non-intersection higher order mutants within
        the subsuming higher order mutants
    end
end
```

Algorithm 5: Experimental procedure

Table 5.1: Selected Subject Programs: Scale shows the size of the programs expressed in Lines of Code (LoC), No. of FOMs is a count of all FOMs generated for each program. The 'possible equivalent' FOMs are those not killed by any test cases, while the 'dumb FOMs' are those killed by all test cases.

Programs	Scale	No. of Test Cases	No. of FOMs	No. of possible Equivalent FOMs	No. of Dumb FOMs
`Triangle`	50 LoC	60	601	62	35
`TCAS`	150 LoC	1,608	744	239	60
`Schedule2`	350 LoC	2,710	1,603	238	970
`Schedule`	400 LoC	2,650	1,213	155	810
`Totinfo`	500 LoC	1,052	2,316	245	1,100
`Replace`	550 LoC	5,542	4,195	486	3,133
`Printtokens2`	600 LoC	4,054	1,714	345	569
`Printtokens`	750 LoC	4,071	1,237	557	210
`Gzip`	5,500 LoC	228	12,027	1,124	5,770
`Space`	6,000 LoC	13,498	68,843	26,401	5,378

but search-based optimisation is used to locate them so that size is not a problem. Rather, it provides a rich set from which to seek useful higher order mutants.

However, in order to answer questions about relative proportions, a kind of sampling approach is required to approximate the answers. Each 'sample' is a set of subsuming higher order mutants, constructed by one of the search-based optimisation algorithms from an allowed 'budget of consideration' of 10,000 higher order mutants. The particular algorithm used is a parameter to the procedure.

In answering RQ5, results are reported for the performance of four algorithms: a greedy algorithm, a hill climb algorithm, a genetic algorithm and (for baseline comparison) a random search algorithm. However, to answer the questions regarding the proportions of higher order mutants that have the properties described in RQ1–RQ3, only the genetic algorithm is used, since this was found to locate the most subsuming higher order mutants. From each set of 10,000 higher order mutants, the proportion of higher order mutants constructed by the genetic algorithm that were subsuming is computed. From the set of subsuming higher order mutants, both the proportion that were strongly subsuming and the proportion that is entirely decoupled are computed. These proportions are reported as percentages. In order to factor out possible effects from sampling, thereby arriving at a more accurate approximation to the true proportion, the entire process is repeated for 100 trials per program and per program averages are reported over the 100 trials.

5.5 Results and Analysis

This section will present the answer to each research question in turn, indicating how the results answer each question.

Answer to RQ1

RQ1 is designed to investigate the number of the subsuming higher order mutants that exist. To begin the analysis, the second and third columns of Table 5.2 present the overall results for the sum of percentage subsuming higher order mutants found in each subject programs by the genetic algorithm with 10,000 fitness evaluations, repeated for 100 trials (giving 1,000,000 fitness evaluations in total per program). From the smallest `Triangle` program (50 LoC) to the largest `Space` program (6,000 Loc), there exist subsuming higher order mutants.

Table 5.2: This table shows the proportion of higher order mutants which are subsuming higher order mutants (SHOMs) and the proportion of these subsuming higher order mutants that are strongly subsuming higher order mutants (SSHOMs) and non–intersecting higher order mutants (NIHOMs).

Program	Non-trivial FOMs	% of SHOMs	% of SSHOMs	% of NIHOMs
`Triangle`	504	81.6%	0.24%	80.4%
`TCAS`	445	89.5%	0.11%	97.2%
`Schedule2`	395	57.5%	0.27%	77.2%
`Schedule`	248	75.1%	0.39%	64.1%
`Totinfo`	971	58.2%	0.24%	49.3%
`Replace`	576	67.5%	0.31%	62.2%
`Printtokens2`	800	47.0%	0.10%	31.2%
`Printtokens`	470	52.2%	0.01%	50.9%
`Gzip`	5,133	71.4%	0.08%	43.3%
`Space`	39,064	77.5%	0.21%	32.4%

Answer to RQ2

RQ2 is designed to investigate the proportion of entirely decoupled subsuming higher order mutants. Figure 5.7 shows the percentage of higher order mutants that are constructed of non-intersecting first order mutants on the vertical axis against the order of the higher order mutant concerned on the horizontal axis. For example, a point at (x, y) means that $y\%$ of all higher order mutants of order x are non–intersecting. That is, their first order mutants are entirely decoupled; there is no pairwise intersection between any of the sets of test cases that kill each of the constituent first order mutants.

As the figure shows, there is a tendency for decoupling to increase as the order of the higher order mutant increases (for all programs studied). However, the figure reveals that this property is very different for different programs. For example, for the program `totinfo`, only approximately 5% of the 9th order mutants are composed of entirely decoupled first order mutants, whereas approximately 90% of the 9th order mutants for `triangle` and `TCAS` consist of entirely decoupled first order mutants.

The rightmost column of Table 5.2 shows the proportion of all higher order mutants constructed that were found to be composed of entirely decoupled first order mutants. Notice that the number of non-intersecting higher order mutants appears to decrease as the number of first order mutants increases. A Spearman Rank Correlation test was performed to investigate statistically this observation more rigorously. The test showed a strong rank correlation between the proportion of subsuming higher order mutants that are non-intersecting higher order mutants and the number of first order mutants and also between the proportion of subsuming higher order mutants that are non-intersecting higher order mutants and the number of non-trivial first order mutants.

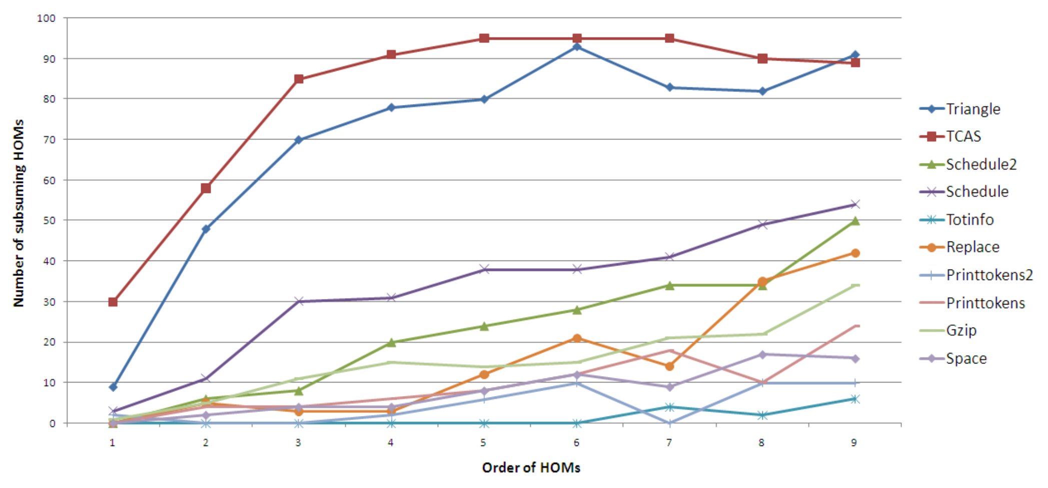

Figure 5.7: Overall Type Distribution

Answer to RQ3

RQ3 is designed to investigate the proportion of strongly subsuming higher order mutants. Of all subsuming higher order mutants found, between approximately 0.01% and 0.4% were found to be of the highly valuable, strongly subsuming type. This is a very small overall proportion, but there is a very large number of subsuming higher order mutants because the proportion of all higher order mutants that are subsuming higher order mutants is very large, and so the *number* of strongly subsuming higher order mutants is high.

Answer to RQ4

RQ4 focused on the study of strongly subsuming higher order mutants. To answer RQ4, a case study of a strongly subsuming higher order mutant that was found by a genetic algorithm in the `Triangle` program is presented. The `Triangle` is a small C program (50 LoC) that has been studied for at least 30 years [68]. The program takes the length of the sides of a potential triangle and outputs whether the triangle is a valid shape, and if so, whether it is equilateral, isosceles or scalene. Program 65 details the source of the `Triangle` program. There are two main factors that determine the type of the triangle. The first is the side length constraint; the sum of the length of any two sides must be greater than the length of the third. The second is captured by the variable *trian*, whose value is used to specify the type of triangle. For instance, if a triangle's *trian* value equals 0, and the side lengths satisfy the side length constraint, it is a 'valid scalene' triangle.

Program 7 shows two first order mutants and the subsuming higher order mutant constructed from them, which was found by the proposed optimised genetic algorithm in the `Triangle` program. The way in which the higher order mutant strongly subsumes the two first order mutants is subtle and involves an interplay between

```
Program: Triangle
Input  : Three sides a, b, c
Output : Type of triangle

int trian
if ($a <= 0$ || $b <= 0$ || $c <= 0$) then
    return INVALID
trian = 0
if ($a == b$) then trian = trian + 1
if ($a == c$) then trian = trian + 2
if ($b == c$) then trian = trian + 3
if (trian == 0) then
    if ($a + b < c$ || $a + c < b$ || $b + c < a$) then
        return INVALID
    else return SCALENE
if (trian > 3) then return EQUILATERAL
if (trian == 1 && $a + b > c$) then
    return ISOSCELES
else if (trian == 2 && $a + c > b$) then
    return ISOSCELES
else if (trian == 3 && $b + c > a$) then
    return ISOSCELES
return INVALID
```

Program 6: The original `Triangle` program

```
Mutant : FOM_i
if (trian > 1 && a + b > c) then
    return ISOSCELES
else if (trian == 2 && a + c > b) then
    return ISOSCELES
else if (trian == 3 && b + c > a) then
    return ISOSCELES
return INVALID
```

```
Mutant : FOM_j
if (trian == 1 && a + b <= c) then
    return ISOSCELES
else if (trian == 2 && a + c > b) then
    return ISOSCELES
else if (trian == 3 && b + c > a) then
    return ISOSCELES
return INVALID
```

```
Mutant : HOM_ij
if (trian > 1 && a + b <= c) then
    return ISOSCELES
else if (trian == 2 && a + c > b) then
    return ISOSCELES
else if (trian == 3 && b + c > a) then
    return ISOSCELES
return INVALID
```

Program 7: The strongly subsuming higher order mutant and its two constituent first order mutants for the `Triangle` program. As this case study demonstrates, even from this trivially small program, extremely subtle strongly subsuming higher order mutants can be constructed. Table 5.3 depicts the corresponding killing test cases.

the validity and type-of-triangle tests in the original program. It is reasonable to postulate that it is just this sort of subtle interaction that leads to faults that may go unnoticed in less rigorous testing.

Table 5.3 summarises the reasons why this is an instance of strong subsumption. From the table, only three types of test cases are able to kill FOM_i while two types of test cases are able to kill FOM_j. However, careful inspection reveals that HOM_ij can only be killed by test cases of the form ($a == b$ && $a + b > c$). Test cases of this form also kill FOM_i and FOM_j. There is no other test case that is able to kill HOM_ij. Therefore, strongly subsuming HOM_ij can be used to replace both FOM_i and FOM_j in mutation testing.

Mutant	Test Case	Original Result	Mutant Result
M_1	$a == b$ && $a + b > c$	Isosceles	Invalid
	$a == c$ && $a + b > c$ && $a + c <= b$	Invalid	Isosceles
	$b == c$ && $a + b > a$ && $b + c <= a$	Invalid	Isosceles
M_2	$a == b$ && $a + b > c$	Isosceles	Invalid
	$a == b$ && $a + b <= c$	Invalid	Isosceles
M_{12}	$a == b$ && $a + b > c$	Isosceles	Invalid

Table 5.3: Killing Test Cases for the Triangle HOM and its FOMs

Answer to RQ5

RQ5 is designed to investigate the effectiveness of the proposed algorithms. The chart in Figure 5.8 presents the results of the comparison of the four algorithms, which answers RQ5. An oracle of all subsuming higher order mutants found is used to provide a reference against which each algorithm is assessed. The oracle contains the union of the resulting subsuming higher order mutants from each algorithm. The greater the percentage of this oracle an algorithm can find, the better the algorithm is deemed to perform. In Figure 5.8 the x-axis shows the four algorithms, and the y-axis shows the percentage of oracle higher order mutants found. The genetic algorithm performs best since it finds the highest percentage of oracle higher order mutants; this is likely because the subsuming higher order mutants are easier to generate from existing subsuming higher order mutants. In the genetic algorithm, this observation favours crossover, which is one of the genetic algorithm's distinguishing features.

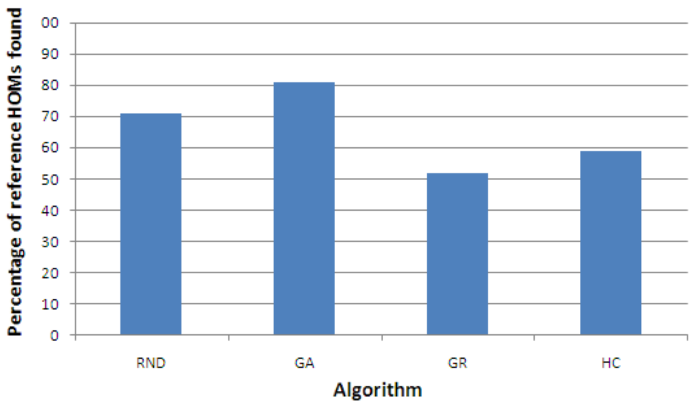

Figure 5.8: Algorithm comparison

Although the genetic algorithm found more of the subsuming higher order mutants, the hill climbing algorithm and the greedy algorithm also have their advantages. The hill climbing algorithm always finds the highest fitness higher order mutants because its subroutine repeatedly improves the fitness of higher order mutants, while the greedy algorithm finds the highest order first order mutants because it starts from a random first order mutant and tries to achieve as high an order as possible. Therefore, the results reveal that the genetic algorithm is the best performing algorithm, and the greedy algorithm and hill climbing algorithm can also be used to augment results and to search for extreme cases. The results also show that even the random search algorithm is able to find a large number of subsuming higher order mutants indicating that there are a large number of available subsuming higher order mutants that are relatively easy to find.

5.6 Discussion

5.6.1 Threats to Validity

This section considers the threats to validity of the experiments presented in this chapter. Although due to limitations of the experiments the following threats may affect some of the results (for example the distribution and classification of subsumed higher order mutants), it should be noted that they do not affect the proof of the existence of strongly subsuming higher order mutants found by the experiments.

The selection of mutation operators is the first threat. In order to reduce the computational cost, in the experiments of this chapter a subset of the 77 mutation operators for the C language [6] were selected to generate higher order mutants. However, the selected subset belongs to the five selective mutation operator categories suggested by Offutt [212, 203], so it is typical and also widely used by other researchers. This threat to validity will be overcome by future work which will investigate the relationship between higher order mutants and mutation operators.

The quality of the test sets is another potential threat. Since the fitness of higher order mutants is computed in terms of their fragility, low quality test sets may affect the results. Although the test sets provided by SIR achieve branch coverage [78], given a different test set as input, the experiment may lead to different results in terms of distribution and classification. To overcome this threat, plans for future work include the combination of higher order mutation testing with the co-evolutionary mutation testing approach of Adamopoulos et al. [5]. This will allow test sets to be co-evolved that are adequate to kill the co-evolving higher order mutant set.

The last threat is the existence of equivalent mutants. Although the problem of equivalent mutants has been studied by numerous researchers [124, 200, 208], there is no approach that can solve it in both an effective and a precise way. In order to avoid this problem, the fitness function for finding interesting higher order mutants is designed to filter out potential equivalent mutants. With a low quality test set, some of the 'stubborn decoupled' higher order mutants may be incorrectly treated as equivalent mutants. However, this would only reduce the number of higher order mutants found, so the results presented in this chapter can be considered to be a lower bound on the number of subsuming higher order mutants to be found.

5.6.2 Related Work

This chapter introduces the paradigm of higher order mutation testing. This is the first time that higher order mutation testing has been considered as a valid alternative to first order mutation testing, and, indeed, this author prefers the full precision of strong mutation testing. Weak and firm higher order mutation testing remain interesting and potentially important topics for future work.

The closest research area related to this work is the previous work on the coupling effect hypothesis. Although the coupling effect has been studied by many researchers [196, 197, 184, 37, 274, 275], these studies all focus on verifying or disproving the coupling effect, rather than finding subsuming higher order mutants, which can be thought of as special cases.

The experimental studies presented by Offutt [196, 197] show results that support Offutt's version of the mutation coupling effect. However, Offutt modifies Demillo et al.'s original statement of the coupling effect [68]:

> “ Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors [68]. ”

The original formulation appears to suggest that *all* higher order mutants are coupled, whereas Offutt [197] weakens this to suggest that a ‘large percentage’ are coupled:

> “ Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will detect a large percentage of the complex mutants [197]. ”

Some of the ‘subsuming higher order mutants’ presented here are drawn from the minority ‘de-coupled’ mutant set. Offutt’s experiments were based on three small Fortran77 programs (16-28 LoC). All of the second order and some of the third order mutants of these programs were generated by the mutation testing tool Mothra. The results suggested that the selected adequate test set which killed all the first order mutants killed over 99% of the second and third order mutants. This study implied that the mutation coupling effect is valid in the most general case, in agreement with the empirical study by Lipton and Sayward [163] and Morell [184].

