An Analysis and Survey of the Development of Mutation Testing

Yue Jia and Mark Harman

Technical Report TR-09-06

***Abstract*— Mutation Testing is a fault–based software testing technique that has been widely studied for over three decades. The literature on Mutation Testing has contributed a set of approaches, tools, developments and empirical results which have not been surveyed in detail until now. This paper provides a comprehensive analysis and survey of Mutation Testing. The paper also 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.**

***Index Terms*— mutation testing, survey**

I. Introduction

Mutation Testing is a fault-based testing technique which provides a testing criterion called the "mutation adequacy score". The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults.

The general principle underlying Mutation Testing work is that the faults used by Mutation Testing represent the mistakes that programmers often make and other related testing heuristics. Such faults are deliberately seeded into the original program, by a simple syntactic change, to create a set of faulty programs called mutants, each containing a different syntactic change. To assess the quality of a given test set, these mutants are executed against the input test set to see if the seeded faults can be detected. The history of Mutation Testing can be traced back to 1971 in a student paper by Richard Lipton [158]. The birth of the field can also be identified in other papers published in the late 1970s by DeMillo et al. [65] and Hamlet [107].

Mutation Testing is capable of testing software at the unit level, the integration level, and the specification level. It has been applied to many programming languages as a white box unit test technique. For example, Fortran programs [3], [35], [39], [133], [145], [183], Ada programs [28], [194], C programs [6], [55], [98], [214], [215], [236], [238], Java programs [43], [44], [129]–[132], [150], [151], C# programs [68]–[72], SQL code [42], [213], [232], [233] and AspectJ programs [11], [12], [16], [91]. Mutation Testing has also been used for integration testing [53]–[55], [57]. 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 [19], [27], [89], [112], State charts [96], [231], [255], Estell Specifications [223], [224], Petri Nets [87], Network protocols [126], [204], [217], [237], Security Policies [140], [154], [167], [168], [203] and Web Services [141], [143], [144], [195], [244], [254].

King's College London
Centre for Research on Evolution, Search and Testing (CREST)
Strand, London
WC2R 2LS, UK
{yue.jia,mark.harman}@kcl.ac.uk

Mutation Testing has been increasingly widely studied since it was first proposed in the 1970s. There has been much research work on the various kinds of techniques seeking to turn Mutation Testing into a practical testing approach. However, there is little survey work in the literature on Mutation Testing. The first survey work was conducted by DeMillo [61] in 1989. This work summarized the background and research achievements of Mutation Testing at this early stage of development of the field. A survey review of the (very specific) sub area of Strong, Weak, and Firm mutation techniques was presented by Woodward [249]. Two introductory chapters on Mutation Testing can be found in the book by Mathur [155] and the book by Ammann and Offutt [10]. The most recent survey work was conducted by Offutt and Untch [193] in 2000. They summarized the history of Mutation Testing and an overview of the existing optimization techniques for Mutation Testing. However, since then, there have been more than 170 publications on Mutation Testing, but there has been no further survey work.

In order to provide a complete survey covering all the publications related to Mutation Testing since 1970s, we constructed a Mutation Testing publication repository, which includes more than 350 papers from 1977 to 2009 [122]. We also searched for Master and PhD theses that have made a significant contribution to the development of Mutation Testing. These are listed in Table I. We took four steps to built this repository. We first searched from 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 its title or abstract. Then we went through the references for each paper in our repository, to find missing papers using the same keywords rules effectively performing a 'transitive closure'. Mutation Testing work which was not concerned with software, for example, hardware mutation, and papers not written in English were filtered out. Finally we sent a draft of this paper to all cited authors using them to check our citations. We have made the repository publicly available at http://www.dcs.kcl.ac.uk/pg/jiayue/repository/ [122].

A. Global Temporal Trends

To understand the general trend for the Mutation Testing research area, we analysed the number of publications by year from 1977 to 2008 (the publication data in 2009 is not complete). This results are depicted in Figure 1. In Figure 1, there are

TABLE I
A LIST OF PHD AND MASTER WORK ON MUTATION TESTING

Author	Title	Type	University	Year
Acree [2]	On Mutation	PhD	Georgia Institute of Technology	1980
Hanks [108]	Testing Cobol Programs by Mutation	PhD	Georgia Institute of Technology	1980
Budd [33]	Mutation Analysis of Program Test Data	PhD	Yale University	1980
Tanaka [229]	Equivalence Testing for Fortran Mutation System Using Data Flow Analysis	PhD	Georgia Institute of Technology	1981
Morell [166]	A Theory of Error-Based Testing	PhD	University of Maryland at College Park	1984
Offutt [196]	Automatic Test Data Generation	PhD	Georgia Institute of Technology	1988
Craft [46]	Detecting Equivalent Mutants Using Compiler Optimization Techniques	Master	Georgia Institute of Technology	1989
Krauser [134]	Compiler-Integrated Software Testing	PhD	Purdue University	1991
Fichter [92]	Parallelizing Mutation on a Hypercube	Master	Clemson University	1991
Lee [142]	Weak vs. Strong: An Empirical Comparison of Mutation Variants	Master	Clemson University	1991
Zapf [256]	A Distributed Interpreter for the Mothra Mutation Testing System	PhD	Clemson University	1993
Delamaro [51]	Proteum - A Mutation Analysis Based Testing Environment	PhD	University of São Paulo	1993
Wong [246]	On Mutation and Data Flow	PhD	Purdue University	1993
Pan [199]	Using Constraints to Detect Equivalent Mutants	Master	George Mason University	1994
Untch [235]	Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method	PhD	Clemson University	1995
Ghosh [99]	Testing Component-Based Distributed Applications	PhD	Purdue University	2000
Ding [73]	Using Mutation to Generate Tests from Specifications	Master	George Mason University	2000
Okun [197]	Specification Mutation for Test Generation and Analysis	PhD	University of Maryland Baltimore	2004
Ma [148]	Object-oriented Mutation Testing for Java	PhD	KAIST University in Korea	2005
May [162]	Test Data Generation: Two Evolutionary Approaches to Mutation Testing	PhD	University of Kent	2007
Hussain [117]	Mutation Clustering	Master	King's College London	2008
Adamopoulos [4]	Search Based Test Selection and Tailored Mutation	Master	King's College London	2009

three apparent outliers in years 2001, 2006 and 2007. The reason for this, is that there were three Mutation Testing workshops held in 2000 (with proceedings published in 2001), 2006 and 2007. The reader will also notice that 1986 is unique; there were no publications found. An interesting anecdote was provided by Offutt [179]: "1986 was when we were maximally devoted to programming Mothra. I spent 70+ hour per week banging my head against vi, gcc, and dbx."

We performed a regression analysis on these data, and found there is a strong positive correlation between year and the number of publications (r = 0.8005). In order to 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 1 is the best fit line we found. It uses a quadratic model, which achieves the highest coefficient of determination (R^2 = 0.723). From this analysis is clear that Mutation Testing remains an active research area with growing interest.

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 on the hypotheses supporting Mutation Testing, optimization techniques, such as 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 2 shows the overall cumulative result. It is clear to see that both theatrical and practical work is increasing by year, and in 2006, the total number of practical publications surpasses the number of theoretical publications for the first time. To take a closer look at this relationship, Figure 3 shows the number of publications by year. From 1977 to 2000, there were fewer practical publications than theoretical. From 2000 to 2008, 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.

The rest of the paper is organized as follows. Section II introduces the fundamental theory of Mutation Testing including the hypotheses, the process and the problems of Mutation Testing. Section III explains the techniques for the reduction of the computational cost. Section IV introduces the techniques for detecting equivalent mutants and the applications of Mutation Testing are introduced in Section V. Section VI summarised the empirical experiments of the research work on Mutation Testing. Section VII describes the development work on mutation tools. Section VIII discusses the future trend of Mutation Testing and the paper will conclude in Section IX.