The validity of the mutation coupling effect has also been considered in a theoretical study by Wah [274, 275]. A simple theoretical model, the q function model, considers a program to be a set of finite functions. By applying test sets of orders 1 and 2 to this model, the results indicated that the average survival ratio of high order mutants is $1/n$ and $1/n^2$ respectively, which is also similar to the estimated results of the empirical studies mentioned above. However, compared to a real world program, this model is too simplistic. In real programs, the data and control flow between functions are more complex and unpredictable.

This chapter proposed using strongly subsuming higher order mutants in mutation testing. This idea has been partly proven by Polo et al.'s work [227]. In their experiment, they focused on a specific order of higher order mutants, namely second order mutants. They proposed different algorithms to combine first order mutants to generate the second order ones. By applying the second order mutants, test effort was reduced by approximately 50%, without much loss of test effectiveness. However, Polo et al. did not use search based optimisation, and so they were limited to a small number of lower orders. Future work will consider the question of whether search can find arbitrary order HOMs that can reduce test effort.

In order to apply mutation testing to real world programs, strong mutation testing is adopted in the experiments of this chapter. In strong mutation testing, a mutant is killed if its final output is different from the original program. Therefore, each mutant is executed until it terminates or is killed. In order to reduce the running cost, previous work also considered weak mutation testing, first proposed by Howden in 1982 [129]. In weak mutation testing, mutants are evaluated immediately after execution of their mutation point. This is faster than strong mutation testing but with a loss of precision. There are also other approaches that lie between strong and weak mutation testing, known as firm mutation testing [184, 287].

Summary

This chapter focused on an investigation of higher order mutants and their relationship to first order mutants. It introduced the concept of subsuming higher order mutants; a higher order mutant that is more difficult to kill than its constituent first order mutants. In terms of fragility, the whole is greater than the sum of its parts. That is, the higher order mutant is greater than the collection of first order mutants from which it is constructed because it is less fragile. This chapter introduced a search-based approach to find these subsuming higher order mutants and presented an empirical study that compared a greedy algorithm, a genetic algorithm and a hill climbing algorithm.

The experimental results from ten programs indicate that there exist many subsuming higher order mutants in each program studied. The results also reveal that the genetic algorithm is the most efficient algorithm for finding those subsuming higher order mutants while the greedy algorithm and hill climbing algorithm can also be used to improve the quality of the results.

The chapter also introduced the concept of a strongly subsuming higher order mutant. A strongly subsuming higher order mutant is only killed by a subset of the intersection of the set of test cases that kill its constituent first order mutants. Therefore, a strongly subsuming higher order mutant is one that is so much more difficult to kill than the first order mutants from which it is constructed that one can replace all the first order mutants with the subsuming higher order mutant without any loss of test effectiveness.

The chapter showed that the search-based approach was able to find a number of these strongly subsuming higher order mutants in each of the ten programs studied. Although the proportion of all higher order mutants that are strongly subsuming higher order mutants is small, the size of the higher order mutant set grows exponentially, so the number of these valuable strongly subsuming higher order mutants is relatively high. This chapter illustrated the intricate interplay between faults that strongly subsuming higher order mutants exhibit by describing one of the higher order mutants found by the genetic algorithm and the test sets that kill it and its constituent first order mutants in detail. The next chapter will introduce a hybrid approach to generate test data to kill higher order mutants strongly.

Chapter 6

SHOM: Strong Mutation Based Test Data Generation

Higher order mutation testing has been the subject of much recent attention [24, 120, 170]. As well as its ability to model more complex masking faults [136], there is evidence to suggest that it may reduce mutation effort [227] and also the proportion of mutants that are equivalent [155, 223]. Comprehensive higher order mutation testing requires a test data generation approach for killing higher order mutants.

In test data generation approaches, if a test input distinguishes the behaviour of the original program from that of one of its mutants, then the test input is said to 'kill' the mutant. If the test input merely causes the state to change after the mutation point is executed, then the mutant is said to be 'weakly' killed. However, if the test input causes this state change to propagate to an output, where there is an observable failure, then the test input is said to 'strongly' kill the mutant. Strong mutation testing embodies a more demanding criterion for test adequacy than weak mutation testing so that, wherever possible, it is preferable to use test suites that are suitable for testing strong mutants [286]. By definition, a test that strongly kills a mutant must also weakly kill it, but not necessarily *vice versa*.

There has been much work on different techniques and tools for generating mutants, with over 250 publications on mutation testing in the literature. However, only ten of these publications (about 4% of the total) address the problem of automatically generating test data to kill mutants [139]. A summary of these publications is presented in Table 6.1. While mutation generation remains important, it is also clearly desirable to be able to use mutation testing to generate test inputs as well as to assess them.

Previous work on the generation of test data to kill mutants has used traditional structural-oriented test data generation techniques, for example, traditional symbolic execution [70, 164, 195, 201, 216], dynamic symbolic execution (DSE) [222, 225, 293] and search based software testing (SBST) [17, 102]. However, all of the existing techniques are designed to achieve only weak mutation adequacy and only for first order mutants. There is neither existing work on killing higher order mutants nor any work on generating test data that is adequate for strong mutations.

This chapter presents SHOM, a novel hybrid test data generation approach that draws on previous work from both DSE and SBST to achieve strong higher order mutation adequacy [1]. This chapter also presents evidence to support the claim that SHOM is efficient and effective for both first order and higher order mutations. Of course there remains the question of what a 'single syntactic change' is. There are many definitions of such sets of mutation operators in the literature [6, 203]. Since higher order mutations must be defined with reference to a set of first order mutation operators, for the purposes of this thesis it is important only to define the first order mutation. The contributions of the chapter can be summarised as follows:

[1]A first order mutant is a special case of a higher order mutant so that SHOM also achieves first order mutation adequacy.

Table 6.1: Mutation-based Test Data Generation. (‡) The work of Fraser and Zeller achieved (R)eachability and (I)nfection and also a constrained form of (P)ropagation, because it sought to maximize the mutant's effect on assertions, providing a form of propagation and also a way to maximise mutant impact.

Authors [Ref]	Year	(R)eaches, (I)nfects, (P)ropagates	Technique	Subject Language	Largest Subject	Average Mutation Score
DeMillo and Offutt [195, 70]	1991	R,I (Weak)	Static Domain Reduction	Fortran	55 Lines	98%
Offutt et al. [201, 216, 218]	1994	R,I (Weak)	Dynamic Domain Reduction	Fortran	100 Lines	98%
Liu et al. [164]	2006	R,I (Weak)	Dynamic Domain Reduction	C	21 Lines	95%
Zhang et al. [293]	2010	R,I (Weak)	DSE	C#	472 Lines	90%
Papadakis et al. [224]	2010	R,I (Weak)	DSE	C	500 Lines	63%
Ayari et al. [17]	2007	R (Weak)	SBST	Java	72 Lines	88%
Papadakis et al. [225]	2010	R (Weak)	DSE	Java	100 Loc	90%
Fraser and Zeller [102]	2010	R,I (Firm)‡	SBST	Java	412 Classes	72%

1. A hybrid test data generation approach for strongly killing both first order and higher order mutants is introduced. This approach is evaluated on seventeen subject programs, including seven real world programs (four from two different closed-source industrial systems and three for which source code is publicly available). For backward compatibility with comparable recent studies (that use C) and older studies (that use Fortran), C versions of ten of the smaller programs studied in this previous work are also included [195, 70, 201, 216]. However, the work presented in this thesis also includes programs an order of magnitude larger than any of these smaller programs.

2. The results of an empirical evaluation of the efficiency and effectiveness of SHOM for strong first order mutation adequacy are reported. The results show that SHOM can kill up to 38% of the first order mutants that remain unkilled using reachability and infection, which in turn kills up to 36% of the mutants that remain unkilled using reachability alone.

3. The results of a further empirical study of the efficiency and effectiveness of SHOM for strong second order mutation adequacy are reported. The results show that SHOM can kill up to 48% of the second order mutants that remain unkilled using reachability and infection, which in turn kills up to 41% of the mutants that remain unkilled using reachability alone.

The rest of this chapter is organised as follows. Section 6.1 introduces a novel hybrid DSE/SBST approach, while Section 6.2 briefly describes implementation details. Section 6.3 describes the experimental method, the results of which are discussed in Section 6.4.

6.1 Strongly Killing Higher Order Mutants Using DSE and SBST

First, it is necessary to define a mutant and a higher order mutant and what it means to kill them, then the approach taken to generating test data using a combination of DSE and SBST to strongly kill higher order mutants will be explained.

Definition 9 (First Order Mutant)**.** *A first order mutant p' of a program p is constructed by making a single syntactic change to p. A transformation that produces a mutant from the original program is called a 'mutation operator'.*

Of course there remains the question of what constitutes a 'single syntactic change'. There are many definitions of such sets of mutation operators in the literature [6, 203]. As previously stated, for the purposes of this thesis it is only important to define the first order mutation so that a higher order mutation can be defined in terms of it, since a higher order mutation can only be formally defined with respect to a set of first order mutation operators.

Definition 10 (Higher Order Mutant)**.** *Given a set of first order mutation operators M, if a mutant p' is created from a program p by the application of k operators from M, then p' is said to be a k^{th} order mutant of p.*

Definition 10 of the higher order mutation subsumes Definition 9 of the first order mutation because setting $k = 1$ in Definition 10 yields Definition 9. In general, care will be required to ensure that all of the k mutation operations create a distinct syntactic change when applied to p. It may also be necessary to define the order of application of the k mutation operators since different application orders may produce a different overall syntactic effect. However, these topics will be left for future studies on higher order mutation.

Higher order mutants are generally easier to kill than first order mutants. However, there also exists a small set of higher order mutants that is more difficult to kill than the first order mutants from which they are constructed. This type of higher order mutant is known as a subsuming higher order mutant (SHOM), see Chapter 5 for more details. Figure 6.1 gives a simple illustrative example of a SHOM. Both mutant 1 and mutant 2 are so-called 'dumb' mutants (those which are very easy to kill). In this case, both are killed by *any and every* test case; the most dumb possible. However, the higher order mutants created by inserting both mutant 1 and mutant 2 together is far from dumb; it is much more difficult to kill them than either of its first order mutants. Essentially, in this kind of situation, fault masking can create subtle higher order bugs from unsubtle first order bugs.

The killing conditions required to (strongly) kill a first order mutant are well described in the literature: A test input needs to satisfy following three conditions: reachability, infection and propagation (RIP), each of which subsumes the preceding condition(s):

1. **Reachability**: The location of the mutant in the program must be executed by the test case; the mutant is said to have been 'reached'. Reaching all the mutants of a program can be achieved by any branch adequate test set; therefore, reachability is an instance of branch coverage, a research area that is widely studied in literature [10, 111, 123, 241].

2. **Infection**: Immediately after mutant execution, the states of the original program and the mutant must differ. It can be said that the mutant 'infects' the state. A test case that achieves infection for a mutant m is also said to 'weakly kill' the mutant m [70, 139, 185].

3. **Propagation**: The infected state must propagate to some point in the program at which it can be observed, such as an output statement. A test case

Figure 6.1: Illustrative example: two dumb first order mutants combine to make a more subtle second order mutant

```
inputs: a, x, y
       1  z = x;
       2  z = z + y;
       3  if(a > 0)
       4      return z;
       5  else
       6      return 2 * x + z;
```

mutant 1: changes line 1 to z = ++x

mutant 2: changes line 2 to z = z + - -y

higher order mutant: combines mutant1 and mutant2

tests	original	mutant 1	mutant 2	mutant 12
$a > 0$	$x + y$	$x + y + 1$	$x + y - 1$	$x + y$
$a \leq 0$	$3x + y$	$3x + y + 3$	$3x + y - 1$	$3x + y + 2$
	n/a	killed by all	killed by all	killed by half

that achieves propagation for a mutant m is also said to 'strongly kill' the mutant m [70, 139, 185].

6.1.1 Weakly Killing Mutants

DSE has proven to be an effective means of satisfying both the reachability and infection conditions [111, 241], and, as a result, there has been much work on DSE as a technique for achieving weak mutation adequacy [222, 225, 293]. However, it has not been adapted to handle strong mutations.

The work of this thesis uses DSE to generate weakly killing constraints and test data that satisfy them. When generating mutants, properties denoting reachability infection are collected for each mutant. The reachability property is captured by the set of critical predicate nodes that transitively control mutant reachability. This property is generated by traditional control dependence analysis. The second property is the infection constraint, which is determined by the specific type of mutant. This thesis will use the infection conditions proposed by DeMillo and Offutt [70].

6.1.2 Handling Higher Order Mutants

Previous work on DSE for first order mutation testing will be adopted and adapted so that it is able to handle higher order mutants in addition to first order mutants. A higher order mutant, m, of order n is a composition of n first order mutants. These n first order mutants will be termed the 'constituent' mutants of m. For each higher order mutant, there are two important cases to consider. Case 1: There exists a path that traverses all constituent first order mutants. Case 2: There does not exist such a path.

If Case 1 applies, then it is possible that the higher order mutant is a subsuming higher order mutant. A 'subsuming' higher order mutant is one that is more difficult to kill than any of its first order constituents due to fault masking among the constituent first order mutants [138]. In testing terms, it can be said that 'the sum of the collection of first order mutants is more demanding to test than the union of its parts'. However, if there does not exist a path that passes through all constituent first order mutants, then, by definition, they cannot all mask one another, and so the 'sum is merely the union of its parts' and is, therefore, easier to kill.

Of course, in Case 2 there could be a path that traverses some subset, S, of the constituent first order mutants, but this would mean that there would also be a lower order mutant composed of precisely the S constituent mutants. If one seeks to increase progressively the order of mutants considered, then such a case will already have been encountered. Therefore, here attention will focus on Cases 1 and 2, as defined above.

Suppose a higher order mutant that one seeks to weakly kill is constructed from a set of constituent first order mutants $f_1, \ldots, f_n$. If there is a path in the control flow graph of the program that passes through all the critical predicate nodes of $f_1, \ldots, f_n$, then the higher order mutant may be subsuming; this is Case 1. For these higher order mutants, the critical predicate nodes of the higher order mutant are defined as the union of the critical predicate nodes of the $f_1, \ldots, f_n$. By extension, the infection constraint of the higher order mutant is the conjunction of the infection constraints of $f_1, \ldots, f_n$.

If there is no such path (Case 2), then it is not possible to find a test case that executes all the constituent first order mutants that combine to make the higher order mutant. In this situation, the proposed approach treats the higher order mutant as merely a set of first order mutants; the higher order mutant is killed if any of the constituent first order mutants are killed.

Here, a different variant of the DSE algorithm to that previously used for mutation testing will be used [222, 225, 293]. The proposed reachability approach is the same as that of previously published work, and this is inherited from the standard DSE approach to branch coverage [111, 241]. However, in the proposed approach, infection constraints are handled differently due to the need to retain and extend the constraints for subsequent generation of strongly killing test cases.

Previous work uses a testability transformation to transform the traditional branch adequacy problem, which is handled well by DSE, into weak mutation adequacy. This is performed simply by replacing mutants with additional branches, whose predicates capture the infection constraint.

The approach proposed in this thesis does not transform the program. Rather, once a mutant is reached, the DSE variant continues to generate test data to satisfy the weak killing constraint of the mutant. This makes it possible to retain a mapping of mutants and the corresponding infection constraints so that the fitness of each individual mutant can be assessed when it subsequently comes to the task of propagating infections. The pseudo code of this DSE algorithm is shown in Algorithm 8.

```
Require: the set of critical predicate nodes $N$ reaching the mutant
Require: the $InfectionConstraint$ of the mutant
  For program P, randomly generate concrete test input $T$
  while within execution upper bound do
    execution path $p \leftarrow$ dynamic execution $(P, T)$
    symbolic expression $sc \leftarrow$ symbolic execution $(P, T)$
    if $p$ does not reach the mutant then
      current critical node $n \leftarrow$ get next critical node $(N, p)$
      $p \leftarrow$update constraints $(p, n)$
      $T \leftarrow$ constraint solver$(p, sc)$
    else
      break
    end if
  end while
  weak killing constraint $wkc \leftarrow InfectionContraint \wedge p$
  $T \leftarrow$ constraint solver$(wkc)$
  return $T$
```

Program 8: The dynamic symbolic execution algorithm

If the DSE approach fails to generate weakly adequate test data for a mutant, standard SBST approaches are used to seek to weakly kill it. This is because it is known [153] that DSE and SBST achieve coverage of distinct, but overlapping, sets of branches. For example, SBST is well adapted to test data generation in the presence of floating point computation.

Such distingct coverage motivated work on a hybrid DSE–SBST approach, now incorporated into the Pex tool [154]. However, for the experiments described in this thesis (reported in Section 6.3), this search-based weak killing feature of the SHOM implementation is switched off so that weak adequacy is achieved by DSE alone; this is because it is desirable to compare the additional effort required and effectiveness achieved in terms of strong adequacy compared to the DSE–only approaches to weak adequacy.

Having used DSE to generate weakly adequate test data, the proposed hybrid DSE–SBST approach uses SBST to search for test inputs that propagate infected data states to outputs, thereby transforming weak mutation into strong mutation. The next section describes the proposed SBST approach to strong higher order mutation testing, which lies at the heart of the proposed overall SHOM approach.

6.1.3 Strongly Killing Mutants

In order to strongly kill a mutant, its infection must be propagated to an output so that the fault is manifested as a failure. The propagation problem has previously been considered to be difficult because there may be infinitely many paths from the infection point to the point at which an output occurs. Therefore, the problem of propagation, *for each mutant*, can be reduced to the path coverage problem. Even if path coverage is approximated, this process must still be repeated for each mutant, resulting in a potentially prohibitive computational cost.

The proposed approach uses SBST to search for paths from the infection point to the output that are *more likely* to propagate the infection, based on heuristic assumptions regarding the differences in paths taken by the original and the mutant, which should be maximised using the search.

In this way, it is not necessary to try all paths from infection to output explicitly. Rather, paths that are more likely to propagate are *searched for*, guided by a fitness function that measures control flow differences between the original program and the mutant. First, a testability transformation is used to ensure that the program has only a single return point; the return of the procedure in which the mutant resides. This simple transformation is always possible because multiple return statements can be directed to a single 'gathered' return point.

One seeks to maximally disrupt the path taken by the mutant version of the program from the infection point to this unique return statement. This increases the likelihood that any output statement that can be executed after the infection point will be executed differently (or even not at all). This, in turn, increases the likelihood that the output of the mutant will be distinguishable from that of the original, thereby strongly killing the mutant.

It is desirable to favour tests that maximise disagreement on predicate choices made by the original program and mutant, thereby maximally disrupting the control flow path from the infection to the return. If a test makes the mutant follow a different path to the original after execution, then it is very likely to produce a different value at the return point, thereby strongly killing the mutant. Let $Branch(p, i, t)$ denote the branch taken by program p at predicate i on input t. Let $inf(m)$ denote the infection point of mutant m and let $ret(m)$ denote the return point of the procedure containing m. Let $pred(p, x, y)$ denote all critical predicates between point x and point y in program p.

The decision function d for program p and mutant m at predicate i on input t is defined as follows:

$$d(p,m,i,t) = \begin{cases} 1 & \text{if } Branch(p,i,t) = Branch(m,i,t) \\ 0 & \text{if } Branch(p,i,t) \neq Branch(m,i,t) \end{cases}$$

The fitness $f(p,m,t)$ of a test case t executed on a mutant m of an original program p is defined to maximise the average 'predicate disagreement' between m and p:

$$f(p,m,t) = \frac{\sum_{i \in Pred(m,inf(m),ret(m))} d(p,m,i,t)}{n}$$

Recent results [122] have demonstrated that 'random restart hill climbing' provides an effective and efficient way to generate test data using SBST. Motivated by this finding, the work of this thesis uses a random restart hill climbing algorithm to search for the test inputs that propagate the infection, as shown in Algorithm 9. However, the particular choice of SBST algorithm is a parameter to the proposed approach and a pluggable component to its implementation.

6.1.4 Preserving Weak Adequacy Using Constrained Search

The proposed representation and move operations are designed to *guarantee* that the previously obtained reachability and infection constraints are also satisfied by any candidate input considered during the SBST phase of the overall approach. To do this, an individual candidate solution to the problem of killing a mutant is represented as a conjunction of constraints. This conjunction starts off as the reachability and infection constraints, to which it is only ever possible subsequently to add additional conjuncts during the search process.

In order to express a potential move to a new test input in the search, an extra conjunct is added to the current constraint, representing the result in conjunctive normal form (CNF). In this way it is only ever possible to consider weakly killing test cases. The constraint solver is used to generate a candidate using the extended CNF consisting of the weakly killing constraint plus some candidate new constraint. The test input generated by the constraint solver is then evaluated for fitness, and, if it improves fitness, it becomes the new current solution in the hill climb.

This 'constrained search' approach to searching for test data is a novel aspect of the proposed mutant killing technique since it has not been used in any previous work on SBST, although it may be found in other applications, in more general work on SBST outside the domain of mutation testing. This approach enables the combination of constraint solving and SBST in a manner that preserves the value captured by the constraints, while extending it to achieve some additional aspiration using search.

6.2 SHOM Implementation

Figure 6.2 depicts the architecture of SHOM, the implementation of the proposed hybrid DSE–SBST approach to strong higher order mutation. To compute adequacy scores, the tool MiLu[136, 138], which was introduced in Chapter 4 Section 4.2, is used. MiLu is a higher order mutant generation and assessment tool that supports general purpose first and higher order mutant generation for C. The subset of the Agrawal et al.'s 77 C mutation operators [6] that fall into the widely studied 'selective' mutation operators, defined and studied first by Offutt el al. [203], are used here. In addition, a specific implementation of the DSE phase was used so that it was possible to extend it to include the subsequent SBST phase.

The CIL transformation system [192] was used to pre-process the program and its mutants for the DSE and SBST phases of the proposed implementation. However,

```
Require: A weak killing test $T$
Require: The weak killing constraint $wkc$
  if $T$ kills the mutant strongly then
    return $T$
  else
    while current evaluation ¡ max evaluation do
      $NeighboursTests \leftarrow$ neighbours($T$)
      for all $t$ in $NeighboursTests$ do
        if $t$ kills the mutant strongly then
          return $t$
        end if
        for all $t$ in $NeighboursTests$ do
          if fitness($t$) $> bestfitness$ then
            $BestTest \leftarrow t$
            $bestfitness \leftarrow$fitness($BestTest$)
          end if
        end for
        if $bestfitness \leq$ fitness($T$) then
          $T \leftarrow$ get a weak killing test $T$
        else
          $T \leftarrow BestTest$
        end if
      end for
    end while
  end if
```

Program 9: Our hill climbing algorithm this is merely a testability transformation that reduces constraint and path analysis effort. It does not alter the semantics of the program under test, nor does it affect the test adequacy criteria involved. As illustrated in Figure 6.2, the test data generated using the proposed approach are evaluated on the mutants generated by MiLu, not the transformed versions.

Three transformation steps are performed. First, the expressions denoting array indices and other memory access operators are simplified. In this step, additional temporary variables are introduced to hold intermediate values for complex memory expressions which involve more than one memory reference. After this step, the value of the simplified expression only contains a memory constructor. This simplifies the subsequent static analysis and dynamic symbolic execution by reducing the number of cases that have to be considered.

The standard transformations of CIL are used to simplify loop and switch statements, reducing all such control flow constructs to a simple canonical form, consisting of conditionals and branches. Once again, this leaves the semantics of the original unaltered but eases the subsequent downstream analyses.

Finally, each procedure is transformed to an equivalent single–entry/single–exit version so that it contains exactly one single return statement, to which the propagation of infection of all mutants that lie inside that procedure is sought. As explained in the previous section, this simplifies strong mutation testing since it means that the proposed SBST phase need only consider a single exit node. For this single exist node, SBST seeks inputs that cause execution to flow from the infection point along a maximally disrupted control flow path to the exit node.

CIL is also used to perform a control dependence analysis. This collects the critical predicate nodes for each mutant, used to form the reachability and infection conditions. The dependence analysis is also used to identify those predicates for which

SBST seeks to cause the mutant and original to disagree from infection to return.

The constraints for reachability and infection are represented in conjunctive normal form. SHOM uses the Yices constraint solver [87] to solve these constraints. Yices is a satisfiability modulo theories (SMT) constraint solver that uses a collection of advanced constraint solving techniques to find a satisfying assignment of values to variables in formulæ; here, it is used to satisfy the constraints for reachability and infection. Yices was chosen for two reasons:

1. Yices provides a C language application interface. This is necessary since it is not possible simply to use constraint solving as a 'black box' component. While this is possible for weak mutation killing techniques that simply use testability transformation to reformulate weak mutation as branch coverage, it is not possible for strong mutation. For strong mutation, it is necessary to control over exactly which constraints need to be satisfied at each part of the overall SHOM process.

2. Yices provides state-of-the-art constraint solving. It supports a wide range of constraints, including linear expressions, scalar types, recursive data types, tuples, records, arrays and bit-vectors, all of which can arise in the constraints found in programming languages. It won first place for several of the categories of the 2005, 2006 and 2007 SMT-COMP competitions organised as part of the computer aided verification (CAV) conference.

6.3 Empirical Study

The studies presented here consider first order and higher order mutations separately, because first order mutation has been the subject of previous work, while no other

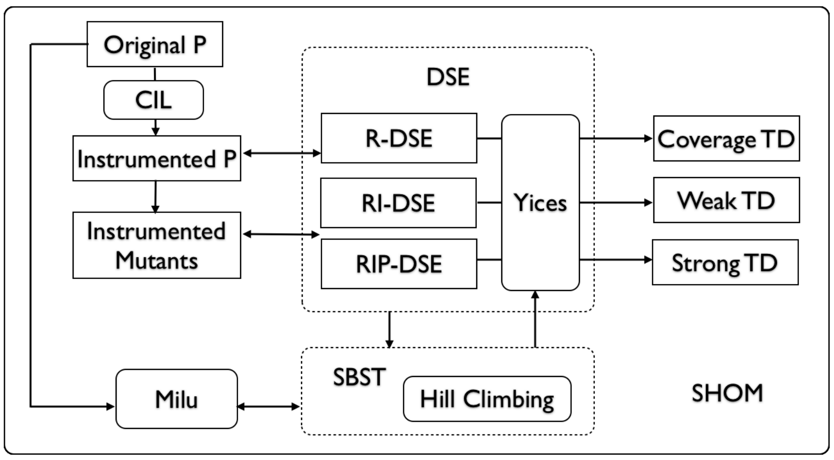

Figure 6.2: The SHOM Architecture. The DSE and SBST components were built from scratch; however, the DSE component delegates constraint solving to Yices [87]. It performs its analysis on transformed mutants, but all test data generated by SHOM are executed and evaluated by MiLu[136]. Transformation is performed by CIL [192].

previous studies have considered test data generation to kill higher order mutants. Only second order mutants and, for larger programs, only sets of randomized samples from the set of all possible second order mutants are considered. Sampling is required because of the infeasibility of considering all higher order mutants due to the very large increase in mutant numbers that occurs at higher orders.

Subject Programs Studied

SHOM was applied to the example subjects in Tables 6.2 and 6.3. The examples in Table 6.2 are non-trivial real world programs; four are modules from closed-source industrial production code. Two of them, DeFroster and F1, come from Daimler and are used in automotive control systems for a rear window defrosting system and an engine controller, respectively, and have been used in previous studies [117]. The other two, Hash and Buff, come from ABB and are used in robot controller systems.

It is not possible to provide the source code for these examples, because it is proprietary closed-source code from industrial partners with whom non-disclosure agreements have been signed. However, to support replication and more robust evaluation, three additional larger programs are also included, for which source code is readily available.

Table 6.2: The seven larger programs used in the experiments. The first five programs are industrial proprietary programs, while the final two are open source.

Program Name	Lines Of Code	Functions	Branches	First Order Mutants	Second Order Mutants
`DeFroster`	237	2	76	215	22,732
`F2`	511	1	42	212	22,113
`Hash`	1,011	12	76	465	107,211
`Space`	9,564	136	1,190	4,410	9,715,606
`Buff`	1,371	14	182	1,544	1,189,040
`GArray`	808	58	17	1,363	926,286
`Gzip`	7,933	97	1,717	10,182	51,816,418

The program Space is a widely studied interpreter for an array definition language used by the European Space Agency. It is not open source, but its code is available from the Software-artifact Infrastructure Repository (SIR) [77]. The other two programs, Gzip (v1.5) and GArray (v2.26), are both open source; Gzip is a widely used compression program and GArray is an array data structure used in the GNU Glib. All programs in this non-trivial subject set of examples are summarised in Table 6.2.

The second set of programs, summarised in Table 6.3, contains smaller laboratory programs that have been studied widely in the literature on mutation-based test data generation. This set of relatively small programs is included to provide backward compatibility with these previously studied examples. The set includes three programs taken from SIR that originated in the Siemens suite: Tcas is an aircraft anti-collision system, Schedule is a program that prioritises schedulers and Replace performs pattern matching and substitution.

Table 6.3: Ten smaller programs included for backward compatibility with previous studies.

Program Name	Lines Of Code	Functions	Branches	First Order Mutants	Second Order Mutants
Triangle	88	1	32	253	31,522
Bubble	35	1	6	80	3,032
Days	86	1	28	242	28,849
Find	88	1	22	201	19,791
Mid	43	1	10	65	1,970
GCD	43	1	6	73	2,526
MinMax	44	1	6	39	657
Tcas	166	8	66	223	24,496
Replace	595	23	176	714	253,585
Schedule	425	18	66	230	26,000

The remainder is a sample of some of the very small programs used in previous studies. No attempt is made to infer findings from the results obtained using these very small examples, but include them to facilitate replication. Triangle classifies the type of a triangle by the lengths of its three edges. Bubble is the standard bubble sort algorithm. Days calculates the number of days between two given days. Find locates and sorts the input array with a given index. Mid returns the middle value of three inputs. GCD is Euclid's greatest common divisor algorithm and MinMax returns the minimum and maximum values of an input array.

Mutant Generation

Some of the programs studied give no output. For example, many of the very small programs simply compute a single value as their result. For such programs it is necessary to clarify what is meant by ‘output’. If an overly pedantic and literal definition of output is taken, for example: ‘something that appears on an output device’, then all mutants of such programs would be equivalent because no mutation can make a change to a non-existent output. Therefore, ‘output’ is allowed to include the result of the computation returned to the environment (such as a return value or the result computed in a global variable).

For the larger programs where the code is not a support routine but an entire program, there is no such issue. These larger programs perform output to screen and/or files, and this is monitored and compared with the output of the original to determine whether the mutant is strongly killed.

Research Questions

The work of this chapter asks three research questions, which will now be defined, explaining how the experiments are designed to address them.

RQ1: How first-order-adequate is SHOM? To explore SHOM’s test effectiveness for strong first order mutation, SHOM is compared with RI-DSE. The improvement RI-DSE achieves over R-DSE is also reported. In both cases test sets for R-DSE and RI-DSE are generated, and the number of mutants each kills strongly is computed and compared to the number of mutants strongly killed by SHOM. This allows an evaluation of the degree to which a reached mutant is infected and propagates merely by reaching it using DSE and also the degree to which those infected mutants infected using DSE also already happen to propagate. All experiments are repeated ten times and averaged to cater for the stochastic nature of the search algorithm.

RQ2: How second-order-adequate is SHOM? The number of higher order mutants grows exponentially with the order k, presenting obvious experimental design challenges. For all of the ten programs detailed in Table 6.3, the total number of second order mutants is 392,458, which is manageable. However, for the real world programs detailed in Table 6.2 the total number of second order mutants is 63,799,406, which is unmanageable.

The quadratic increase in the number of second order mutants makes it impractical to consider all second order mutants. Therefore a sampling approach is adopted here. For programs with 0–4,999 second order mutants, 100% of the mutants are used. For programs with 5,000–49,999 mutants, 10% of the mutants are used. For programs with 50,000–499,999 mutants, 1% of the mutants are used. For programs with 500,000–4,999,999 mutants, 0.1% of the mutants are used. For programs with 5,000,000 or more mutants, 0.01% of the mutants are used. To avoid sampling bias, a random sample was taken from the set of all second order mutants. The sampling experiment was also repeated ten times and the average level of strong second order mutation adequacy achieved over all ten samples was computed.

To answer RQ2, SHOM was compared with RI-DSE, and RI-DSE was compared with R-DSE. However, there is no previous work on generating test data to kill second order mutants (either weakly or strongly). Therefore, to provide a baseline for comparison, the union of all test data generated for each of the two first order mutants from which the second order mutant is constructed was used as follows:

Suppose s is a second order mutant with constituent first order mutants f_1 and f_2. R-DSE is used to generate test data to kill f_1, creating a set of test data d_1. R-DSE is then used to generate test data to kill f_2, creating a second set of test data d_2.