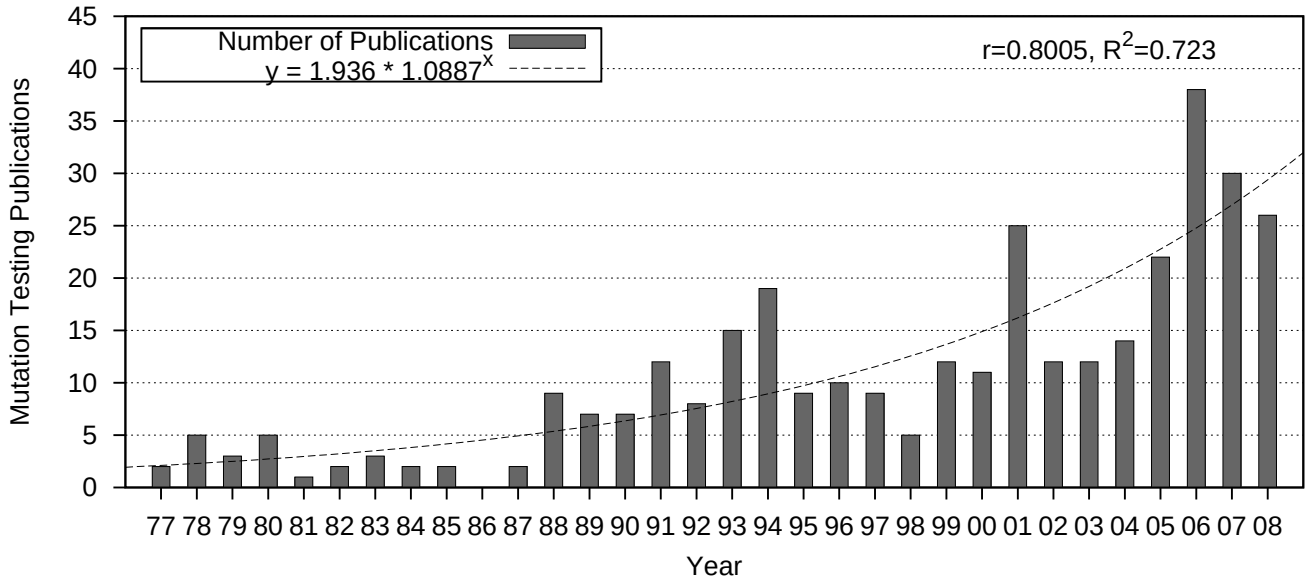

Fig. 1. Mutation Testing Publication from 1978-2008

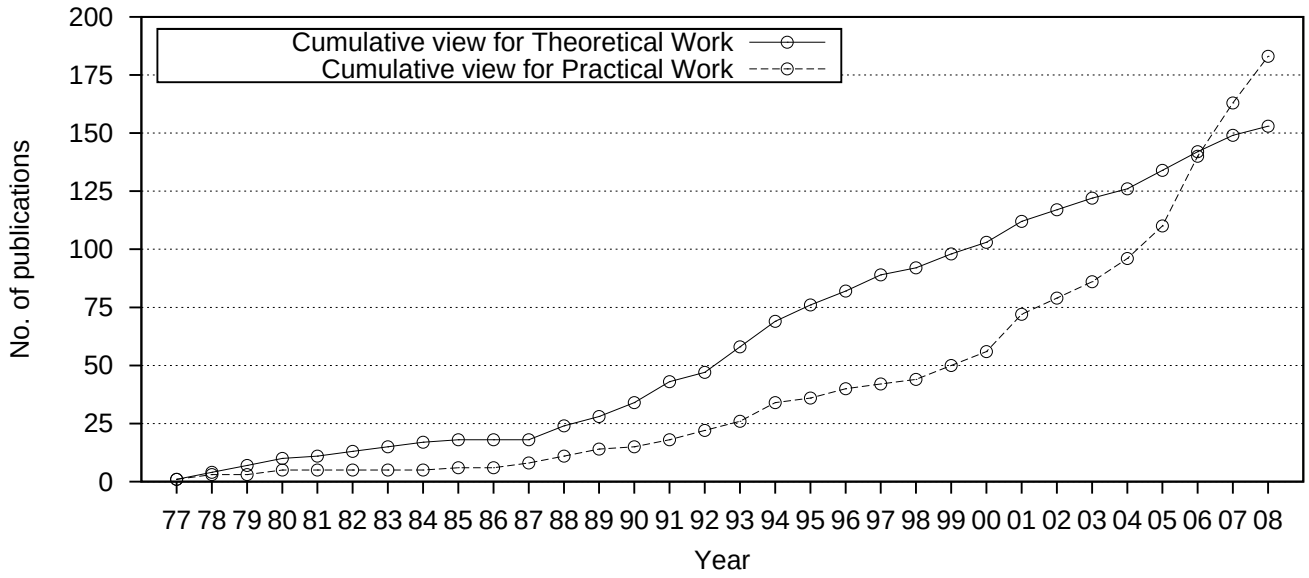

Fig. 2. Theoretical Publications VS. Practical Publications (Cumulative view)

II. The Theory of Mutation Testing

A. Fundamental Hypotheses

Mutation Testing promises to be effective in identifying adequate test data which can be used to find real faults [97]. 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 only targets 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) [3], [65] and Coupling Effect [65].

The CPH was first introduced by DeMillo et al. in 1978 [65]. 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 represents the faults that are made by "competent programmers". An example of the CPH can be found in Acree et al.'s work [3] and a theoretical discussion using the concept of program neighbourhoods can also be found in Budd et al.'s work [36].

The Coupling Effect was also proposed by DeMillo et al. in 1978 [65]. Unlike the CPH concerning 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 [177], [178] extended this into the Coupling Effect Hypothesis and the Mutation Coupling Effect Hypothesis with a precise

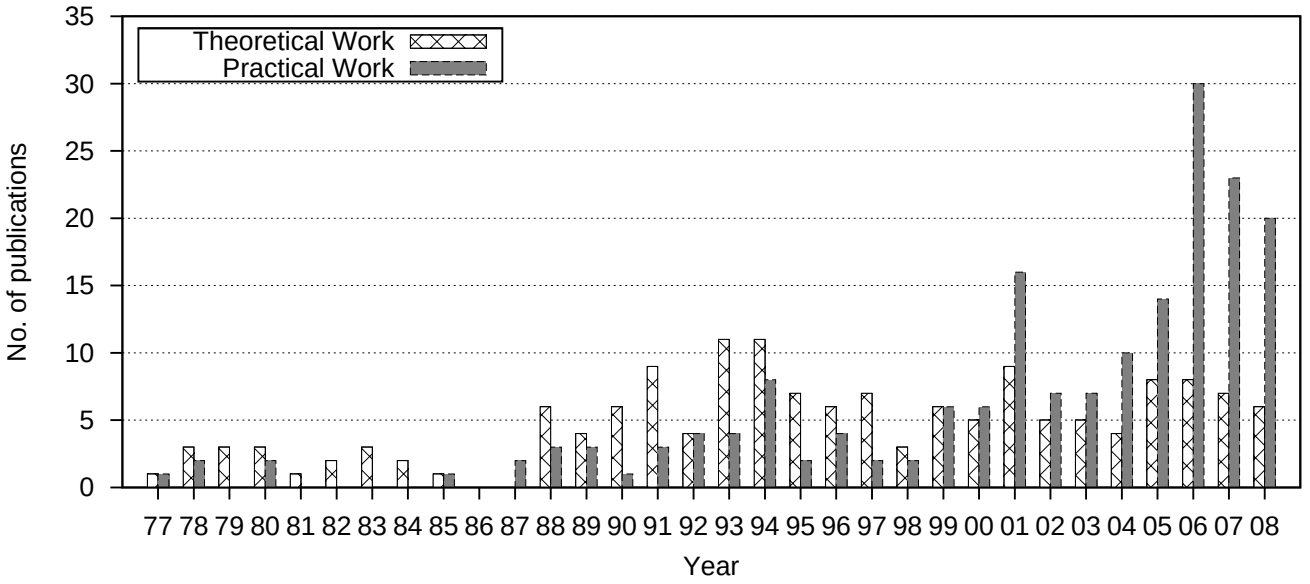

Fig. 3. Theoretical Publications VS. Practical Publications

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 " [178]. 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" [178]. 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 [145], [166], [177], [178]. Lipton and Sayward [145] conducted an empirical study using a small program, FIND. In their experiment, a small sample of 2nd-order, 3rd-order and 4th-order mutants are 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 [177], [178] 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 is valid also agreeing with the empirical study by Morell [166].

The validity of the mutation coupling effect has also been considered in the theoretical studies of Wah [241]–[243] and Kappoor [127]. In Wah's work [242], [243], 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 are $1/n$ and $1/n^2$ respectively where n is the order of the domain [242]. This result is also similar to the estimated results of the empirical studies mentioned above. A formal proof coupling effect on the boolean logic faults can be also found at Kappoor's work [127].