The result of applying R-DSE to s is defined to be $d_1 \cup d_2$. Similarly, for RI-DSE, two test sets are generated, one for each of f_1 and f_2 and the test set produced by RI-DSE for s is defined to be the union of the two.

Using this approach, R-DSE and RI-DSE should be capable, in theory, of killing all those second order mutants that are coupled to their first order constituents in a way that killing either first order mutant kills the second order mutant. However, for second order mutants where fault masking may take place, a test set that kills both constituent first order mutants is not guaranteed to kill the second order mutant.

RQ3: How efficient is the SHOM data generation approach? Here, the efficiency of the SHOM approach is investigated. The efficiency is measured using both the elapsed time for test data generation and the number of fitness evaluations required. Again, the stochastic nature of the algorithm needs to be taken into account. In order to achieve stable and robust results, catering for variation due to the stochastic nature of the search process, each experiment was repeated ten times, and average values were calculated.

The time was recorded using the Linux `time` utility; this is the elapsed time, so it includes all time taken to generate mutants and test data, and to run test data on the program under test. As such, the timing information denotes a worst case upper bound on the total amount of time a tester would be expected to wait for test data to be produced by each technique. The experiments were performed on a MacBook Pro laptop with Intel Duo2 2.6 GHz CPU, 4GB memory in the Ubuntu 10.10 operating system.

6.4 Results and Analysis

Here, results from the research questions will be presented. Strong first order effectiveness is considered first, followed by strong second order effectiveness, and, finally, the efficiency of the SHOM implementation is considered.

Table 6.4: The results for SHOM's first order and second order adequacy.

Program	**R**-DSE %		**RI**-DSE %				SHOM %			
	Order		Order		Imp. on **R**		Order (Std.)		Imp. on **RI** (passed)	
	1st	2nd	1st	2nd	1st	2nd	1st	2nd	1st	2nd
Triangle	48	49	59	61	21	24	62 (1.6)	67 (3.8)	7 (10)	15 (10)
Bubble	76	77	76	77	0	0	76 (0.0)	77 (0.0)	0 (0)	0 (0)
Days	62	66	64	68	5	6	65 (0.5)	72 (2.6)	3 (10)	13 (10)
Find	64	59	69	60	14	2	69 (0.0)	61 (0.2)	0 (0)	3 (10)
Mid	65	62	66	73	3	29	82 (4.2)	82 (2.3)	47(10)	33 (10)
GCD	71	73	73	82	7	33	73 (0.0)	82 (0.0)	0 (0)	0 (0)
MinMax	75	64	77	75	8	31	77 (0.0)	76 (0.2)	0 (0)	4 (10)
Tcas	42	55	54	67	21	27	62 (2.1)	69 (9.1)	17 (10)	6 (10)
Replace	46	42	53	56	13	24	72 (2.2)	77 (11.5)	40 (10)	48 (10)
Schedule	55	57	57	62	4	12	69 (5.4)	70 (7.2)	28 (10)	21 (10)
Hash	51	54	56	61	10	15	63 (2.9)	64 (3.5)	16 (10)	8 (10)
Buff	63	64	71	73	22	25	82 (6.1)	85 (6.5)	38 (10)	44 (10)
GArray	64	68	77	81	36	41	82 (3.7)	86 (5.7)	22 (10)	26 (10)
DeFroster	53	55	62	63	19	18	66 (2.1)	68 (4.0)	11 (10)	14 (10)
F2	44	44	63	60	34	29	66 (1.2)	67 (8.4)	8 (10)	18 (10)
Space	30	32	46	51	23	28	52 (2.3)	57 (12.2)	11 (10)	12 (10)
Gzip	34	33	42	44	12	16	50 (1.5)	52 (13.4)	14 (10)	14 (10)
Average	55	56	62	65	15	21	69 (2.1)	71 (5.3)	15 (7.6)	16 (8.8)

SHOM’s First Order Adequacy

The results relating to RQ1 are summarised in Table 6.4. Columns labelled ‘R-DSE’ and ‘RI-DSE’ report, as percentages, the strong adequacy achieved by R-DSE and RI-DSE, respectively, for first and second order mutants. Of the four columns labelled RI-DSE, the third and fourth columns report the percentage of mutants left unkilled by R-DSE which are killed by RI-DSE. The four columns labelled ‘SHOM’ report the strong adequacy achieved by the proposed SHOM approach. The figures in the first two columns for SHOM report the average percentage of first and second order mutants killed over ten runs. The parenthetic numbers report the standard deviation. The figures in the third and fourth columns for SHOM report the average percentage improvement of SHOM over RI-DSE for first and second order mutations. In these two columns the parenthetic numbers report the number of runs, out of ten, for which SHOM outperformed RI-DSE.

Answer to RQ1: SHOM produces increases in strong first order mutation adequacy compared to RI-DSE, which, in turn, produces noticeable improvements on the strong adequacy achieved by R-DSE. For the smaller programs, the improvement in strong adequacy achieved by both RI-DSE and SHOM is less notable than it is for the larger programs.

This difference in behaviour is a further justification for including larger programs in the study of mutation-based test data generation. As already seen, using only very small program examples may skew the results due to the relatively trivial nature of the test data generation problem for these very small programs.

R-DSE and RI-DSE are entirely deterministic. SHOM builds on RI-DSE, but it is a randomised algorithm, so it can produce different values each time it is run. However, it is guaranteed to perform no worse than RI-DSE by construction, so the improvement it achieves (averaged over ten runs) together with standard deviation are reported.

These are the first results reported for strong mutation test data generation, so it is not possible to directly compare the current results with previous findings, such as those in Table 6.1. Perhaps the closest work to that presented here is that of Frazer and Zeller [102].

Although Fraser and Zeller report on test data generation for Java, while here test data generation for C is reported, their work is evaluated on two larger, non-trivial subjects, and it achieves a form of propagation (to assertions in the program rather than outputs). Fraser and Zeller reported an overall average first order mutation score of 72%, which lies between the weighted average strong first order mutation score for the whole programs (which was 59%) and that achieved for the libraries (which is 76%) in the work presented here.

There is a noticeable difference in the performance of all techniques for smaller and larger programs. For the smaller programs, from Table 6.3, R-DSE is able to strongly kill between 42% and 76% of the first order mutants. RI-DSE can improve on this, but for some of the programs the test problem is so trivial that even weakly adequate test sets achieve high levels of strong mutation adequacy.

For the larger programs the results are more interesting. The behaviour of all three techniques falls into two distinct categories, depending on whether the larger program is a whole program or merely a collection of library routines to be called by some other program. Of the larger programs, Hash, Buff and GArray are each collections of routines to be called from elsewhere; these three programs consist of libraries of subordinate routines and they have no `main` function. The other four of the larger programs, DeFroster, F2, Space and Gzip, are invoked, in their entirety, from their `main` function so that the whole program is tested.

It has been known for some time [16] that whole program analyses are more challenging than inter-procedural analyses that focus on a single procedure. This is also true for test data generation. For the libraries, it is merely needed to test each procedure in turn, thereby focusing the testing on a single procedure body rather than a whole program. The single procedure may call others in the library, so testing is still an inter-procedural activity, but it is not a 'whole program activity'.

This dichotomy between whole programs and libraries is borne out in the results. For the libraries, R-DSE is able to strongly kill between 51% and 64% of the first order mutants, whereas for the whole programs, it kills between 30% to 53% of the mutants. RI-DSE improves on this, killing between 10% and 36% of the remaining mutants for the libraries and between 12% and 34% of the remaining mutants for the whole programs.

SHOM further improves strong first order mutation scores in all of the larger programs studied. For the library programs, it manages to kill between 16% to 38% of the remaining mutants left unkilled by RI-DSE. For the whole programs, SHOM kills between 8% and 14% of the remaining mutants unkilled by RI-DSE.

Second Order Adequacy of SHOM

As can be seen from Table 6.4, on average, over all programs studied, all three techniques (R-DSE, RI-DSE and SHOM) are better at killing second order mutants than first order mutants. This is to be expected since second order mutants are, in general, coupled to first order mutants [139, 155]. These are the first results reported in the literature for automated test data generation to kill second order mutants, so they provide a baseline for future work.

The results also provide a baseline against which to evaluate SHOM. Over all programs studied, SHOM produces an improvement in strong second order adequacy over RI-DSE, which, in turn, produces an improvement over R-DSE. Once again, average performance for SHOM (over ten runs) and standard deviation are reported. Note that statistical tests such as the *t*-test or Mann Whitney test are not suitable here. The empirical evaluation is required to determine the size of this improvement, but SHOM is *guaranteed* to perform no worse than RI-DSE by construction.

For the larger programs detailed in Table 6.2, the dichotomy between libraries and whole programs is evident for second order mutation (as it is for first order mutation). For whole programs, the adequacy of all techniques is reduced compared to that for libraries. Over all larger programs, RI-DSE kills between 15% and 41% of the second order mutants left unkilled by R-DSE, while SHOM further increases this effectiveness, killing between 8% and 44% of the mutants left unkilled by RI-DSE

Efficiency of SHOM

Table 6.5 reports the number of fitness evaluations and time required to kill all mutants. In Table 6.5, the two columns labelled 'Time' report the average time taken by SHOM (in minutes). The next two columns, labelled 'Fitness' report the average number of thousand fitness evaluations required. The number of fitness evaluations required is not dissimilar to that required for branch coverage of similar sized programs using search-based techniques [10], so performance can be expected to be in line with previous work on SBST.

For the practicing software tester, the number of fitness evaluations, though machine-independent, will be of less interest; the results for the time taken to find an adequate test set are more important. The largest of the programs previously studied for mutation-based test data generation with C are the Siemens suite examples (Schedule, Replace and Tcas from Table 6.3). For these programs, it is possible to generate a weakly killing test set in a matter of seconds.

It is not possible to compare these findings with the previously reported results from the literature on mutant test data generation for C. This is because the relevant papers for which a comparison would be meaningful reported, in detail, upon only the effectiveness (mutation score) of the approaches, while they did not report the execution time details required for a comparison.

Of course, after two decades of Moore's Law, even if timing data were available for the older studies from the 1990s, a head-to-head time-based comparison would be grossly unfair to the achievements of previous work. For the more recently reported results (from 2010), even if timing data were available, differences in techniques, platforms and configurations would also make comparison problematic. Here, execution times, configuration and platform details are reported in order to support potential backward comparison in future work on strong and higher order mutation testing.

Mutation testing is generally regarded as a comparatively slow and expensive approach to testing. Despite this, it has endured as a research topic for more than three decades, perhaps because of results that demonstrate that it provides a particularly demanding test adequacy criterion and one that is attractively generic and flexible.

Given these historical perspectives, the time findings presented here are encouraging because they indicate that weak, strong and higher order mutation testing can all be used to generate test data within reasonable time-scales on a standard laptop. Generation of test data by hand (the only currently available alternative for either strong or higher order mutation) would take considerably longer, and, using human effort rather than machine effort would be (perhaps prohibitively) more expensive.

Table 6.5: The results for SHOM efficiency experiments.

Program	Time		Fitness	
	Order		Order	
	1st	2nd	1st	2nd
Triangle	13	102	3	17
Bubble	22	141	0.2	8
Days	14	114	5	13
Find	28	191	2	9
Mid	6	48	0.3	4
GCD	12	88	0.2	5
MinMax	22	84	0.1	3
Tcas	200	272	8	18
Schedule	110	202	4	15
Hash	81	128	5	8
Buff	152	176	11	7
GArray	95	131	2	3
DeFroster	102	272	2	11
F2	122	321	2	14
Space	1,423	884	43	18
Gzip	2,762	1,794	92	64
Average	307	301	11	13

Threats to Validity

The experiments presented here attempt to compare the proposed hybrid DSE test data generation approach with the traditional branch-based and mutation-based DSE-based approaches in terms of test effectiveness. Although the experiment was designed to be as fair as possible, it also faces a number of threats to the validity. There are two main threats, one relating to the mutant generation process and another relating to the test data generation process.

When generating mutants, the results can be influenced by the mutation operator used. To reduce the computation cost, Offutt's five sufficient mutation operators set was used to generate both first and higher order mutants. These operators have been used widely in many mutation studies [139]. However, for different types of subject programs, the generated mutants might be different in terms of the number and the type of distributions. To reduce the effect of this threat, the experiment was conducted on a variety of open-source programs, which represent many application domains.

Another possible threat related to mutants is that only second order mutants were studied in the higher order experiment. Generating all possible higher order mutants for a typical source unit is impossible, therefore only second order mutants were focused on here. Second order mutants are very good examples of higher order mutants; they not only have similar test effectiveness as first order mutants, having, thus, been suggested as a replacement to the first order mutations to reduce the running cost of mutation testing [223, 227], but they also contain some interesting subsuming cases which are more subtle than the first order mutants they are constructed from [136, 138]. In future experiments this threat can be addressed by searching the strongly subsuming mutants based on the their fragility. There are also some threats in the test data generation process. These threats mainly come from the limitation of the CIL library and the Yieas constraints solver.

Summary

In this chapter, a hybrid DSE and SBST approach to generate strongly adequate test data to kill first and higher order mutants was introduced. The approach was implemented in a tool called SHOM. Two previously published approaches were also implemented, based on reachability alone and reachability together with infection, and these implementations were used to evaluate the proposed approach in 17 example programs. The results show that SHOM is able to achieve higher levels of strong mutation coverage than either previously published approach for first order mutants. For second order mutants there is no previous work on test data generation, so the presented second order test sets were compared with those composed from the union of the corresponding first order sets. Once again, SHOM was found to outperform approaches based on either reachability alone or reachability and infection.

Chapter 7

Conclusions and Future Work

It is widely believed that higher order mutation testing is too computationally expensive to be practical and, as a result, work in the field of mutation testing has focused largely on first order mutants. This thesis has shown that higher order mutation testing can be practical when implemented as a search process that seeks fit mutants (both first and higher order) from the space of all possible mutants.

The fitness function can be tailored to the program under test and the specific goals of testing, thereby reducing the number of mutants required (compared to the traditional enumerative approach) and simultaneously increasing the quality and fitness for purpose of the selected mutants. The fitness function is able to take account of fault histories, known problems and likely pitfalls and is thereby able to simulate relevant potential faults that may have gone unnoticed in preceding testing efforts.

In this way the search based approach is able not only to generate smaller sets of more fit mutants, but also to target more realistic sets of mutants. It may even prove possible to use appropriately defined fitness functions to guide the search away from likely equivalent mutants, thereby reducing the impact of the equivalent mutant problem.

7.1 Summary of Achievements

The overall aim of this thesis was to make higher order mutation testing applicable and practical using a search process that seeks fit mutants (both first and higher order) from the space of all possible mutants. The detailed aims and objectives of this thesis were as follows:

1. To investigate higher order mutants from a fault interaction perspective.
2. To apply search-based optimisation approaches to locate very fit mutants (both first and higher order) within the search space of all possible mutants and to investigate empirically the higher order mutants found by the algorithms.
3. To extend the current state-of-the-art mutant-based test data generation techniques to handle higher order mutants and to evaluate this extended test data generation approach on both first order mutants and higher order mutant.

Higher order mutant classification

A fault-based higher order mutant classification was introduced in Chapter 4. Based on different types of fault interactions, this approach classifies higher order mutants into four categories: expected, worsening, fault masking and fault shifting. The chapter proposed a theoretical model for second order mutants and produced a classification tree for all second order mutants. In order to investigate practically the class of higher order mutants, the chapter also presents MiLu, a C mutation testing tool that is able to handle both first and higher order mutants. In an empirical study, all second order mutants were enumerated and third to fifth order mutants were sampled for six subject programs. In total, more than two million higher order mutants were generated with 9.2 billion test executions. The results show that 30% of higher order mutant faults fell within the fault mask and the fault shift categories; a potentially interesting result for future work on software testing.

Search-based higher order mutation

Subsuming and strongly subsuming higher order mutants were introduced in Chapter 5. These form subsets of fault masking and fault shifting higher order mutants which can be used in higher order mutation testing. A subsuming higher order mutant is a higher order mutant that is more difficult to kill than its constituent first order mutants. This chapter introduced a search-based approach to find these subsuming higher order mutants and presented an empirical study that compared a greedy algorithm, a genetic algorithm and a hill climbing algorithm. A strongly subsuming higher order mutant is only killed by a subset of the intersection of the set of test cases that kill its constituent first order mutants. Therefore, a strongly subsuming higher order mutant is one that is so much more difficult to kill than the first order mutants from which it is constructed that one can replace all the first order mutants with the subsuming higher order mutant without any loss of test effectiveness.