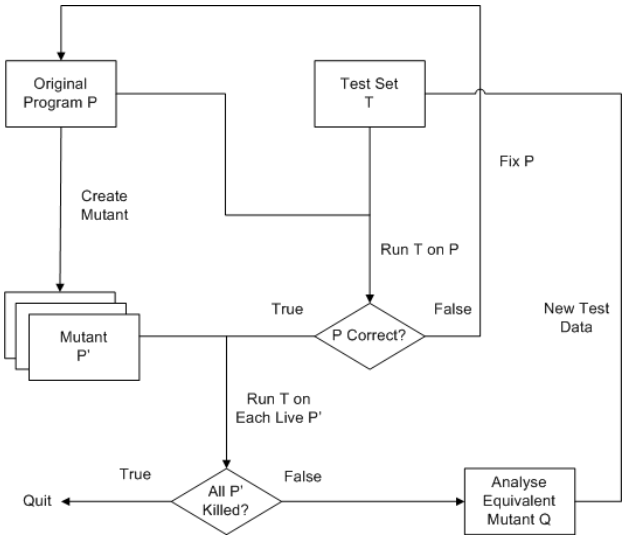

Fig. 4. Generic Process of Mutation Analysis (adapted from [193])

B. The Process of Mutation Analysis

The traditional process of mutation analysis is illustrated in Figure 4. In mutation analysis, from a program p, a set of faulty programs p' called mutants, is generated by a few single syntactic changes into the original program p. As an illustration, Table II shows the mutant p', generated by changing the and operator (&&) of the original program p, into the or operator (||), thereby producing of the mutant p'.

A transformation rule that generates a mutant from the original program is known as a mutation operator[1]. Table II contains only one example of a mutation operator; there are many others.

[1]In the literature of Mutation Testing, mutation operators are also known as mutant operators, mutagenic operators, mutagens and mutation rules [193].

TABLE II
A EXAMPLE OF MUTATION OPERATION

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

TABLE III
THE FIRST SET OF MUTATION OPERATORS: THE 22 "MOTHRA" FORTRAN MUTATION OPERATORS (ADAPTED FROM [133])

Mutation Operator	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

Typical mutation operators are designed to modify variables and expressions by replacement, insertion or deletion operators. Table III lists the first set of formalized mutation operators for the Fortran programming language [184]. To increase the flexibility of Mutation Testing in practical applications, Jia and Harman introduced a scripting language, the Mutation Operator Constraint Script (MOCS) [124]. The MOCS provides two types of constraint: Direct Substitution Constraint and Environmental Condition Constraint. The Direct Substitution Constraint allows users to select a specific transformation rule that performs a simple change while the Environmental Condition Constraint is used to specify the domain for applying mutation operators. Simao et al. [218] also proposed a transformation language, MUDEL, used to specify the description of mutation operators.

In 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 be still 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. These mutants are called Equivalent Mutants. They are syntactically different but functionally equivalent to the original program. Automatically detecting equivalent mutants is impossible [34], [189], 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 IV.

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.

C. The Problems of Mutation Analysis

Although Mutation Testing is able to effectively assess the quality of a test set, it still suffers from a number of problems. One problem that prevents Mutation Testing from becoming a practical testing technique is the high computational cost of executing the enormous number of mutants against a test set. The other problems are related to the amount of human effort involved in using Mutation Testing, for example, the human oracle problem and the equivalent mutant problem.

The human oracle problem 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 of set inputs has been arrived at, there remains the problem of checking output. However, mutating testing is effective precisely because it is demanding and this can lead to an increase in the number of test cases, thereby increasing oracle cost. This oracle cost is often the most expensive part of the overall test activity. Also, because of the undecidability of mutant equivalence, the detection of equivalent mutants often involves additional human effort.

Although it is impossible to completely solve these problems, with existing advances in Mutation Testing, the process of Mutation Testing can be automated and the run time can allow for reasonable scalability, as this survey will show. A lot of previous work has focused on techniques to reduce computational cost, a topic to which we now turn.

III. COST REDUCTION TECHNIQUES

Mutation Testing is widely behind to 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 [193], 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 combines do fewer and do faster) and reduction of the execution cost (which corresponds to do faster). Figure 5 provides an overview of the chronological development of published ideas for cost reduction.

To take a closer look at the cost reduction research work, we counted the number of publications for each technique (see Figure

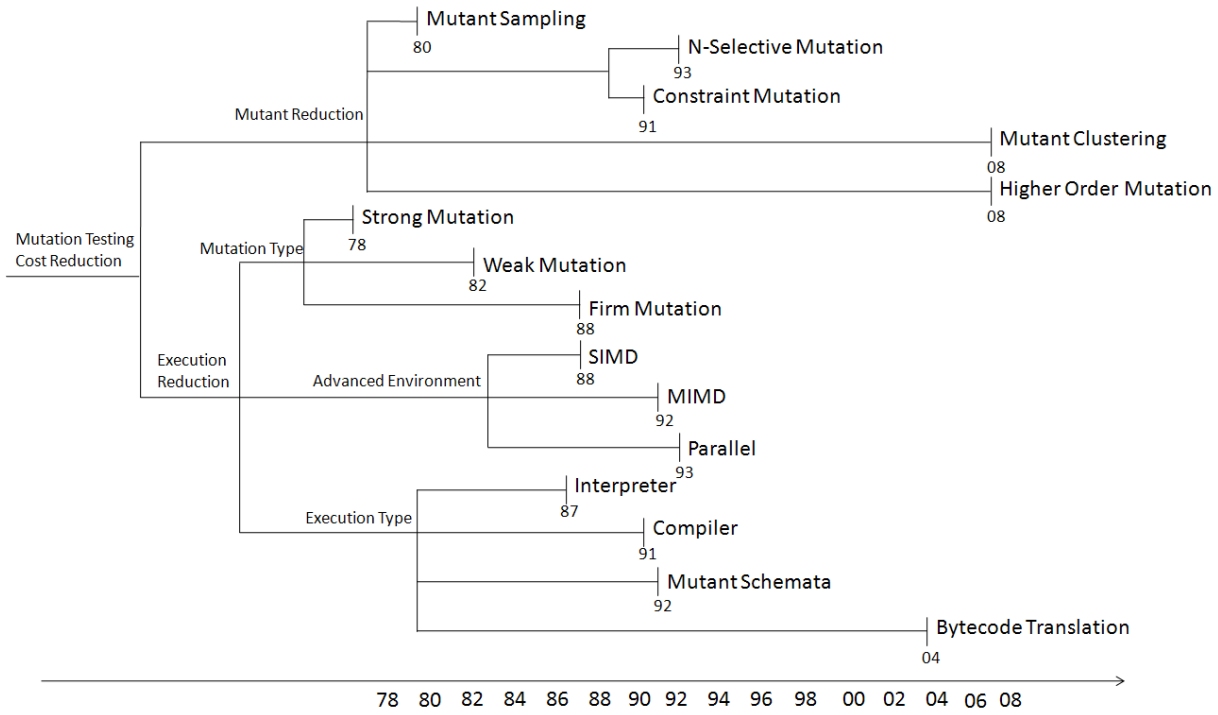

Fig. 5. Overview of the Chronological Development of Mutant Reduction Techniques

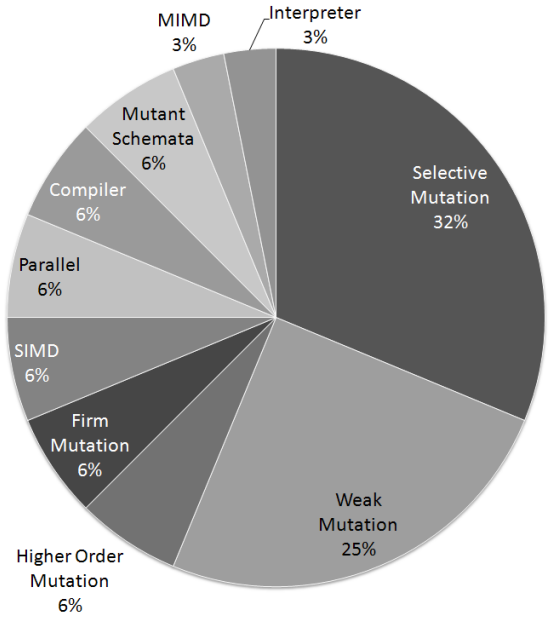

Fig. 6. Percentage of publications using each Mutant Reduction Technique

6). 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 5 papers, to date. The rest of the section will introduce each cost reduction technique in detail. Section III-A will presents work on mutant reduction techniques, while Section III-B will covers execution reduction techniques.

A. *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 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, Mutant Clustering, Selective Mutation and Higher order Mutation.

1) 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 [2] and Budd [33]. 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 [160], [246], 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. [63] and King and Offutt [133]. Instead of fixing the sample rate, Sahinoglu and Spafford [209] 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.

2) Mutant Clustering: The idea of the Mutant clustering was first proposed in Hussain's masters thesis [117]. 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. [121]. Ji et al. use a domain reduction technique to avoid the need to execute all mutants.

3) 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 [156]. Offutt et al. [192] 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 40% to 60% of all mutants [133]. To effectively reduce the generated mutants, Mathur [156] 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. [192].

Offutt et al. [192] 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 adopt another type of selection strategy, selection based on test effectiveness [246], [248], known as constraint mutation. Wong and Mathur suggested using only two mutation operators: ABS and RAR. The reason to choose the ABS operator is that killing the mutants generated from ABS requires the test cases which select from different parts of the input domain related to the mutated expression [246], [248]. The reason to choose the ROR operator is that killing the mutants generated from ROR requires the test cases which 'examine' the mutated predicate [246], [248]. 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. [184] 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 [169] 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. [18] 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 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. [170]–[172]. 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 achieves the highest rate of reduction compared with other approaches.

4) Higher Order Mutation: Higher Order Mutation is a comparatively new form of Mutation Testing introduced by Jia and Harman [123]. The underlying motivation was to seek to find those rare but valuable higher order mutants that denote subtle faults. In traditional Mutation Testing, mutants can be classified into first order mutants (FOM) and higher order mutant(HOM). FOMs are created by applying mutation operator only once. HOM are generated by applying mutation operators more than once.

In their work, Jia and Harman introduced the concept of subsuming HOMs. A subsuming HOM is harder to kill than the FOMs from which it is constructed. As a result, it may be preferable to replace FOMs with the single HOM to reduce the number of the mutants. In particular, they also introduced the concept of a strongly subsuming HOM (SSHOM) which is only killed by a subset of the intersection of test cases that kill each FOM from which it is constructed.

Figure 7 illustrates an simple example of using SSHOM to reduce test effort and to increase test effectiveness at the same time. Suppose there is a SSHOM h which is constructed from FOMs f_a and f_b. The two regions T_a and T_b in Figure 7 represent the test sets containing all the test cases that kill FOMs f_a and f_b respectively, while the region T_h represents the test set containing all test cases that kill SSHOM h. In traditional mutation testing, it is not hard to find test cases like t_a and t_b which kill both FOM f_a and f_b. However the test case t_h that kills SSHOM h is a better choice, because it kills FOM 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

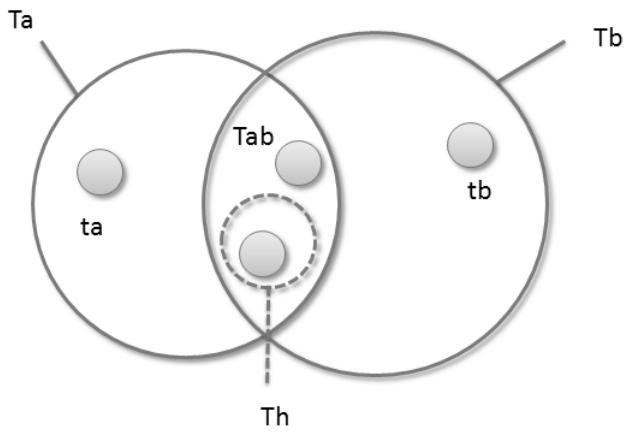

Fig. 7. Test Effort Reduction Example (Taken from [125])

test case selected from this intersection can achieve the same test effort as the test cases that kill the SSHOM h, such a test case like t_{ab} might not able to find the subtle fault represented by SSHOM h, thereby losing test effectiveness.

This idea has been partly proved by Polo et al.'s work [201]. In their experiment, they focused on a specific order of HOMs, the second order mutants. They proposed different algorithms to combine first order mutants to generate the second order ones. Empirical results suggests that applying the second order mutants reduced effort was reduced by approximately 50%, without much loss of test effectiveness. More recently, Langdon et al. have applied multi-object genetic programming to the generation of higher order mutants [137], [138]. In their experiment, they have found realistic higher order mutants that are harder to kill than any first order mutant.

B. Execution Cost Reduction Techniques

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

1) Strong, Weak, and Firm Mutation: Based on the way in which we decide whether to analysis 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 as traditional Mutation Testing. That is, it is the formulation originally proposed by DeMillo et al. [65]. 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 optimize the execution of the Strong Mutation, Howden [116] 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 only need to be checked immediately after the execution point of the mutant or mutated component.

In Howden's work [116], the component C referred to one of the following 5 types: variable reference, variable assignment, arithmetic expression, relational expression, and boolean expression. This definition of component was later refined by Offutt and Lee [185], [186]. Offutt and Lee defined four types of execution: evaluation after the first execution of (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 [249]. 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 set. In this way weak mutation scarifies test effectiveness for improvements in test effort. This raises the question as to what kind of sacrificial trade off can be achieved.

There were many empirical studies on the Weak Mutation trade off. Girgis and Woodward [104] implemented a weak mutation system for Fortran 77 programs. Their system as a analytical type weak mutation system which the mutants are killed by examining the program's internal state. In their experiment, four of Howden's five program components were considered, and the results suggested that weak mutation is less computationally expensive than the strong mutation which is also agree by Marick's work [153].

A theoretical proof of Weak Mutation by Horgan and Mathur [114] shown that under certain conditions, test set generated by weak mutation can also be expected by strong mutation. Offutt and Lee [185], [186] 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 5 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 [114] and experimental results of Girgis and Woodward [104] and Marick [153].

Firm Mutation was first proposed by Woodward and Halewood [252]. 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 [120] proposed a parallel Firm Mutation approach for Java programs. Unfortunately, there is to date no publicly available firm mutation tool.

2) Runtime Optimization Techniques: The interpreter-based Technique is one of the optimization techniques used in first generation of mutation testing tools [183]. In 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. 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.

To reduce the cost of interpretation, the compiler-based technique was proposed [51], [52], because execution of compiled binary code is much faster than interpretation. In compiler-based

techniques, the mutant source is compiled into an executable program first, then each compiled mutant will be executed by a number of test cases. As each mutant generated are The overall cost of this technique is the sum of the compilation cost and the running cost. There is also a speed limitation due to the high compilation cost for large programs [45].

DeMillo et al. proposed the compiler-integrated technique [64] 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, a 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 to executable code for a mutant directly. 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 [134].

The Mutant Schema Generation approach is also designed to reduce the overhead cost of the traditional interpreter-based techniques [234]–[236]. 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 runtime 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 [236] 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 [235]. 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 [157].

The most recent work on reduction of the compilation cost is the Bytecode Translation technique. This technique is another way to first proposed by Ma et al. [151], [187]. 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 [151], [152], [187], [210]. 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 [210].

Bogacki and Walter introduced an alternative approach to reduce compilation cost, called Aspect-Oriented Mutation [25], [26]. 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 [25].

TABLE IV
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 $(int\ i = 0;\ i\ !\ = 10; i{+}{+})$
	{
	$...(the\ value\ of\ i$
	$is\ not\ changed)$
	}

3) Advanced Platforms Support for Mutation Testing: Mutation Testing has also been applied to many novel computer architectures to distribute the overall computational cost among many processors. In 1988, Mathur and Krauser [159] were the first to perform Mutation Testing on a vector processor system. Krauser et al. [135], [136] proposed an approach for concurrent execution mutants under SIMD machines. Fleyshgakker and Weiss [93], [245] proposed an algorithm that significantly improved techniques for parallel Mutation Testing. Choi and Mathur [45] and Offutt et al. [191] have distributed the execution cost of Mutation Testing through MIMD machines. Zapf [256] extended this idea in a network environment, where each mutant is executed independently.

IV. EQUIVALENT MUTANT DETECTION TECHNIQUES

To detect if a program and one of its mutants programs are equivalent is an known undecidable problem, this has been proved in the work of Budd and Angluin [34]. As a result, the detection of equivalent mutants alternatively may have to be carried out by humans. 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 IV 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.

There are many reasons why a mutant may be equivalent. Grün et al. [106] manually investigated eight equivalent mutants generated from the JAXEN XPATH query engine program. They pointed out four main reasons that cause a mutant to be equivalent, which are the mutant is generated from unneeded code, the mutant improves speed, the mutant only alters the internal states, and the mutant cannot be triggered.