The results from ten test programs indicate that there exist many subsuming higher order mutants in each of the programs studied. It is revealed that the genetic algorithm is the most efficient algorithm for finding those subsuming higher order mutants, while the greedy algorithm and hill climbing algorithm can be used to improve the quality of the results. The results also show that the search-based approach was able to find a number of strongly subsuming higher order mutants in each of the ten programs studied. Although the proportion of all higher order mutants that are strongly subsuming higher order is small, the size of the higher order mutant set grows exponentially, so the number of these valuable strongly subsuming higher order mutants is relatively large. This chapter illustrated the intricate interplay between faults exhibited by strongly subsuming higher order mutants by describing in detail one of the higher order mutants found by the genetic algorithm and the test sets that kill it and its constituent first order mutants.

Mutation-based test data generation

Chapter 6 introduced a hybrid mutation testing approach whereby the DSE and SBST approaches are combined to generate strongly adequate test data to kill first and higher order mutants. The approach was implemented in the SHOM research tool. Two previously published approaches were also implemented, based on reachability alone and reachability together with infection, as a means to evaluate the proposed approach in 17 example programs. The results show that, for first order mutants, SHOM is able to achieve improved strong mutation scores than either of the previously published approaches. There is no previous work on test data generation for second order mutants, so the presented second order test sets were compared with those composed from the union of the corresponding first order sets. Once again, SHOM was found to outperform approaches based on either reachability alone or reachability and infection.

7.2 Summary of Future Work

Since the research presented in this thesis was published, there has been increasing interest in the topic of higher order mutation testing [24, 120, 223, 170, 149]. Significantly, there is evidence now to suggest that higher order mutants may reduce mutation effort [227] and also the proportion of mutants that are equivalent [223].

Genetic programing has also been used to generate interesting higher order mutants [155]. Recently higher order mutation has been applied to concurrent programs [170] as well as to detect equivalent mutants [149]. However, much more remains that can be done.

Applying fault models in higher order mutant testing

There is often fault data available for systems that are, or have been, under development over a substantial period of time. For systems developed in a certain domain or by a certain team of developers there may also be fault information available regarding the domain or team. In such situations a fault model is, in effect, developed; rather than simply constructing all possible faults, it is possible to focus on the faults characterised by the fault model. Future work will include, by using higher order mutant testing, to seek combinations of faults that may have gone undetected due to partial masking. By definition, a subsuming higher order mutant is one in which the first order constituent mutants partly mask one another so that the higher order mutant so-constructed is more difficult to kill than its constituent first order mutants.

The search based approach proposed in this thesis is well adapted to the presence of a fault model; it can be used to search for faults that are not only exemplars of the fault model, but also higher order mutants which denote subtle combinations of known likely faults. It is planned that in the future the search based approach will also be used to seek out near neighbours of known faults, using the fault model as a guide. In this way the search based approach can relax constraints so that the fault model is not used literally. Rather, it is treated as a guide to the kind of faults that may occur.

Co-evolving higher order mutants and test data

Co-evolution is an approach to evolutionary optimisation whereby two or more candidate populations evolve together, with the fitness of one population being determined by the fitness of the other [51]. In this way, the two populations evolve simultaneously. This can be a cooperative process, simulating symbiotic behaviour in natural evolution, or it can be competitive, simulating the familiar predator/prey model of co-evolutionary adaption and advancement.

For mutation testing it has been argued [5] that the predator/prey model of competitive co-evolution can be used to develop sets of hard-to-kill higher order mutants and, simultaneously, sets of very good quality test cases that are adapted to reveal subtle and hard-to-detect faults. In this approach the two populations are the population of candidate higher order mutants and the population of candidate test cases. The fitness for the higher order mutants is measured in terms of their ability to evade the test cases (how many test cases fail to kill them). The fitness of the test cases is measured in terms of their ability to kill the mutants.

A low fitness can be given to mutants that evade all test cases. These may be equivalent mutants. Of course, these mutants may also merely be stubborn so that the presented test cases are insufficient to reveal them. Such stubborn (nearly equivalent) mutants are precisely the kind of mutants that it is desirable to find. However, evolution is a mercifully robust process and the genes of such stubborn mutants will be scattered throughout the mutant population. If mutants which initially appeared to be equivalent are, in fact, merely stubborn, then it is likely that they will be rediscovered at a later stage of the evolution because they remain distributed through the gene pool. As ever, this means that maintenance of population diversity will be important for this form of co-evolution to succeed.

The argument for mutation testing, developed over the thirty years of its history, may seem circular. That is, mutants are 'good' if they avoid being killed by test cases, but it is difficult to ascertain the quality of the test cases; test cases are deemed to be 'good' if they kill all, or at least many, mutants. The co-evolutionary approach turns this uncomfortable circularity from a problem into an advantage.

Therefore future work will co-evolve sets of strongly subsuming higher order mutants with the test cases that are able to kill them with the goal of generating a set of very subtle faults and a set of test data that is sufficient to reveal them. That is, the apparently circular nature of mutation testing makes it an ideal candidate for a co-evolutionary approach. The aim is to make this a virtuous circle of co-evolutionary improvement.

Appendix A

Subject Programs used in the Literature of Mutation Testing

Table A.1: Programs used in Empirical Studies

Name	Size	Description	First Use	No. of Uses
Triangle	30 Loc	Return the type of a triangle	1978	25
Find	30 Loc	Patition the input array by order using input index	1988	22
Bubble	10 Loc	Bubble sort algorithm	1988	18
MID	15 Loc	Return the mid value of three integers	1989	16
Calendar/Days	30 Loc	Compute number of days between input days	1988	15
Euclid	10 Loc	Euclide's algorithm to find the greatest common divisor of two intergers	1991	15
Quad	10 Loc	Find the root of a quadratic equation	1991	14
Insert	15 Loc	Insert sort algorithm	1991	13
Warshall	10 Loc	Calculates the ttransitive closure of Boolean matrix.	1991	12
Pat	20 Loc	Decide if a pattern is in a subject	1991	10

Continued on next page

Table A.1 – continued from previous page

Name	Size	Description	First Use	No. of Uses
SPACE	6000 Loc	European Space Agency program	1997	9
Bsearch	20 Loc	Binary search on an interger array	1992	6
Totinfo	350 Loc	Information measure	1998	6
Schedule1	300 Loc	Priority scheduler	1998	6
Schedule2	300 Loc	Priority scheduler	1998	6
TCAS	140 Loc	Altitude separation	1998	6
Printtok1	400 Loc	Lexical analyzer	1998	6
Printtok2	480 Loc	Lexical analyzer	1998	6
Replace	510 Loc	Pattern replacement	1998	6
Max	5 Loc	Return the greater from the inputs	1978	4
STRMAT	20 Loc	Search String based on input pattern	1993	4
TEXTFMT	30 Loc	Text formating program	1993	4
Banker	40 Loc	Deadlock avoid algorithm	1994	4
Cal	160 Loc	Print a calendar for a specified year or month	1994	4
Checkeq	90 Loc	Report missing or unbalanced delimiters and .EQ / .EN pairs	1994	4
Comm	145 Loc	Select or reject lines common to two sorted files	1994	4
Look	135 Loc	Find words in the system dictionary or lines in a sorted list	1994	4
Uniq	85 Loc	Report or remove adjacent duplicate lines	1994	4
Gcd	55 Loc	Compute greatest common divisor of an array	1988	3
Sort	20 Loc	Sort algorithm foran array	1988	3
Binom	6 Func	Solves binomial equation	1994	3
Col	275 Loc	Filter reverse paper motions from nroff output for display on a terminal	1994	3
Sort(Linux)	842 Loc	Sort and merge files	1994	3
Spline	289 Loc	Interpolate smooth curve based on given data	1994	3

Continued on next page

Table A.1 – continued from previous page

Name	Size	Description	First Use	No. of Uses
Tr	100 Loc	Translate characters	1994	3
Ant	21,000 Loc	A build tool from Apache	2002	3
Determinant	60 Loc	Matrix manipulation programs based on LU decomposition	1994	2
Matinv	30 Loc	Matrix manipulation programs based on LU decomposition	1994	2
Transpose	80 Loc	Transpose routine of a sparse-matrix package	1994	2
Deadlock	50 Loc	Check for deadlock	1994	2
Stats	4 Func	Not reported	1994	2
Twenty-four	2 Func	Not reported	1994	2
Conversions	8 Func	Not reported	1994	2
Operators	4 Func	Not reported	1994	2
Crypt	120 Loc	Encrypt and decrypt a file using a user supplied password	1994	2
Bisect	20 Loc	Not reported	1996	2
NewTon	15 Loc	Not reported	1996	2
MRCS	Not reported	Mars Robot Communication System	2004	2
Xml-Security	143 Class	Implements security XML	2005	2
Jmeter	389 Class	A Java desktop application designed to load test functional behavior and measure performance	2005	2
JTopas	50 Class	A java library used for parsing text data	2005	2
ATM	5500 Loc	The ATM component are ValidatePin	2005	2
Tetris	Not reported	AspectJ benchmark	2006	2
Max_index	15 Loc	Find the max value in the input array	1988	1
NASA's planetary lander control software	Not reported	NASA's planetary lander control software	1992	1

Continued on next page

Table A.1 – continued from previous page

Name	Size	Description	First Use	No. of Uses
QCK	Not reported	Non-recurisive interger quicksort	1992	1
Gold Version G	2000 Loc	A battle simulation software	1992	1
Count	10 Loc	Not reported	1994	1
Dead	10 Loc	Not reported	1994	1
TCAS	Not reported	Air craft avoid colision system	1994	1
STU	15 Func	A part of a nuclear reactor safety shutdown system that periodically scans the position of the reactor's control rods.	1996	1
DIV/MOD	Not reported	Not reported	1996	1
EBC	10 Loc	Not reported	1996	1
Search	14 Nod	Not reported	1997	1
Secant	9 Nod	Not reported	1997	1
State chart of Citizen watch	Not reported	State chart of Citizen watch	1999	1
Queue	Not reported	ADS class library	1999	1
Dequeue	Not reported	ADS class library, double-ended queue	1999	1
PriorityQueue	Not reported	ADS class library, priority queue	1999	1
Areasg	50 Loc	Calculates the areas of the segments formed by a rectangle inscribed in a circle	1999	1
Minv	44 Loc	Computes the inverse ofthe square N by N matrix A	1999	1
Rpcalc	55 Loc	Calculates the value of a reverse polish expression using a stack	1999	1
Seqstr	70 Loc	Locate sequences of integers within an input array and copies them to an output array	1999	1
Streql	45 Loc	Compares two strings after replacing consecutive white space characters with asingle space	1999	1

Continued on next page

Table A.1 – continued from previous page

Name	Size	Description	First Use	No. of Uses
Tretrv	55 Loc	Performs an in-order traversal of a binary tree of integers to produce a sequence of integers	1999	1
Alternating-bit protocol	Not reported	Estelle specification Alternating-bit protocol	2000	1
Append	15 Loc	A component of a text editor	2001	1
Archive	15 Loc	A component of a text editor	2001	1
Change	15 Loc	A component of a text editor	2001	1
Ckglob	25 Loc	A component of a text editor	2001	1
Cmp	15 Loc	A component of a text editor	2001	1
Command	70 Loc	A component of a text editor	2001	1
Compare	20 Loc	A component of a text editor	2001	1
Compress	15 Loc	A component of a text editor	2001	1
Dodash	15 Loc	A component of a text editor	2001	1
Edit	25 Loc	A component of a text editor	2001	1
Entab	20 Loc	A component of a text editor	2001	1
Expand	15 Loc	A component of a text editor	2001	1
Getcmd	30 Loc	A component of a text editor	2001	1
Getdef	30 Loc	A component of a text editor	2001	1
Getfn	10 Loc	A component of a text editor	2001	1
Getfns	25 Loc	A component of a text editor	2001	1
Getlist	20 Loc	A component of a text editor	2001	1
Getnum	20 Loc	A component of a text editor	2001	1
Getone	25 Loc	A component of a text editor	2001	1
Gtext	15 Loc	A component of a text editor	2001	1
Makepat	30 Loc	A component of a text editor	2001	1
Omatch	35 Loc	A component of a text editor	2001	1
Optpat	15 Loc	A component of a text editor	2001	1
Spread	20 Loc	A component of a text editor	2001	1
Subst	35 Loc	A component of a text editor	2001	1

Continued on next page

Table A.1 – continued from previous page

Name	Size	Description	First Use	No. of Uses
Translit	35 Loc	A component of a text editor	2001	1
Unrotate	30 Loc	A component of a text editor	2001	1
LogServiceProvider	230 Loc	An abstract class which is extended by classes providing logging services.	2001	1
Print Writer Log Service Provider	85 Loc	Used for writing textual log messages to a print stream (for example, to the console)	2001	1
Logger	170 Loc	Provides the central control for the PSK logging service such as registering multiple log service providers to be operative concurrently	2001	1
LogMessage	150 Loc	A Message format to be logged by the logging service	2001	1
LogException	55 Loc	Base exception class for exceptions thrown by the logger and log service providers	2001	1
Junit	1,500 Loc	A unit testing framework	2002	1
GraphPath	150 Loc	Finds the shortest path and distance between specified nodes in a directed graph	2002	1
Paint	330 Loc	Calculates the amount of paint needed to paint a hous	2002	1
MazeGame	1,600 Loc	A game that involves finding a rescuing a hostage in a maze	2002	1
Specification of electrionic purse		Specification of electrionic purse	2003	1
Parking Garage system	12 Class	Java	2004	1
Video shop manager	17 Class	Java	2004	1
EJB Trading	Not reported	An EJB trading Component	2004	1

Continued on next page

Table A.1 – continued from previous page

Name	Size	Description	First Use	No. of Uses
RSDIMU	Not reported	The application was part of the navigation system in an aircraft or spacecraft	2005	1
Roots	Not reported	Determines whether a quadratic equation has real roots or not	2005	1
Calculate	Not reported	Calculates sum, product and average of the inputs	2005	1
BAMean	Not reported	Calculates mean of the input and both averages of numbers below and above mean	2005	1
SCMSA	Not reported	Application defined by the Web Services Interoperability Organization	2005	1
BOOK	250 Loc	An application between the diagnosis accuracy and the DBB sizes	2006	1
VirtualMeeting	1500 Loc	A server that simulates business meetings over network	2006	1
Nunit	20,000 Loc	A .NET unit test application	2006	1
Nhibernate	100,000 Loc	Library for object-relational mapping dedicated for .NET	2006	1
Nant	80, 000 Loc	.Net build tool	2006	1
System.XML	100,000 Loc	The Mono class libraries	2006	1
Assign_value	Not reported	A safety-critical software component of the DARTs	2006	1
Vending Machine	50L Loc	A vending maching example	2006	1
Sudoku	3360 Loc	A puzzle board game	2006	1
Polynomial Solver	450 Loc	A Polynomial solver	2006	1
MinMax	10 Loc	Return the maximum and minimum elements of an interger array	2006	1
Field	65 Loc	org.apache.bcel.classfile	2006	1
BranchHandle	80 Loc	org.apache.bcel.generic	2006	1

Continued on next page

Table A.1 – continued from previous page

Name	Size	Description	First Use	No. of Uses
String Representation	190 Loc	org.apache.bcel.verifier.statics	2006	1
Pass2Verifier	1000 Loc	org.apache.bcel.verifier.statics	2006	1
ConstantPoolGen	405 Loc	org.apache.bcel.generic	2006	1
LocalVariable	145 Loc	org.apache.bcel.classfile	2006	1
ClassPath	250 Loc	org.apache.bcel.until	2006	1
IntructionList	560 Loc	org.apache.bcel.generic	2006	1
JavaClass	465 Loc	org.apache.bcel.classfile	2006	1
CodeExceptionGen	120 Loc	org.apache.bcel.generic	2006	1
LocalVariables	95 Loc	org.apache.bcel.structurals	2006	1
NextDate	70 Loc	Determines the date of the next input day	2007	1
TicketsOrderSim	75 Loc	A simulation program in which agents sell airline tickets	2007	1
LinkedList	300 Loc	A program that has two threads adding elements to a shared linked list	2007	1
BufWriter	213 Loc	A simulation program that contains a number of threads that write to a buffer and one thread that reads from the buffer	2007	1
AccountProgram	145 Loc	A banking simulation program where threads are responsible for managing accounts	2007	1
Finance	5500 Loc	A reuses interfaces provided by an open source Java library MoneyJar.jar	2007	1
iTrust	2630 Loc	A web-based healthcare application	2007	1
Bean	Not reported	AspectJ benchmark suites	2008	1
NullCheck	Not reported	AspectJ benchmark suites	2008	1
Cona-sim	Not reported	AspectJ benchmark suites	2008	1
Spring.NET	100,000 Loc	An environment for programs execution	2008	1
Castle.DynamicProxy	6,600 Loc	A library for implementation of the Proxy design pattern	2008	1

Continued on next page

Table A.1 – continued from previous page

Name	Size	Description	First Use	No. of Uses
Castle.Core	6,200 Loc	Comprises the basic classes used in Castle projects	2008	1
Castle.ActiveRecord	21,000 Loc	Implements the ActiveRecord design pattern	2008	1
Adapdev	68,000 Loc	Extends the standard library of the .NET environment	2008	1
Ncover	4,300 Loc	A tool for the quality analysis of the source code in .NET programs	2008	1
CruiseControl	31,300 Loc	A server supporting a continuous integration of .NET programs	2008	1
Pprotection	220 Loc	Password Protection controls a reserved area	2008	1
Hhorse MP3	170 Loc	Manages MP3 audio files	2008	1
PHPP.Protect	1,300 Loc	Protects files	2008	1
AmyQ	200 Loc	Control a FAQ System	2008	1
EasyPassword	490 Loc	Manages password	2008	1
Show Pictures	1140 Loc	A mini Web portal	2008	1
Administrator	1400 Loc	Controls and administers reserved area	2008	1
Cmail	720 Loc	Sends email	2008	1
Workflow	7500 Loc	Manages a workflow system	2008	1

Bibliography

[1] Milu website. http://www.cs.ucl.ac.uk/staff/Y.Jia/Milu/.