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 [180], [189]. Fortunately, there has been much research work on the detection of the equivalent mutants.

Baldwin and Sayward [17] proposed an approach that used compiler optimization techniques to detect equivalent mutants. This approach is based on the idea that the optimization procedure of source code will produce an equivalent program, so a mutant

might be detected as equivalent mutants by either optimizations or a "de-optimization process". Baldwin and Sayward [17] proposed six types of compiler optimization rule applied to the detection of equivalent mutants. These six were implemented and empirically studied by Offutt and Craft [180]. 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 [188], [189] 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 (45% among 11 subject programs) for most the programs. Their results suggest that the constraint satisfaction formulation is more powerful than the compiler optimization technique [180].

The Program slicing technique has also been proposed to assist in the detection of equivalent mutants [110], [111], [240]. Voas and McGraw [240] were the first to suggest the application of program slicing to Mutation Testing. Hierons et al. [111] demonstrated an approach using slicing to assist the human analysis of equivalent mutants. This is achieved by generation of a simplest program denotes the answer to equivalent mutant. This work was later extended by Harman et al. [110] 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 a 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 detection of mutants are selected.

Ellims et al. [84] 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 this might be used to kill the potential mutants.

The most recent work on the equivalent mutants was conducted by Grün et al. [106] who investigated the impact of mutants. The impact of a mutant was defined as the different program behavior 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.

V. The Application of Mutation Testing

Since Mutation Testing was proposed in the 1970s, it has been applied to test both program source code (Program Mutation) [59] and program specification (Specification Mutation) [105]. 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.

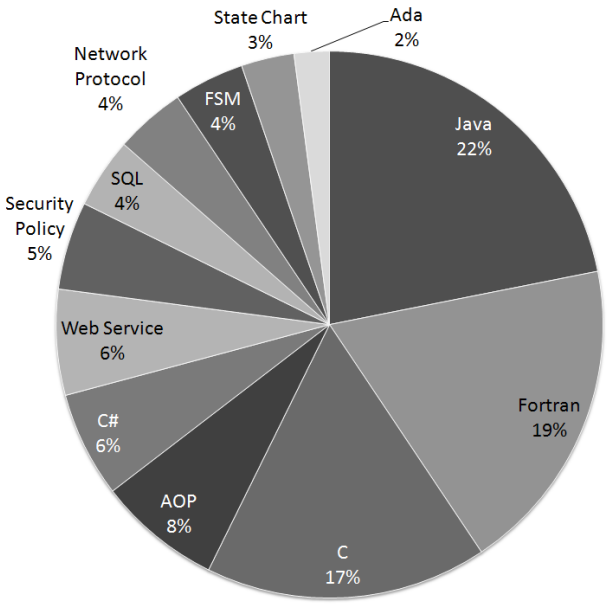

Fig. 9. Percentage of Publications addressing each language to which Mutation Testing has been applied

Figure 8 shows the chronological development of research work on Program Mutation and Specification Mutation. Figure 9 shows the percentage of the publications addressing each language to which Mutation Testing has been applied. As the figure 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 languages. Fortran features highly because a lot of the earlier work on Mutation Testingwas carried out on Fortran programs. In the following Section, the applications of Program Mutation and Specification Mutation are summarized by the programming language targeted.

A. Program Mutation

Program Mutation has been applied to both the unit level [65] and the integration level [54] 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 [239]. Applying Program Mutation at the integration level is also known as Interface Mutation which was first introduced by Delamaro et al. [54] in 1996. Interface Mutation has been applied to C Programs by Delamaro et al. [53]–[55], and also to the CORBA Programs by Ghosh and Mathur [99], [101]–[103]. Empirical evaluations of Interface Mutation can be found at Vincenzi et al.'s work [239] and Delamaro et al.'s work [56], [57].

1) Mutation Testing for Fortran: In the earliest days of Mutation Testing, most of the experiments on Mutation Testing targeted Fortran. Budd et al. [35], [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 [3], [35], [145]. However, there were no formal definitions of mutation operators for Fortran until 1987. In 1987, Offutt and King [133], [183] summarized the results from

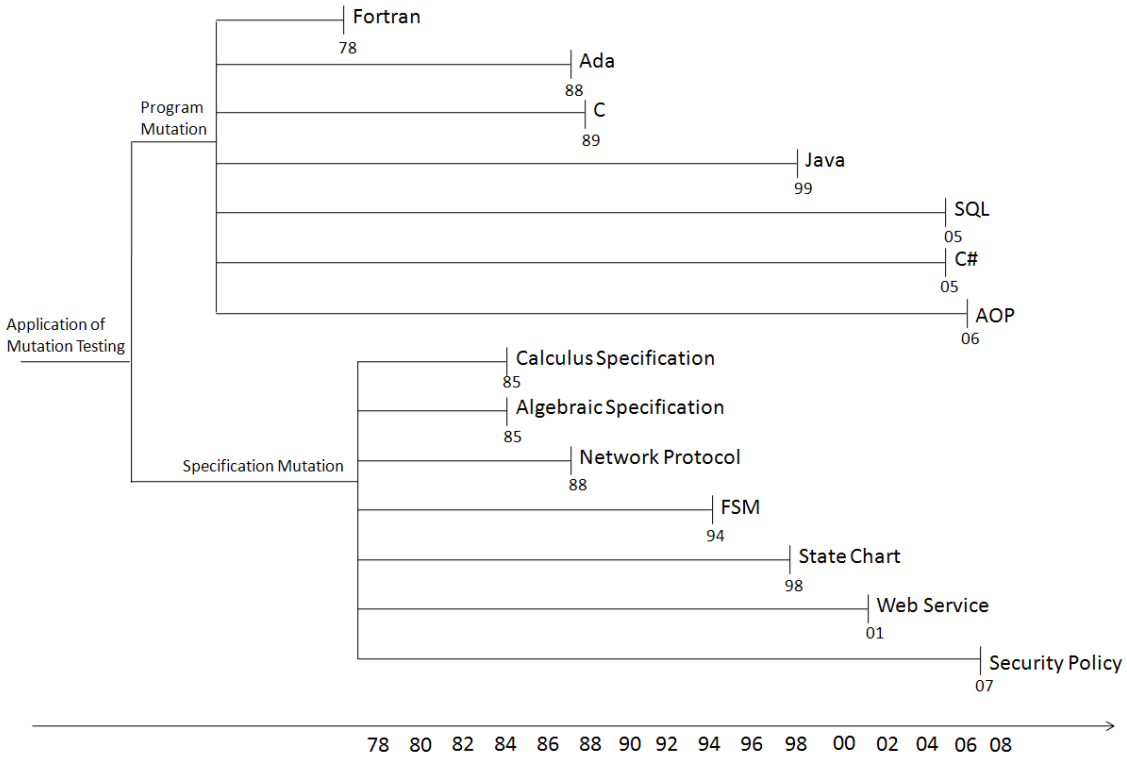

Fig. 8. Publications of the Applications of Mutation Testing

previous work and proposed 22 mutation operators for Fortran 77. This set of mutation operators became the first set of formalized mutation operators and had consequently great influence to later definitions of mutation operators for applying Mutation Testing to the other programming languages. These mutation operators are divided into three groups, which are Statement analysis group, Predicate analysis group, and Coincidental correctness group.

2) Mutation Testing for Ada: Ada mutation operators were first proposed by Bowser [28] in 1988. In 1997, based on previous work of Bowser's Ada mutation operators [28], Agrawal et al.'s C mutation operators [6] and the design of Fortran 77 mutation operators for MOTHRA [133], Offutt et al. [194] 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 are divided into five groups: Operand Replacement Operators group, Statement Operators group, Expression Operators group, Coverage Operators group and Tasking Operators group.

3) 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. [53]–[55], [57] investigated the application of Mutation Testing at the integration level. They selected 10 mutation operators from Agrawa 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 Mutation Testing has also been applied to C Programs by Jia and Harman [123].

There are also mutation operators that target specific C program defects or vulnerabilities. Shahriar and Zulkernine [215] proposed 8 mutation operators to generate the mutants that can be effectively represent Format String Bugs (FSBs). Vilela et al. [238] proposed 2 mutation operators representing faults associated with static and dynamic memory allocations, which was used to defect Buffer Overflows (BOFs). This work was subsequently extended by Shahriar and Zulkernine [214] 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. [98] have applied mutation testing to an Adaptive Vulnerability Analysis (AVA) to detect BOFs.

4) Mutation Testing for Java: Traditional mutation operators are not sufficient for testing Object Oriented (OO) programming languages like Java [132], [151]. 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. [130] 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. [129] 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. [131] 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 [132]. A similar approach was also adopted by Chevalley and Thevenod-Fosse in their work [43], [44].

Ma et al. [150], [151] pointed out that the design of mutation operators should not start with the selected approach (Kim et al.'s approach [129]), they suggested that the selected mutation operators should be obtained from empirical results on the effectiveness of all mutation operators. Therefore, instead of continuing Kim et al.'s work [131], Ma et al. [150] 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 [149]. The results suggested that some class mutation model faults can be detected by traditional mutation testing, however the mutants generated by the EOA and EOC class mutation 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. [9], [23] 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. [30] extended it introduced an extension to the concurrent Java environment.

5) Mutation Testing for C#: Based on previous proposed Java mutation operators, Derezińska introduced an extension to a set of C# specialized mutation operators [69], [70] and implemented them in a C# mutation tool named CREAM [71]. Empirical results for this set of C# mutation operators using the CREAM were reported by Derezińska and Szustek [70], [72].

6) 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. [42] in 2005. They proposed 7 SQL mutation operators based on the enhanced entity-relationship model. Tuya et al. [233] proposed another set of mutant operators for SQL query statements. This set of mutation operators are 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 [232]. A development of this work targeting Java database applications can be found in the work of Zhou and Frankl [259]. Shahriar and Zulkernine [213] have also proposed a set of mutation operators to handle 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.

7) 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. [91] proposed 26 mutation operators based on a generalization 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 [91]. 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 [50].

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 [12]. Baekken and Alexander [16] summarised previous research work on the fault model associated with AspectJ pointcuts, and proposed a complete AspectJ fault model based on the incorrect pointcut pattern, which were used as a set of mutation operators for AspectJ programs. Based on this work, Anbalagan and Xie [11], [12] proposed a framework to generate mutants for pointcuts and to detect equivalent mutants. To reduce the total number of the mutants, a classification and ranking approach based on the strength of the pointcuts was also introduced in their framework.

Besides these programming languages, Mutation Testing has also been applied to Lustre programs [81], [82], PHP programs [216], Cobol program [108], Matlab/Simulink [257] and spreadsheets [1]. There are also research work investigating the design of mutation operators for real-time systems [97], [173], [174], [228] and concurrent programs [8], [30], [40], [100], [147].

B. Specification Mutation

Although Mutation Testing was originally proposed as a white box testing technique at the implementation level, it has also been applied at software design level. Mutation Testing at design level is often referred as 'Specification Mutation', which was first introduced by Gopal and Budd in 1983 [37], [105]. 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 [197].

1) Mutation Testing for Formal Specifications: The formal specifications can be presented in many forms, for example calculus expression, Finite State Machines (FSM), Petri Nets and Statecharts. The earlier research work on Specification Mutation considered specifications of simple logical expressions. Gopal and Budd [37], [105] considered of 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 [250], [252] 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 [251].

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. [89] 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 [86]. An empirical evaluation of these mutation operators was reported by them [86]. Hierons and Merayo [112], [113] 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. [19], Bombieri et al. [27] and Belli et al. [22].

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. [88], based on the their previous work on FSM mutation operators. Using Fabbri et al.'s Statecharts mutation operators, Yoon et al. [255] introduced a new test criterion, State-based Mutation Test Criterion (SMTC). In the work of Trakhtenbrot [231], the author proposed new mutations to assess the quality of tests for statecharts at the implantation level as well as the model level. Other work on Statechart mutation can be found in the work of Fraser et al. [96].

Besides FSMs and Statecharts, Specification Mutation has been also applied to a variety of specification languages. For example, Souza et al. [223], [224] investigated the application of Mutation Testing to the Estell Specification language. Fabbri et al. [87] proposed mutation operators for Petri Nets. Srivatanakul et al. [226] performed an empirical study using Specification Mutation to CSP Specifications. Olsson and Runeson [198] and Sugeta et al. [227] proposed mutation operators for SDL. Definitions of mutation operators for formal specification language can be found in the work of Black et al. [24] and the work of Okun [197].

2) Mutation Testing for Running Environment: During the process of implementing specifications, bugs might be introduced by programmers because insufficient knowledge of the final target environment. These bugs are called 'environment bugs' and they can be hard to detect. For example, the bugs caused by memory limitations, numeric limitations, value initialization, constant value interpretation, exception handling and system errors [225]. Mutation Testing was first applied to the detection of such bugs by Spafford [225] 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" [79]. As a result, Mutation Testing was also applied to the validation of security vulnerabilities. Du and Mathur [79] 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 [80].

3) Mutation Testing for Web Services: Lee and Offutt [143] were the first to apply Mutation Testing to Web Services. In 2001, they introduced an Interaction Specification Model to formalize the interactions of between web components [143]. Based on this specification model, a set of generic mutation operators were proposed to mutate the XML data model. This work was later extend by Xu et al. [195], [254] targeting the mutation of XML data, and they renamed it to 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 [254]. Another set of XML schema mutation operators was proposed by Li and Miller [144].

There is also Web Service mutation work targeting specific XML-based language features, for example, the OWL-S specification language [141], [244] and WS-BPEL specification language [85]. 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. [141], the authors propose mutation operators for detection of semantic errors caused by the misuse of the ontology classes.

4) Mutation Testing for Networks: Protocol robustness is an important aspect of any network system. Sidhu and Leung [217] investigated fault coverage of network protocols. Based on this work, Probert and Guo proposed a set of mutation operators to test network protocols [204]. Vigna et al. [237] applied Mutation Testing to network-based intrusion detection signatures, which are used to identify malicious traffics. Jing et al. [126] 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. [258].

5) Mutation Testing for Security Policy: Mutation Testing has also been applied to security policies [140], [154], [167], [168], [203]. Much research work designed mutation operators to inject common flaws into different types of security policies. For example, Xie et al. [154] 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. [168]. Le Traon et al. [140] introduced 8 mutation operators for the Organization Based Access Control OrBAC policy. Mouelhi et al. [167] proposed a generic meta-model for security policy formalisms. Based on this formalism, a set of mutation operators are introduced to apply to all rule-based formalisms. Hwang et al. proposed an approach to apply Mutation Testing to test firewall policies [118].

C. Other Testing Application

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.

1) Test Data Generation: The main idea of mutation based test data generation is to generate test data that can effectively kill mutants; thereby such a generated test set will achieve a high mutation adequacy score. Constraint-based test data generation (CBT) is one of the automatic test data generation techniques using mutation testing. It was first proposed in Offutt's PhD work [196]. Offutt suggested that there are three conditions for a test case to kill a mutant: reachability, necessity and sufficiency. In CBT, each condition for a mutant is turned into a constraint. Test data that guarantees to kill this mutant can be generated by finding input values that satisfy these constraints.

Godzilla is a test data generator that uses the CBT technique. It was implemented by DeMillo and Offutt [66] under the

Mothra system. Godzilla applied control-flow analysis, symbolic evaluation and a constraint satisfaction techniques to generate and solve constraints for each mutant. Empirical results suggest that 90% of mutants can be kill using the CBT technique for most programs [67]. However, the CBT technique also suffers from some of the drawbacks associated with symbolic evaluation. Offutt et al. [181], [182] addressed these problems by proposing the Dynamic Domain Reduction technique.

Baudry et al. proposed an approach to automatically generate test data for components implemented by contract [21]. In this research work, a testing-for-trust methodology was introduced to keep the consistency of the three component artifacts: specification, implementation and test data. Baudry et al. applied a genetic algorithm to generate test data. The generated test data is then considered as a predator which is used to validate the program and the contract at the same time. Experimental results showed that 75% of mutants can be can be killed using this test data generation technique.

Besides generating test data directly, Mutation Testing has also been applied to improve the quality of test data. Baudry et al. [20] proposed an approach to improve the quality of test data using Mutation Testing with a Bacteriological Algorithm. Smith and Williams applied Mutation Testing as a guidance to test data augmentation [220]. Le Traon et al. [139] use mutation analysis to improve component contract. Xie et al. [253] applied Mutation Testing to assist programmers in writing parametrised unit tests.

2) Regression testing: Test case prioritization techniques are one way to assist regression testing. Mutation Testing have been applied as a test case prioritization technique by Do and Gregg [75], [76]. 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 prioritization can effectively improve the rate of fault detection of test suites [76].