[2] R. Abraham and M. Erwig. Mutation Operators for Spreadsheets. *IEEE Transactions on Software Engineering*, 35(1):94–108, January-February 2009.

[3] A. T. Acree. *On Mutation.* Phd thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.

[4] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation Analysis. Technique Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, Georgia, 1979.

[5] K. Adamopoulos, M. Harman, and R. M. Hierons. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'04)*, volume 3103 of *LNCS*, pages 1338–1349, Seattle, Washington, USA, 26th-30th, June 2004. Springer.

[6] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technique Report SERC-TR-41-P, Purdue University, West Lafayette, Indiana, March 1989.

[7] B. K. Aichernig. Mutation Testing in the Refinement Calculus. *Formal Aspects of Computing*, 15(2-3):280–295, November 2003.

[8] B. K. Aichernig and C. C. Delgado. From Faults Via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE'06)*, volume 3922 of *LNCS*, pages 324–338, Vienna, Austria, 27-28 March 2006. Springer.

[9] R. T. Alexander, J. M. Bieman, S. Ghosh, and B. Ji. Mutation of Java Objects. In *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, pages 341–351, Annapolis, Maryland, 12-15 November 2002. IEEE Computer Society.

[10] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A Systematic Review of the Application and Empirical Investigation of Search-Based Test-Case Generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[11] P. Ammann and J. Offutt. *Introducation to Software Testing.* Cambridge University Press, 2008.

[12] P. Anbalagan and T. Xie. Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 3, Raleigh, North Carolina, November 2006. IEEE Computer Society.

[13] P. Anbalagan and T. Xie. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs. In *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE'08)*, pages 239–248, Redmond, Washingto, 11-14 November 2008. IEEE Computer Society.

[14] J. H. Andrews, L. C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 402 – 411, St Louis, Missouri, 15-21 May 2005.

[15] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, August 2006.

[16] D. C. Atkinson and W. G. Griswold. The Design of Whole-Program Analysis Tools. In *International Conference on Software Engineering (ICSE '96)*, pages 16–27, 1996.

[17] K. Ayari, S. Bouktif, and G. Antoniol. Automatic Mutation Test Input Data Generation via Ant Colony. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)*, pages 1074–1081, London, England, 7-11 July 2007.

[18] J. S. Baekken and R. T. Alexander. A Candidate Fault Model for AspectJ Pointcuts. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 169–178, Raleigh, North Carolina, 7-10 November 2006. IEEE Computer Society.

[19] D. Baldwin and F. G. Sayward. Heuristics for Determining Equivalence of Program Mutations. Research Report 276, Yale University, New Haven, Connecticut, 1979.

[20] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability*, 11(2):113–136, May 2001.

[21] S. S. Batth, E. R. Vieira, A. R. Cavalli, and M. U. Uyar. Specification of Timed EFSM Fault Models in SDL. In *Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'07)*, volume 4574 of *LNCS*, pages 50–65, Tallinn, Estonia, 26-29 June 2007. Springer.

[22] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[23] F. Belli, C. J. Budnik, and W. E. Wong. Basic Operations for Generating Behavioral Mutants. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 9, Raleigh, North Carolina, 2006. IEEE Computer Society.

[24] F. Belli, N. Güler, A. Hollmann, G. Suna, and E. Yõldõz. Model-Based Higher-Order Mutation Analysis. In *Advances in Software Engineering*, volume 117 of *Communications in Computer and Information Science*, pages 164–173. Springer Berlin Heidelberg, 2010.

[25] J. Bieman, S. Ghosh, and R. T. Alexander. A Technique for Mutation of Java Objects. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, page 337, San Diego, California, 26-29 November 2001.

[26] P. E. Black, V. Okun, and Y. Yesha. Mutation of Model Checker Specifications for Test Generation and Evaluation. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 14–20, San Jose, California, 6-7 October 2001. published in book form, as *Mutation Testing for the New Century*.

[27] B. Bogacki and B. Walter. Evaluation of Test Code Quality with Aspect-Oriented Mutations. In *Proceedings of the 7th International Conference on*

eXtreme Programming and Agile Processes in Software Engineering (XP'06), volume 4044 of *LNCS*, pages 202–204, 2006, Oulu, 17-22 June 2006.

[28] B. Bogacki and B. Walter. Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing. In *Software Engineering Techniques: Design for Quality*, volume 227 of *IFIP*, pages 273–282, 2007.

[29] N. Bombieri, F. Fummi, and G. Pravadelli. A Mutation Model for the SystemC TLM2.0 Communication Interfaces. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*, pages 396–401, Munich, Germany, 10-14 March 2008.

[30] L. Bottaci. A genetic algorithm fitness function for mutation testing. In *Proceedings of the 8th Wrokshop on Software Engineering using Metaheuristic INovative Algorithms (SEMINAL'01)*, pages 3–7, 2001.

[31] J. H. Bowser. Reference Manual for Ada Mutant Operators. Technique Report GIT-SERC-88/02, Georiga Institute of Technology, Atlanta, Georgia, 1988.

[32] J. S. Bradbury, J. R. Cordy, and J. Dingel. ExMAn: A Generic and Customizable Framework for Experimental Mutation Analysis. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, pages 57–62, Raleigh, North Carolina, November 2006. IEEE Computer Society.

[33] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation Operators for Concurrent Java (J2SE 5.0). In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, pages 83–92, Raleigh, North Carolina, November 2006. IEEE Computer Society.

[34] J. S. Bradbury, J. R. Cordy, and J. Dingel. Comparative Assessment of Testing and Model Checking Using Program Mutation. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 210–222, Windsor, UK,

2007. IEEE Computer Society. published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*.

[35] P. Brady. MutateMe. http://github.com/padraic/mutateme/tree/master, 2007.

[36] Brinklow. Gestation periods in the Pere David's Deer (Elaphurus davidianus): evidence for embryonic diapause or delayed development. *Reproduction, Fertility and Development*, 5:567–575, 1993.

[37] T. A. Budd. *Mutation Analysis of Program Test Data.* Phd thesis, Yale University, New Haven, Connecticut, 1980.

[38] T. A. Budd and D. Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18(1):31–45, March 1982.

[39] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. The Design of a Prototype Mutation System for Program Testing. In *Proceedings of the AFIPS National Computer Conference*, volume 74, pages 623–627, Anaheim, New Jersey, 5-8 June 1978. ACM.

[40] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'80)*, pages 220–233, Las Vegas, Nevada, 28-30 January 1980.

[41] T. A. Budd and A. S. Gopal. Program Testing by Specification Mutation. *Computer Languages*, 10(1):63–73, 1985.

[42] T. A. Budd, R. Hess, and F. G. Sayward. EXPER Implementor's Guide. Technique report, Yale University, New Haven, Connecticut, 1980.

[43] T. A. Budd and F. G. Sayward. Users Guide to the Pilot Mutation System. Technique Report 114, Yale University, New Haven, Connecticut, 1977.

[44] R. H. Carver. Mutation-Based Testing of Concurrent Programs. In *Proceedings of the IEEE International Test Conference on Designing, Testing, and Diagnostics*, pages 845–853, Baltimore, Maryland, 17-21 October 1993.

[45] Cetress. Certitude. http://www.certess.com/product/, 2006.

[46] W. K. Chan, S. C. Cheung, and T. H. Tse. Fault-Based Testing of Database Application Programs with Conceptual Data Model. In *Proceedings of the 5th International Conference on Quality Software (QSIC'05)*, pages 187–196, Melbourne, Australia, 19 -20 September 2005.

[47] R. N. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, 2005.

[48] P. Chevalley. Applying Mutation Analysis for Object-oriented Programs Using a Reflective Approach. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC 01)*, page 267, Macau, China, 4-7 December 2001.

[49] P. Chevalley and P. Thévenod-Fosse. A Mutation Analysis Tool for Java Programs. *International Journal on Software Tools for Technology Transfer*, 5(1):90–103, November 2002.

[50] B. Choi and A. P. Mathur. High-performance Mutation Testing. *Journal of Systems and Software*, 20(2):135–152, February 1993.

[51] S. Y. Chong, P. Tino, and X. Yao. Measuring generalization performance in co-evolutionary learning. *IEEE Transactions on Evolutionary Computation*, 12(4):479–505, August 2008.

[52] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[53] Y. Crouzet, H. Waeselynck, B. Lussier, and D. Powell. The SESAME Experience: from Assembly Languages to Declarative Models. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 7, Raleigh, North Carolina, November 2006. IEEE Computer Society.

[54] M. Daran and P. Thévenod-Fosse. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. *ACM SIGSOFT Software Engineering Notes*, 21(3):158–177, May 1996.

[55] R. Delamare, B. Baudry, and Y. Le Traon. AjMutator: A Tool For The Mutation Analysis Of AspectJ Pointcut Descriptors. In *Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09)*, pages 200–204, Denver, Colorado, 1-4 April 2009. IEEE Computer Society. published with *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops.*

[56] M. E. Delamaro. *Proteum - A Mutation Analysis Based Testing Environmen.* Masters thesis, University of São Paulo, Sao Paulo, Brazil, 1993.

[57] M. E. Delamaro and J. C. Maldonado. Proteum-A Tool for the Assessment of Test Adequacy for C Programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS'96)*, pages 79–95, New Brunswick, New Jersey, July 1996.

[58] M. E. Delamaro and J. C. Maldonado. Interface Mutation: Assessing Testing Quality at Interprocedural Level. In *Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC'99)*, pages 78–86, Talca, Chile, 11-13 November 1999.

[59] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Integration Testing Using Interface Mutation. In *Proceedings of the seventh International Sympo-*

sium on Software Reliability Engineering (ISSRE '96), pages 112–121, White Plains, New York, 30 October - 02 November 1996.

[60] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface Mutation: An Approach for Integration Testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, May 2001.

[61] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur. Interface Mutation Test Adequacy Criterion: An Empirical Evaluation. Technique report, State University of Maringá, Parana, Brasil, 2000.

[62] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur. Interface Mutation Test Adequacy Criterion: An Empirical Evaluation. *Empirical Software Engineering*, 6(2):111–142, June 2001.

[63] M. E. Delamaro, J. C. Maldonado, and A. Vincenzi. Proteum/IM 2.0: An Integrated Mutation Testing Environment. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 91–101, San Jose, California, 6-7 October 2001. published in book form, as *Mutation Testing for the New Century.*

[64] R. A. DeMillo. Program Mutation: An Approach to Software Testing. Technical report, Georgia Institute of Technology, 1983.

[65] R. A. DeMillo, D. S. Guindi, K. N. King, and W. M. McCracken. An Overview of the Mothra Software Testing Environment. Technique Report SERC-TR-3-P, Purdue University, West Lafayette, Indiana, 1987.

[66] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An Extended Overview of the Mothra Software Testing Environment. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*

(TVA'88), pages 142–151, Banff Alberta,Canada, July 1988. IEEE Computer society.

[67] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-Integrated Program Mutation. In *Proceedings of the 5th Annual Computer Software and Applications Conference (COMPSAC'91)*, pages 351–356, Tokyo, Japan, September 1991. IEEE Computer Society Press.

[68] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[69] R. A. DeMillo and A. P. Mathur. On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis in Detecting Errors in Production Software. Technique Report SERC-TR-92-P, Purdue University, West Lafayette, Indiana, 1992.

[70] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[71] R. A. DeMillo and A. J. Offutt. Experimental Results From an Automatic Test Case Generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.

[72] A. Derezińska. Object-oriented Mutation to Assess the Quality of Tests. In *Proceedings of the 29th Euromicro Conference*, pages 417– 420, Belek, Turkey, 1-6 September 2003.

[73] A. Derezińska. Advanced Mutation Operators Applicable in C# Programs. Technique report, Warsaw University of Technology, Warszawa, Poland, 2005.

[74] A. Derezińska. Quality Assessment of Mutation Operators Dedicated for C# Programs. In *Proceedings of the 6th International Conference on Quality Software (QSIC'06)*, Beijing, China, 27-28 October 2006.

[75] A. Derezińska and A. Szustek. CREAM- A System for Object-Oriented Mutation of C# Programs. Technique report, Warsaw University of Technology, Warszawa, Poland, 2007.

[76] A. Derezińska and A. Szustek. Tool-Supported Advanced Mutation Approach for Verification of C# Programs. In *Proceedings of the 3th International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX'08)*, pages 261–268, Szklarska Porêba, Poland, 26-28 June 2008.

[77] H. Do, S. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4):405 – 435, Oct. 2005.

[78] H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[79] H. Do and G. Rothermel. A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 411–420, Budapest, Hungary, 25-30 September 2005.

[80] H. Do and G. Rothermel. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, September 2006.

[81] J. J. Domínguez-Jiménez, A. Estero-Botaro, and I. Medina-Bulo. A Framework for Mutant Genetic Generation for WS-BPEL. In *Proceedings of the 35th*

Conference on Current Trends in Theory and Practice of Computer Science, volume 5404 of *LNCS*, pages 229 – 240, Spindleruv Mlyn, Czech Republic, January 2009. Springer.

[82] W. Du and A. P. Mathur. Vulnerability Testing of Software System Using Fault Injection. Technique Report COAST TR 98-02, Purdue University, West Lafayette, Indiana, 1998.

[83] W. Du and A. P. Mathur. Testing for Software Vulnerability Using Environment Perturbation. In *Proceeding of the International Conference on Dependable Systems and Networks (DSN'00)*, pages 603–612, New York, NY, 25-28 June 2000.

[84] L. du Bousquet and M. Delaunay. Mutation Analysis for Lustre programs: Fault Model Description and Validation. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 176–184, Windsor, UK, 10-14 September 2007. IEEE Computer Society. published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*.

[85] L. du Bousquet and M. Delaunay. Using Mutation Analysis to Evaluate Test Generation Strategies in a Synchronous Context. In *Proceedings of the 2nd International Conference on Software Engineering Advances (ICSEA'07)*, page 40, Cap Esterel, French Riviera, France, 25-31 August 2007.

[86] V. Durelli, J. Offutt, and M. Delamaro. Toward harnessing high-level language virtual machines for further speeding up weak mutation testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 681 –690, april 2012.

[87] B. Dutertre and L. M. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, pages 81–94, 2006.

[88] S. Eldh, S. Punnekkat, H. Hansson, and P. Jönsson. Component Testing Is Not Enough - A Study of Software Faults in Telecom Middleware. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom'07) and the 7th International Workshop (FATES'07)*, Tallinn, Estonia, 26-29 June 2007.

[89] Ellims. Csaw. http://www.skicambridge.com/papers/Csaw_v1_files.html, 2007.

[90] M. Ellims, D. C. Ince, and M. Petre. The Csaw C Mutation Tool: Initial Results. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 185–192, Windsor, UK, 10-14 September 2007. IEEE Computer Society. published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07).*

[91] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. Mutation operators for WS-BPEL 2.0. In *Proceedings of the 21th International Conference on Software and Systems Engineering and their Applications (ICSSEA'08)*, Paris, France, 9-11 December 2008.

[92] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing. In *Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC'99)*, page 96, Talca, Chile, 11-13 November 1999.

[93] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and W. E. Wong. Mutation Testing Applied to Validate Specifications Based on Petri Nets. In *Proceedings of the IFIP TC6 8th International Conference on Formal Description Techniques VIII*, volume 43, pages 329–337, 1995.

[94] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation Testing Applied to Validate Specifications Based on Statecharts. In *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, page 210, Boca Raton, Florida, 1-4 November 1999.

[95] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. Masiero. Mutation Analysis Testing for Finite State Machines. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 220–229, Monterey, California, 6-9 November 1994.