Mutation Testing has also been used to assist the test case minimization process. Test case minimization techniques aim to reduce the size of a test set without losing much test effectiveness. Offutt et al. [176] 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 [211], [212]. Mutation analysis has also been used to assist the evaluation of software clone detection tools [206], [207].

VI. Empirical Evaluation

Empirical study is an important aspect in the evaluation and up take of any technique. In the following sections, the subject programs used in empirical studies are first summarised. Empirical results on the evaluation of Mutation Testing are then reported in detail.

A. Subject Programs

In order to investigate the empirical studies on Mutation Testing, we have collected all the subject programs for each empirical experiment work from our repository, as shown in Table IX. Table IX 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 IX 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. It is clear to see that 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 program, we calculated the size of the largest subject program for each year. A cumulative result is shown in Figure 10. 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 It is clear that the size of the subject programs that can be handle by Mutation Testing is increasing. However, caution is required. We found that although some empirical experiments were reported to handle large programs, some studies only applied a few mutation operators. We also counted the number of newly introduced subject programs for each year. The result are shown in Figure 11. The dashed line in the figure is the cumulative view of the results. The number of newly used subject programs is in gradually increasing, which suggest a growth in practical work as well.

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 12. In this study, we consider a real world program to be either an open source or an industry program. In Figure 12, 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 12 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 laboratoryprograms overall. Since 2002, far more new real programs than laboratoryprograms 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 benchmark. In Table V we have summarised these benchmark programs, presented by each research area. 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.

B. Empirical Results

Many researchers have conducted experiment to evaluates the effectiveness of Mutation Testing [13], [48], [60], [94], [95],

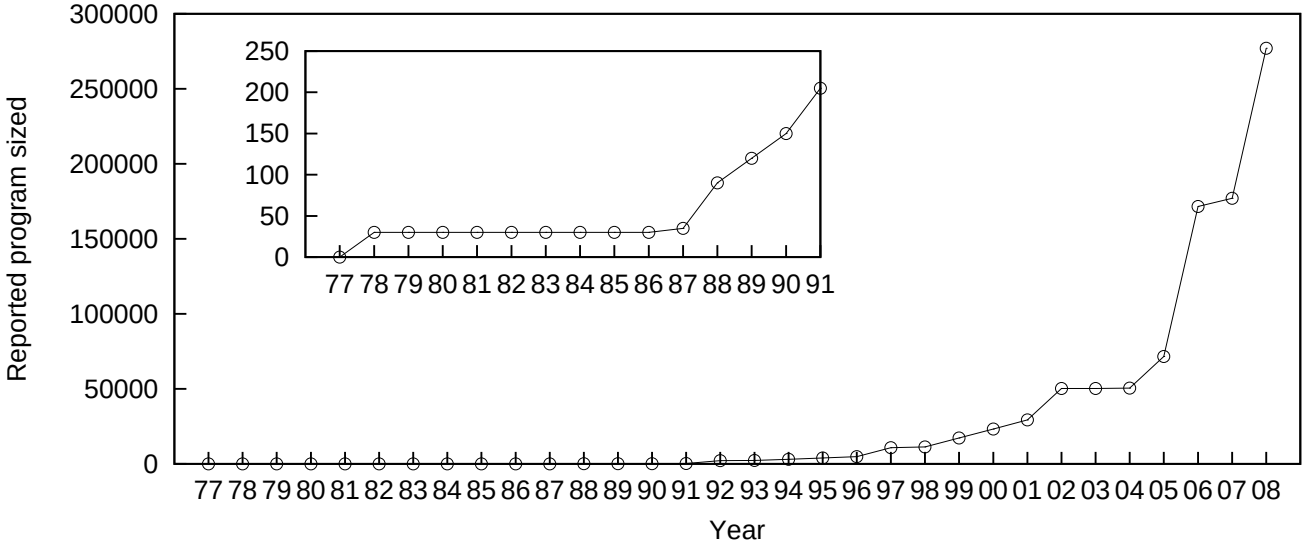

Fig. 10. The largest program applied for each year

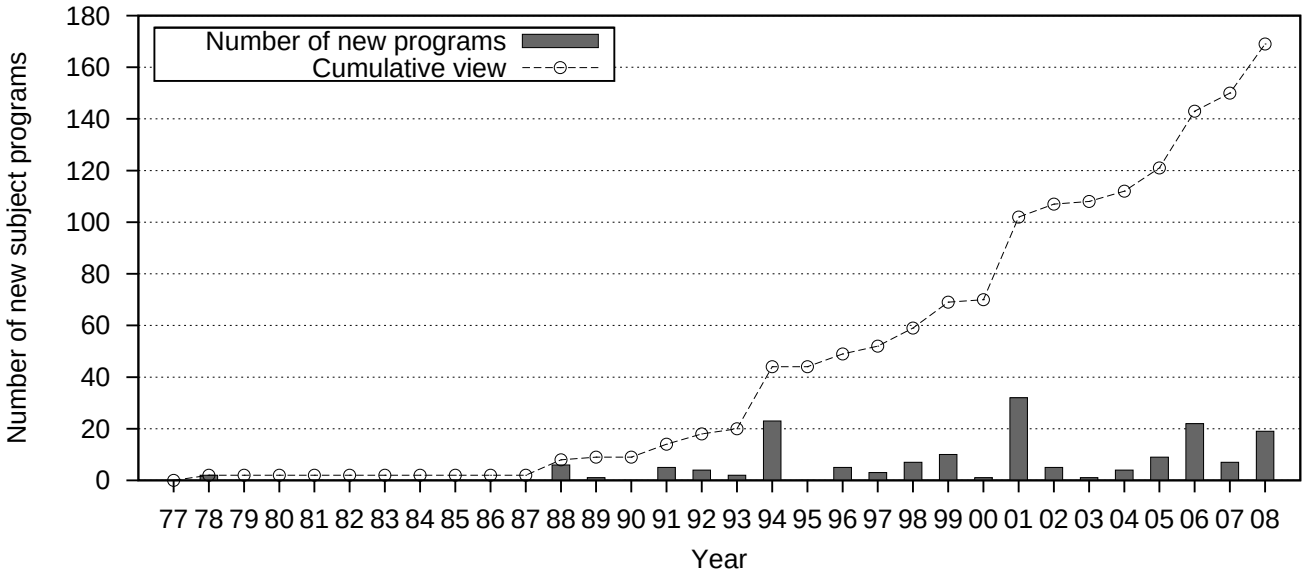

Fig. 11. New programs applied for each year

[161], [190], [246]. 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 VI summarises the evaluation type and the subject programs used in each of these experiments.

Mathur and Wong have conduced experiments to compare the 'all-use' criteria with mutation criteria [161], [246]. 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' the 'all-use' criteria in general.

Offutt et al. conduced a similar experiment using ten different programs [190]. The 'cross scoring' result also provides evidence for Mathur and Wong's probsubsumes relationship [161], [246]. In addition to comparing the two criteria with each other, Offutt et al. also compared the two criteria in terms of 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' than the 'all-use' data flow. This conclusion also agreed with the results of the experiment of Frankl et al. [94], [95]

In addition to comparing mutation analysis with other testing criteria, there have also empirical studies comparing real faults and mutants. In the work of Daran and Thévenod-Fosse [48], 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 is also agreed with DeMillo and Mathur's experiment [60]. DeMillo and Mathur carried out an extensive study of the errors in TeX reported by Knuth [60], and they demonstrated how simple mutants could