[96] X. Feng, S. Marr, and T. O'Callaghan. ESTP: An Experimental Software Testing Platform. In *Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)*, pages 59–63, Windsor, UK, 29-31 August 2008.

[97] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation Testing for Aspect-Oriented Programs. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, pages 52–61, Lillehammer, Norway, 9-11 April 2008. IEEE Computer Society.

[98] V. N. Fleyshgakker and S. N. Weiss. Efficient Mutation Analysis: A New Approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'94)*, pages 185–195, Seattle, Washington, August 1994. ACM Press.

[99] P. G. Frankl, S. N. Weiss, and C. Hu. All-Uses Versus Mutation Testing: An Experimental Comparison of Effectiveness. Technique report, Polytechnic University, Brooklyn, New York, 1994.

[100] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs Mutation Testing: an Experimental Comparison of Effectiveness. *Journal of Systems and Software*, 38(3):235–253, September 1997.

[101] G. Fraser and F. Wotawa. Mutant Minimization for Model-Checker Based Test-Case Generation. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 161–168, Windsor, UK, 10-14 September 2007. IEEE Computer Society. published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*.

[102] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*, pages 147–158, Trento, Italy, 12-16 July ISSTA '10. ACM.

[103] R. Geist, A. J. Offutt, and F. C. Harris. Estimation and Enhancement of Real-Time Software Reliability Through Mutation Analysis. *IEEE Transactions on Computers*, 41(5):550–558, May 1992.

[104] A. K. Ghosh, T. O'Connor, and G. McGraw. An Automated Approach for Identifying Potential Vulnerabilities in Software. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'98)*, pages 104–114, Oakland, California, 3-6 May 1998.

[105] S. Ghosh. *Testing Component-Based Distributed Applications*. Phd thesis, Purdue University, West Lafayette, Indiana, 2000.

[106] S. Ghosh. Towards Measurement of Testability of Concurrent Object-oriented Programs Using Fault Insertion: a Preliminary Investigation. In *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, page 7, Los Alamitos, California, 2002.

[107] S. Ghosh, P. Govindarajan, and A. P. Mathur. TDS: a Tool for Testing Distributed Component-Based Applications. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 103–112, San Jose, California, 6-7 October 2001. published in book form, as *Mutation Testing for the New Century*.

[108] S. Ghosh and A. P. Mathur. Interface Mutation to Assess the Adequacy of Tests for Componentsand Systems. In *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, page 37, Santa Barbara, California, 30 July - 4 August 2000.

[109] S. Ghosh and A. P. Mathur. Interface Mutation. *Software Testing, Verification and Reliability*, 11(3):227–247, March 2001.

[110] M. R. Girgis and M. R. Woodward. An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis. In *Proceedings of the 8th International Conference on Software Engineering (ICSE'85)*, pages 313–319, London, England, August 1985. IEEE Computer Society Press.

[111] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, volume 40 of *6*, pages 213–223, Chicago, Illinois, USA, 11–15 June 2005. ACM.

[112] A. S. Gopal and T. A. Budd. Program Testing by Specification Mutation. Technical Report TR 83-17, University of Arizona, Tucson, Arizona, 1983.

[113] B. J. M. Grün, D. Schuler, and A. Zeller. The Impact of Equivalent Mutants. In *Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09)*, pages 192–199, Denver, Colorado, 1-4 April 2009. IEEE Computer Society. published with *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops.*

[114] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[115] J. M. Hanks. *Testing Cobol Programs by Mutation.* Phd thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.

[116] M. Harman. The Current State and Future of Search Based Software Engineering. In *Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, USA, 2007.

[117] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The Impact of Input Domain Reduction on Search-Based Test Data Generation. In *ACM Symposium on the Foundations of Software Engineering (FSE '07)*, pages 155–164, Dubrovnik, Croatia, September 2007.

[118] M. Harman, R. M. Hierons, and S. Danicic. The Relationship Between Program Dependence and Mutation Analysis. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 5–13, San Jose, California, 6-7 October 2001. published in book form, as *Mutation Testing for the New Century.*

[119] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability Transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

[120] M. Harman, Y. Jia, and W. B. Langdon. A Manifesto for Higher Order Mutation Testing. In *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'10)*, Paris, France, 6 April 2010. IEEE Computer Society. published with *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops.*

[121] M. Harman and B. F. Jones. Search-based Software Engineering. *Information and Software Technology*, 43(14):833–839, December 2001.

[122] M. Harman and P. McMinn. A Theoretical and Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation. In *International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 73 – 83, London, United Kingdom, July 2007.

[123] M. Harman and P. McMinn. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[124] R. M. Hierons, M. Harman, and S. Danicic. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, December 1999.

[125] R. M. Hierons and M. G. Merayo. Mutation Testing from Probabilistic Finite State Machines. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 141–150, Windsor, UK, 10-14 September 2007. IEEE Computer Society. published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07).*

[126] R. M. Hierons and M. G. Merayo. Mutation Testing from Probabilistic and Stochastic Finite State Machines. *Journal of Systems and Software*, 82(11):1804–1818, November 2009.

[127] J. R. Horgan and A. P. Mathur. Weak Mutation is Probably Strong Mutation. Technical Report SERC-TR-83-P, Purdue University, West Lafayette, Indiana, 1990.

[128] S.-S. Hou, L. Zhang, T. Xie, H. Mei, and J.-S. Sun. Applying Interface-Contract Mutation in Regression Testing of Component-Based Software. In *Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07)*, pages 174–183, Paris, France, 2-5 October 2007.

[129] W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[130] S. Hussain. *Mutation Clustering.* Masters thesis, King's College London, UK, 2008.

[131] J. Hwang, T. Xie, F. Chen, and A. X. Liu. Systematic Structural Testing of Firewall Policies. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS '08)*, pages 105–114, Napoli, Italy, 6-8 October 2008.

[132] Itregister. Plextest. http://www.itregister.com.au/products/plextest.htm, 2007.

[133] D. Jackson and M. R. Woodward. Parallel firm mutation of Java programs. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 55–61, San Jose, California, 6-7 October 2001. published in book form, as *Mutation Testing for the New Century.*

[134] C. Ji, Z. Chen, B. Xu, and Z. Zhao. A Novel Method of Mutation Clustering Based on Domain Analysis. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'09)*, Boston, Massachusetts, 1-3 July 2009. Knowledge Systems Institute Graduate School.

[135] Y. Jia. Mutation Testing Repository. http://www.dcs.kcl.ac.uk/pg /jiayue/repository/, 2009.

[136] Y. Jia and M. Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. In *Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 249–258, Beijing, China, 28-29 September 2008.

[137] Y. Jia and M. Harman. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)*, pages 94–98, Windsor, UK, 29-31 August 2008. IEEE Computer Society.

[138] Y. Jia and M. Harman. Higher Order Mutation Testing. *Journal of Information and Software Technology*, 51(10):1379–1393, October 2009.

[139] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering*, To appear, 2010.

[140] C. Jing, Z. Wang, X. Shi, X. Yin, and J. Wu. Mutation Testing of Protocol Messages Based on Extended TTCN-3. In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications (AINA'08)*, pages 667–674, Okinawa, Japan, 25-28 March 2008.

[141] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 612–615, Washington, DC, USA, 2011. IEEE Computer Society.

[142] K. Kapoor. Formal Analysis of Coupling Hypothesis for Logical Faults. *Innovations in Systems and Software Engineering*, 2(2):80–87, July 2006.

[143] S. Kim, J. A. Clark, and J. A. McDermid. Assessing Test Set Adequacy for Object Oriented Programs Using Class Mutation. In *Proceedings of the 3rd Symposium on Software Technology (SoST'99)*, Buenos Aires, Argentina, 8-9 September 1999.

[144] S. Kim, J. A. Clark, and J. A. McDermid. The Rigorous Generation of Java Mutation Operators Using HAZOP. In *Proceedings of the 12th International Cofference Software and Systems Engineering and their Applications (ICSSEA 99)*, Paris, France, 29 November-1 December 1999.

[145] S. Kim, J. A. Clark, and J. A. McDermid. Class Mutation: Mutation Testing for Object-oriented Programs. In *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, 2000.

[146] S. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 207–225, San Jose, California, 6-7 October 2001. published in book form, as *Mutation Testing for the New Century.*

[147] S.-W. Kim, M. J. Harrold, and Y.-R. Kwon. MUGAMMA: Mutation Analysis of Deployed Software to Increase Confidence and Assist Evolution. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 10, Raleigh, North Carolina, November 2006. IEEE Computer Society.

[148] K. N. King and A. J. Offutt. A Fortran Language System for Mutation-Based Software Testing. *Software:Practice and Experience*, 21(7):685–718, October 1991.

[149] M. Kintis, M. Papadakis, and N. Malevris. Isolating first order equivalent mutants via second order mutation. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 701–710, Washington, DC, USA, 2012. IEEE Computer Society.

[150] E. W. Krauser. *Compiler-Integrated Software Testing.* Phd thesis, Purdue University, West Lafyette, 1991.

[151] E. W. Krauser, A. P. Mathur, and V. J. Rego. High Performance Software Testing on SIMD Machines. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)*, pages 171 – 177, Banff Alberta, July 1988. IEEE Computer Society.

[152] E. W. Krauser, A. P. Mathur, and V. J. Rego. High Performance Software Testing on SIMD Machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, May 1991.

[153] K. Lakhotia, P. McMinn, and M. Harman. Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet? In 4^{th} *Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART'09)*, pages 95–104, Windsor, UK, 4th–6th September 2009.

[154] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy — Search-Based Floating Point Constraint Solving for Symbolic Execution. In 22^{nd} *IFIP International Conference on Testing Software and Systems (ICTSS 2010)*, pages 142–157, Natal, Brazil, November 2010. LNCS Volume 6435.

[155] W. B. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of systems and Software*, 83:2416–2430, December 2010.

[156] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[157] Y. Le Traon, T. Mouelhi, and B. Baudry. Testing Security Policies: Going Beyond Functional Testing. In *The 18th IEEE International Symposium on Software Reliability*, pages 93–102, Trollhättan, Sweden, 5-9 November 2007. IEEE Computer Society.

[158] S. Lee, X. Bai, and Y. Chen. Automatic Mutation Testing and Simulation on OWL-S Specified Web Services. In *Proceedings of the 41st Annual Simulation Symposium (ANSS'08)*, pages 149–156, Ottawa, Canada., 14-16 April 2008.

[159] S. C. Lee and A. J. Offutt. Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, pages 200–209, Hong Kong, China, November 2001.

[160] J. B. Li and J. Miller. Testing the Semantics of W3C XML Schema. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, pages 443–448, Turku, Finland, 26-28 July 2005.

[161] N. Li, U. Praphamontripong, and A. J. Offutt. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-uses and Prime Path Coverage. In *Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09)*, pages 220–229, Denver, Colorado, 1-4 April 2009. IEEE Computer Society. published with *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops.*

[162] R. Lipton. Fault Diagnosis of Computer Programs. Student Report, Carnegie Mellon University, 1971.

[163] R. J. Lipton and F. G. Sayward. The Status of Research on Program Mutation. In *Proceedings of the Workshop on Software Testing and Test Documentation*, pages 355–373, December 1978.

[164] M.-H. Liu, Y.-F. Gao, J.-H. Shan, J.-H. Liu, L. Zhang, and J.-S. Sun. An Approach to Test Data Generation for Killing Multiple Mutants. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 113–122, Philadelphia, Pennsylvania, USA, 24-27 September 2006.

[165] B. Long, R. Duke, D. Goldson, P. Strooper, and L. Wildman. Mutation-based Exploration of a Method for Verifying Concurrent Java Components. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 265, Santa Fe, New Mexico, 26-30 April 2004.

[166] Y.-S. Ma, M. J. Harrold, and Y.-R. Kwon. Evaluation of Mutation Testing for Object-Oriented Programs. In *Proceedings of the 28th international Conference on Software Engineering (ICSE '06)*, pages 869–872, Shanghai, China, 20-28 May 2006.

[167] Y.-S. Ma, Y.-R. Kwon, and A. J. Offutt. Inter-class Mutation Operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 352, Annapolis, Maryland, 12-15 November 2002. IEEE Computer Society.

[168] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon. MuJava: An Automated Class Mutation System. *Software Testing, Verification & Reliability*, 15(2):97–133, June 2005.

[169] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon. MuJava: a Mutation System for Java. In *Proceedings of the 28th international Conference on Software Engineering (ICSE '06)*, pages 827–830, Shanghai, China, 20-28 May 2006.

[170] P. Madiraju and A. S. Namin. ParaMu - A Partial and Higher-Order Mutation Tool with Concurrency Operators. In *Proceedings of the 6th International Workshop on Mutation Analysis (Mutation 2011)*, Berlin, Germany, March 2011.

[171] B. Marick. The Weak Mutation Hypothesis. In *Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification (TAV'91)*, pages 190–199, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society.

[172] E. E. Martin and T. Xie. A Fault Model and Mutation Testing of Access Control Policies. In *Proceedings of the 16th International Conference on World Wide Web*, pages 667–676, Banff, Alberta, Canada, 8-12 May 2007. ACM.

[173] P. R. Mateo, M. P. Usaola, and J. Offutt. Mutation at system and functional levels. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 110–119, Washington, DC, USA, 2010. IEEE Computer Society.

[174] A. P. Mathur. Performance, Effectiveness, and Reliability Issues in Software Testing. In *Proceedings of the 5th International Computer Software and Applications Conference (COMPSAC'79)*, pages 604–605, Tokyo, Japan, 11-13 September 1991.

[175] A. P. Mathur. CS 406 Software Engineering I. Course Project Handout, August 1992.

[176] A. P. Mathur and E. W. Krauser. Mutant Unification for Improved Vectorization. Technique Report SERC-TR-14-P, Purdue University, West Lafayette, Indiana, 1988.

[177] A. P. Mathur and W. E. Wong. An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria. Technique report, Purdue University, West Lafayette, Indiana, 1993.

[178] A. P. Mathur and W. E. Wong. An Empirical Comparison of Data Flow and Mutation-based Test Adequacy Criteria. *Software Testing, Verification and Reliability*, 4(1):9 – 31, 1994.

[179] P. McMinn. Search-Based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[180] M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 1996.

[181] I. Moore. Jester - a JUnit test tester. In *Proceeding of eXtreme Programming Conference (XP'01)*, 2001.

[182] I. Moore. Jester and Pester. http://jester.sourceforge.net/, 2001.

[183] L. J. Morell. *A Theory of Error-Based Testing*. Phd thesis, University of Maryland at College Park, College Park, Maryland, 1984.

[184] L. J. Morell. Theoretical Insights Into Fault-Based Testing. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)*, pages 45–62, Banff Alberta, Canada, July 1988. IEEE Computer Society.

[185] L. J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[186] T. Mouelhi, F. Fleurey, and B. Baudry. A Generic Metamodel For Security Policies Mutation. In *Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)*, pages 278–286, Lillehammer, Norway, 9-11 April 2008. IEEE Computer Society.

[187] T. Mouelhi, Y. Le Traon, and B. Baudry. Mutation Analysis for Security Tests Qualification. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 233–242, Windsor, UK, 10-14 September 2007. IEEE Computer Society. published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*.

[188] E. S. Mresa and L. Bottaci. Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study. *Software Testing, Verification and Reliability*, 9(4):205–232, December 1999.

[189] A. S. Namin and J. H. Andrews. Finding Sufficient Mutation Operators via Variable Reduction. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 5, Raleigh, North Carolina, November 2006. IEEE Computer Society.

[190] A. S. Namin and J. H. Andrews. On Sufficiency of Mutants. In *Proceedings of the 29th International Conference on Software Engineering (ICSE COMPANION'07)*, pages 73–74, Minneapolis, Minnesota, 20-26 May 2007.

[191] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient Mutation Operators for Measuring Test Effectiveness. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 351–360, Leipzig, Germany, 10-18 May 2008.

[192] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 209–265. Springer Berlin / Heidelberg, 2002.

[193] R. Nilsson, A. J. Offutt, and S. F. Andler. Mutation-based Testing Criteria for Timeliness. In *Proceedings of the 28th Annual International Computer*

Software and Applications Conference (COMPSAC'04), pages 306–311, Hong Kong, China, 28-30, September 2004.

[194] R. Nilsson, A. J. Offutt, and J. Mellin. Test Case Generation for Mutation-based Testing of Timeliness. In *Proceedings of the 2nd Workshop on Model Based Testing (MBT 2006)*, volume 164 of *ENTCS*, pages 97–114, Vienna, Austria, 25-26 March 2006.

[195] A. J. Offutt. *Automatic Test Data Generation.* Phd thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1988.

[196] A. J. Offutt. The Coupling Effect: Fact or Fiction. *ACM SIGSOFT Software Engineering Notes*, 14(8):131–140, December 1989.

[197] A. J. Offutt. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, January 1992.

[198] A. J. Offutt. Private Communication, July 2008.

[199] A. J. Offutt, P. Ammann, and L. L. Liu. Mutation Testing implements Grammar-Based Testing. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 12, Raleigh, North Carolina, November 2006. IEEE Computer Society.

[200] A. 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.

[201] A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Approach for Test Data Generation: Design and Algorithms. Technical Report ISSE-TR-94-110, George Mason University, Fairfax, Virginia, 1994.

[202] A. J. Offutt and K. N. King. A Fortran 77 Interpreter for Mutation Analysis. *ACM SIGPLAN Notices*, 22(7):177–188, July 1987.

[203] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[204] A. J. Offutt and S. Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[205] A. J. Offutt and S. D. Lee. How Strong is Weak Mutation? In *Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification (TAV'91)*, pages 200 – 213, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society.

[206] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon. An Experimental Mutation System for Java. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, September 2004.

[207] A. J. Offutt and J. Pan. Detecting Equivalent Mutants and the Feasible Path Problem. In *Proceedings of the 1996 Annual Conference on Computer Assurance*, pages 224–236, Gaithersburg, Maryland, June 1996. IEEE Computer Society Press.

[208] A. J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability*, 7(3):165–192, September 1997.

[209] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An Experimental Evaluation of Data Flow and Mutation Testing. *Software:Practice and Experience*, 26(2):165–176, February 1996.

[210] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for Reducing the Size of Coverage-based Test Sets. In *Proceedings of the 12 International Conference on Testing Computer Software*, pages 111–123, Washington, DC, June 1995.

[211] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation Testing of Software Using a MIMD Computer. In *Proceedings of the International Conference on Parallel Processing*, pages 255–266, Chicago, Illinois, August 1992.

[212] A. J. Offutt, G. Rothermel, and C. Zapf. An Experimental Evaluation of Selective Mutation. In *Proceedings of the 15th International Conference on Software Engineering (ICSE'93)*, pages 100–107, Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[213] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the Orthogonal. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 34–44, San Jose, California, 6-7 October 2001. published in book form, as *Mutation Testing for the New Century.*

[214] A. J. Offutt, J. Voas, and J. Payne. Mutation Operators for Ada. Technique Report ISSE-TR-96-09, George Mason University, Fairfax, Virginia, 1996.

[215] A. J. Offutt and W. Xu. Generating Test Cases for Web Services Using Data Perturbation. In *Proceedings of the Workshop on Testing, Analysis and Verification of Web Services (TAV-WEB)*, pages 1 – 10, Boston, Massachusetts, 11-14 July 2004.

[216] A. J. Offutt, J. Zhenyi, and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software:Practice and Experience*, 29(2):167–193, February 1999.

[217] A. J. Offutt, J. Zhenyi, and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. *Software:Practice and Experience*, 29(2):167–193, February 1999.

[218] J. Offutt. Private communication, March 2013.

[219] V. Okun. *Specification Mutation for Test Generation and Analysis.* Phd thesis, University of Maryland Baltimore County, Baltimore, Maryland, 2004.

[220] T. Olsson and P. Runeson. System Level Mutation Analysis Applied to a State-based Language. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'01)*, page 222, Washington DC, 17-20 April 2001.

[221] J. Pan. Using Constraints to Detect Equivalent Mutants. Masters thesis, George Mason University, Fairfax VA, 1994.

[222] M. Papadakis and N. Malevris. An Effective Path Selection Strategy for Mutation Testing. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC'09)*, pages 422 – 429, Penang, Malaysia, 1-3 December 2009. IEEE Computer Society.

[223] M. Papadakis and N. Malevris. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In *Proceedings of the 5th International Workshop on Mutation Analysis (MUTATION'10)*, Paris, France, 6 April 2010. IEEE Computer Society. published with *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops.*

[224] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Proceedings of the 21st International*

Symposium on Software Reliability Engineering (ISSRE'10), California, USA, November 2010.

[225] M. Papadakis, N. Malevris, and M. Kallia. Towards Automating the Generation of Mutation Tests. In *Proceedings of the 5th Workshop on Automation of Software Teste (AST'10)*, pages 111–118, Cape Town, South Africa, 3-4 May 2010. ACM.

[226] Parasoft. Parasoft Insure++. http://www.parasoft.com/jsp/products/home.jsp?product=Insure, 2006.

[227] M. Polo, M. Piattini, and I. Garcia-Rodriguez. Decreasing the Cost of Mutation Testing with Second-Order Mutants. *Software Testing, Verification and Reliability*, 19(2):111 – 131, June 2008.

[228] M. Polo, S. Tendero, and M. Piattini. Integrating techniques and tools for testing automation: Research Articles. *Software Testing, Verification and Reliability*, 17(1):3–39, March 2007.

[229] A. Pretschner, T. Mouelhi, and Y. Le Traon. Model-Based Tests for Access Control Policies. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, pages 338–347, Lillehammer, Norway, 9-11 April 2008. IEEE Computer Society.

[230] R. Probert and F. Guo. Mutation Testing of Protocols: Principles and Preliminary Experimental Results. In *Proceedings of the Workshop on Protocol Test Systems*, pages 57–76, Leidschendam, Netherland, 15-17 October 1991.

[231] R. Purushothaman and D. E. Perry. Toward Understanding the Rhetoric of Small Source Code Changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[232] S. T. Redwine and W. E. Riddle. Software Technology Maturation. In *Proceedings of the 8th International Conference on Software Engineering*, pages 189–200, London, England, 1985.

[233] ReelTwo. http://www.reeltwo.com, 2007.

[234] C. K. Roy and J. R. Cordy. Towards a Mutation-based Automatic Framework for Evaluating Code Clone Detection Tools. In *Proceedings of the Canadian Conference on Computer Science and Software Engineering (C3S2E'08)*, pages 137–140, Montreal, Quebec, Canada, 12-13 May 2008. ACM.

[235] C. K. Roy and J. R. Cordy. A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools. In *Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09)*, pages 157–166, Denver, Colorado, 1-4 April 2009. IEEE Computer Society. published with *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops.*

[236] Rubyforge. Heckle. http://seattlerb.rubyforge.org/heckle/, 2007.

[237] M. Sahinoglu and E. H. Spafford. A Bayes Sequential Statistical Procedure for Approving Software Products. In *Proceedings of the IFIP Conference on Approving Software Products (ASP'90)*, pages 43–56, Garmisch Partenkirchen, Germany, September 1990. Elsevier Science.

[238] D. Schuler, V. Dallmeier, and A. Zeller. Efficient Mutation Testing by Checking Invariant Violations. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'09)*, Chicago, Illinois, 19-23 July 2009.

[239] D. Schuler, V. Dallmeier, and A. Zeller. Efficient Mutation Testing by Checking Invariant Violations. Technique report, Saarland University, Saarbrucken, Telefon, 2009.

[240] D. Schuler and A. Zeller. (Un-)Covering Equivalent Mutants. In *Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST'10)*, Paris, France, 6 April 2010. IEEE Computer Society.

[241] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)*, pages 263–272, Lisbon, Portugal, 2005.

[242] Y. Serrestou, V. Beroulle, and C. Robach. Functional Verification of RTL Designs Driven by Mutation Testing Metrics. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 222–227, Lubeck, Germany, 29-31 August 2007.

[243] Y. Serrestou, V. Beroulle, and C. Robach. Impact of Hardware Emulation on the Verification Quality Improvement. In *Proceedings of the IFIP WG 10.5 International Conference on Very Large Scale Integration of System-on-Chip (VLSI-SoC'07)*, pages 218–223, Atlanta, GA, 15-17 October 2007.

[244] H. Shahriar and M. Zulkernine. MUSIC: Mutation-based SQL Injection Vulnerability Checking. In *Proceedings of the 8th International Conference on Quality Software (QSIC'08)*, pages 77–86, Oxford, UK, 12-13 August 2008.

[245] H. Shahriar and M. Zulkernine. Mutation-Based Testing of Buffer Overflow Vulnerabilities. In *Proceedings of the 2nd Annual IEEE International Workshop on Security in Software Engineering*, pages 979–984, 28 July -1 August, Turku, Finland 2008.

[246] H. Shahriar and M. Zulkernine. Mutation-Based Testing of Format String Bugs. In *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, pages 229–238, Nanjing, China, 3-5 Dec 2008.

[247] H. Shahriar and M. Zulkernine. MUTEC: Mutation-based Testing of Cross Site Scripting. In *Proceedings of the 5th International Workshop on Software Engineering for Secure Systems (SESS'09)*, pages 47–53, Vancouver, Canada, 19 May 2009.

[248] D. P. Sidhu and T. K. Leung. Fault Coverage of Protocol Test Methods. In *Proceedings of the 7th Annual Joint Conference of the IEEE Computer and Communcations Societies (INFOCOM'88)*, pages 80–85, New Orleans, Louisiana, 27-31 March 1988.

[249] A. Simao, J. C. Maldonado, and R. da Silva Bigonha. A Transformational Language for Mutant Description. *Computer Languages, Systems & Structures*, 35(3):322–339, October 2009.

[250] B. H. Smith and L. Williams. An Empirical Evaluation of the MuJava Mutation Operators. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 193–202, Windsor, UK, 10-14 September 2007. IEEE Computer Society. published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*.

[251] SourceForge. Nester. http://nester.sourceforge.net/, 2002.

[252] SourceForge. Jumble. http://jumble.sourceforge.net/, 2007.

[253] S. D. R. S. D. Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. L. D. Souza. Mutation Testing Applied to Estelle Specifications. *Software Quality Control*, 8(4):285–301, December 1999.

[254] S. D. R. S. D. Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. L. D. Souza. Mutation Testing Applied to Estelle Specifications. In *Proceedings of the 33rd*

Hawaii International Conference on System Sciences (HICSS'08), volume 8, page 8011, Maui, Hawaii, 4-7 January 2000.

[255] E. H. Spafford. Extending Mutation Testing to Find Environmental Bugs. *Software:Practice and Experience*, 20(2):181–189, February 1990.

[256] T. Srivatanakul, J. A. Clark, S. Stepney, and F. Polack. Challenging Formal Specifications by Mutation: a CSP Security Example. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 340–350, Chiang Mai, Thailand, 10-12 December 2003.

[257] T. Sugeta, J. C. Maldonado, and W. E. Wong. Mutation Testing Applied to Validate SDL Specifications. In *Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems*, volume 2978 of *LNCS*, page 2741, Oxford, UK, 17-19 March 2004.

[258] A. Sung, J. Jang, and B. Choi. Fault-Based Interface Testing Between Real-Time Operating System and Application. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 8, Raleigh, North Carolina, November 2006. IEEE Computer Society.

[259] A. Tanaka. *Equivalence Testing for Fortran Mutation System Using Data Fow Analysis*. Masters thesis, Georgia Institute of Technology, Atlanta, Georgia, 1981.

[260] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An Experimental Study on Software Structural Testing: Deterministic versus Random Input Generation. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS'91)*, pages 410–417, Montréal, Canada, 25-27 June 1991.

[261] N. Tillmann and J. de Halleux. Pex–White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP'08)*, volume 4966, pages 134–153, Prato, Italy, April 2008.

[262] M. Trakhtenbrot. New Mutations for Evaluation of Specification and Implementation Levels of Adequacy in Testing of Statecharts Models. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 151–160, Windsor, UK, 10-14 September 2007. IEEE Computer Society. published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*.

[263] U. Trier. DBLP. http://www.informatik.uni-trier.de/ ley/db/.

[264] J. Tuya, M. J. S. Cabal, and C. de la Riva. SQLMutation: A Tool to Generate Mutants of SQL Database Queries. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 1, Raleigh, North Carolina, November 2006. IEEE Computer Society.

[265] J. Tuya, M. J. S. Cabal, and C. de la Riva. Mutating Database Queries. *Information and Software Technology*, 49(4):398–417, April 2007.

[266] R. H. Untch. Mutation-based Software Testing Using Program Schemata. In *Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE'92)*, pages 285–291, Raleigh, North Carolina, 1992.

[267] R. H. Untch. *Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method*. Phd thesis, Clemson University, Clemson, South Carolina, December 1995. Adviser-Harrold, Mary Jean.

[268] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation Analysis Using Mutant Schemata. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 139–148, Cambridge, Massachusetts, 1993.

[269] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures using Mutant Exploits. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 21–30, Washington DC, USA, 2004.

[270] P. Vilela, M. Machado, and W. E. Wong. Testing for Security Vulnerabilities in Software. In *Software Engineering and Applications*, 2002.

[271] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro. Unit and Integration Testing Strategies for C Programs Using Mutation. *Software Testing, Verification and Reliability*, 11(4):249–268, November 2001.

[272] J. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.

[273] K. S. H. T. Wah. Fault Coupling in Finite Bijective Functions. *Software Testing, Verification and Reliability*, 5(1):3–47, 1995.

[274] K. S. H. T. Wah. A Theoretical Study of Fault Coupling. *Software Testing, Verification and Reliability*, 10(1):3–46, April 2000.

[275] K. S. H. T. Wah. An Analysis of the Coupling Effect I: Single Test Data. *Science of Computer Programming*, 48(2-3):119–161, August-September 2003.

[276] R. Wang and N. Huang. Requirement Model-Based Mutation Testing for Web Service. In *Proceedings of the 4th International Conference on Next Generation Web Services Practices (NWeSP'08)*, pages 71–76, Seoul, Republic of Korea, 20-22 October 2008.

[277] S. N. Weiss and V. N. Fleyshgakker. Improved Serial Algorithms for Mutation Analysis. *ACM SIGSOFT Software Engineering Notes*, 18(3):149–158, July 1993.

[278] E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25:456–470, 1982.

[279] W. E. Wong. *On Mutation and Data Flow.* Phd thesis, Purdue University, West Lafayette, Indiana, 1993.

[280] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of Test Set Minimization on Fault Detection Effectiveness. *Software:Practice and Experience*, 28:347–369, 1998.

[281] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application. *Journal of Systems and Software*, 48(2):79–89, October 1999.

[282] W. E. Wong and A. P. Mathur. Fault Detection Effectiveness of Mutation and Data Flow Testing. *Software Quality Journal*, 4(1):69–83, March 1995.

[283] W. E. Wong and A. P. Mathur. Reducing the Cost of Mutation Testing: An Empirical Study. *Journal of Systems and Software*, 31(3):185–196, December 1995.

[284] M. R. Woodward. Mutation Testing-An Evolving Technique. In *Proceedings of the IEE Colloquium on Software Testing for Critical Systems*, pages 3/1–3/6, London, UK, 19 June 1990.

[285] M. R. Woodward. OBJTEST: an Experimental Testing Tool for Algebraic Specifications. In *Proceedings of the IEE Colloquium on Automating Formal Methods for Computer Assisted Prototying*, page 2, 14 Jan 1990.

[286] M. R. Woodward. Errors in Algebraic Specifications and an Experimental Mutation Testing Tool. *Software Engineering Journal*, 8(4):221–224, July 1993.

[287] M. R. Woodward and K. Halewood. From Weak to Strong, Dead or Alive? an Analysis of Some Mutationtesting Issues. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)*, pages 152–158, Banff Albert, Canada, July 1988. IEEE Computer Society.

[288] W. Xu, A. J. Offutt, and J. Luo. Testing Web Services by XML Perturbation. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 257–266, Chicago Illinois, 14-16 July 2005.

[289] H. Yoon, B. Choi, and J. O. Jeon. Mutation-Based Inter-Class Testing. In *Proceedings of the 5th Asia Pacific Software Engineering Conference (APSEC'98)*, page 174, Taipei, Taiwan, 2-4 December 1998.

[290] C. N. Zapf. *A Distributed Interpreter for the Mothra Mutation Testing System.* Masters thesis, Clemson University, Clemson, South Carolina, 1993.

[291] Y. Zhan and J. A. Clark. Search-based Mutation Testing for Simulink Models. In *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO'05)*, pages 1061–1068, Washington DC, USA, 25-29 June 2005.

[292] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 435–444, New York, NY, USA, 2010. ACM.

[293] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM'10)*, Timisoara, Romania, September 2010.

[294] S. Zhang, T. R. Dean, and G. S. Knight. Lightweight State Based Mutation Testing for Security. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 223–232, Windsor, UK, 10-14 September 2007. IEEE Computer Society. published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*.

[295] C. Zhou and P. Frankl. Mutation Testing for Java Database Applications. In *Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST'09)*, pages 396–405, Davor Colorado, 01-04 April 2009.