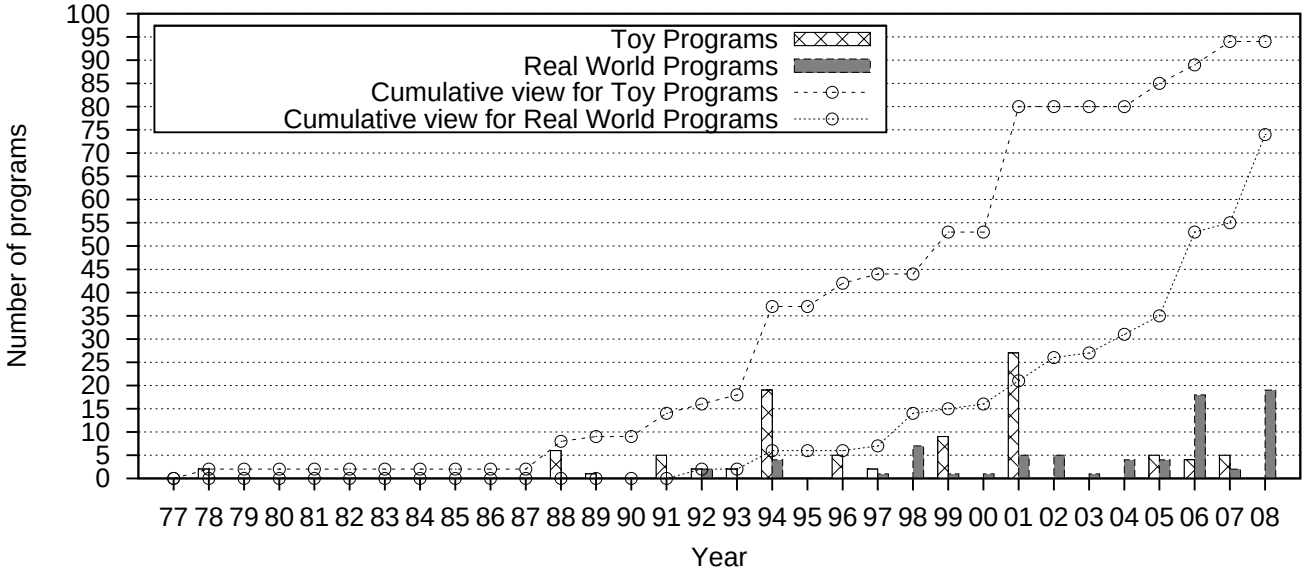

Fig. 12. Laboratory programs VS. Real Programs

TABLE V
SUBJECT PROGRAMS BY APPLICATION

Application	Subject Programs	Reference
Coupling Effect	Triangle, Find, MID	[177], [178]
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	[18], [169], [170], [172], [184], [192]
Weak, Strong, Firm Mutation	Triangle, Find, Bubble, MID, Calendar, Euclid, Quad, Insert, Warshall, Pat, Gcd, Sort, Max_index	[185], [186], [252]
Equivalent Mutant	Triangle, Find, Bubble, MID, Calendar, Euclid, Quad, Insert, Warshall, Pat, Bsearch, Max, Banker, Deadlock, Count, Dead	[180], [188], [189]
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, Xml-Security, Jmeter, JTopas, ATM, BOOK, VirtualMeeting, MinMax, NextDate, Finance	[15], [66], [67], [74], [75], [115], [146], [176], [181], [182], [247]

detect real complex errors from TeX.

Andrews et al. [13] 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 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 coverage.

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. [14] conducted an experiment to compare test data generation using control flow and data flow. Thevenod et al. [230] applied mutation analysis to compare random and deterministic input generation techniques. Bradbury et al. [31] use mutation analysis to evaluate traditional testing and model checking approaches on concurrent programs.

TABLE VI
EMPIRICAL EVALUATION OF MUTATION TESTING

Research	Evaluation Type	Subject Programs
DeMillo and Mathur [60]	real faults vs mutants	Tex
Mathur and Wong [161], [246]	all-use vs mutation criteria	Find, Strmat1, Strmat2 and Textfmt
Offutt et al. [190]	all-use vs mutation criteria	Bub, Cal, Euclid, Find, Insert, Mid, Pat, Quad, Trityp and Warshall
Daran and Thévenod-Fosse [48]	real faults vs mutants	Nuclear Reactor Safety Shutdown System
Frankl et al. [94], [95]	all-use vs mutation criteria	Determinant, Find1, Find2, Matinv1, Matinv2, Strmatch1, Strmatch2, Textformat.r and Transpose
Andrews et al. [13]	hand seeded faults vs mutants	Space, Printtokens, Printtokens2, Replace, Schedule, Schedule2, Tcas and Totinfo

VII. TOOLS SUPPORT MUTATION TESTING

The development of Mutation Testing tools is an important enabler for transformation of Mutation Testing from the laboratory into a practical and widely used testing technique. Without a fully automated mutation tool, Mutation Testing can be hardly applied in the real world and is unlikely to be accepted by industry. In this section, we summarise development work on Mutation Testing tools. Section VII-A provides an overview of the development work since the 1970s, and Section VII-B provides a brief overview of tools.

A. An Overview of Development Work on Mutation Testing

Since the idea of Mutation Testing was first proposed in the 1970s, many mutation tools have been designed 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 VII-A summarises the application, publication time and any notable characteristics for each tool. A brief description of the most widely studied tools will be provided in the next section.

Figure 13 shows the growth in the number of tools introduced. In Figure 13, the development work can be classified into three stages. The first stage was from 1977 to 1981. In this earlier 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 [3] and the Coupling Effect Hypothesis [65]. The second stage was from 1982 to 1999. There were four tools built in this period, three academic tools, MOTHRA for Fortran [62], [63], PROTEUM, TUMS for C [51], [52], [235] and one industry tool called INSURE++. Engineering effort had been put into MOTHRA and PROTEUM so that they were able handle small real programs not just laboratory programs. As a result, this two academic tools were widely studied. Most of the advanced mutation techniques were experimented upon using these two tools, for example, Weak Mutation [185], [186], Selective Mutation [184], [192], Mutant Sampling [160], [246] and Interface Mutation [53], [54]. 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 13, 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.

TABLE VIII
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

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 VIII 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 more recently, indicating that Mutation Testing has gradually become a practical testing technique, embraced by both of the open source and industrial communities.

B. An Introduction of Mutation Testing Tools

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 laboratory programs, they all implemented the basic structure of mutation analysis. Unfortunately none of these early tools remain available. MOTHRA was the most widely studied Mutation Testing tool. It was designed as the integration of a set of tools, including a parser, a mutant maker, a test case formatter, an interpreter and a decoder. MOTHRA provides a set of Fortran 77 mutation operators and it also provides several optimization techniques, such as constraint-based test data generation [66], Selective Mutation [184] and Weak Mutation [185]. MOTHRA and its source code are available at the website [175].

PROTEUM 1.4 was the first C mutation tool [54]. It applied the compiler based technique, and implemented Agrawal et al.'s 75 of the 77 C mutation operators [6]. PROTEUM 1.4 had been extended into PROTEUM/IM 2.0 to perform Interface Mutation [58]. There was also an extended version of the Proteum tool which supported Specification Mutation technique, Proteum/FSM [86]. TUMS is another mutation tool for C programs. It was the first mutation tool using the Mutant Schema Generation technique [234]–[236]. CSAW is a lightweight Mutation Testing tool for C programs, developed by Ellims et al. [84]. MUTFORMAT is

TABLE VII
SUMMARY OF PUBLISHED MUTATION TESTING TOOLS

Name	Application	Year	Character	Available	Reference
PIMS	Fortran	1977	General	No	[35], [39], [145]
EXPER	Fortran	1979	General	No	[3], [33], [38]
CMS.1	Cobol	1980	General	No	[2], [108]
FMS.3	Fortran	1981	General	No	[229]
Mothra	Fortran	1987	General	No	[62], [63]
Proteum 1.4	C	1993	Interface Mutation, Finite State Machines	No	[51], [52]
TUMS	C	1995	Mutant Schemata Generation	No	[234]–[236]
Insure++	C/C++	1998	Source Code Instrumentation (Commercial)	Commercially	[200]
Proteum/IM 2.0	C	2001	Interface Mutation, Finite State Machines	Yes	[58]
Jester	Java	2001	General (Open Source)	Yes	[165]
Pester	Python	2001	General (Open Source)	Yes	[165]
TDS	CORBA IDL	2001	Interface Mutation	No	[101]
Nester	C#	2002	General (Open Source)	Yes	[221]
JavaMut	Java	2002	General	Yes	[44]
MuJava	Java	2004	Weak Mutation, Mutant Schemata, Reflection Technique	Yes	[151], [152], [187]
Plextest	C/C++	2005	General (Commercial)	Commercially	[119]
SQLMutation	SQL	2006	General	Yes	[232]
Certitude	C/C++	2006	General (Commercial)	Commercially	[41]
SESAME	C, Lustre, Pascal	2006	Assembler Injection	No	[47]
ExMAn	C, Java	2006	TXL	Yes	[29]
MUGAMMA	Java	2006	Remote Monitoring	Yes	[128]
MuClipse	Java	2007	Weak Mutation, Mutant Schemata, Eclipse plug-in	Yes	[219]
CSAW	C	2007	Variable type optimization	Yes	[83], [84]
Heckle	Ruby	2007	General (Open Source)	Yes	[208]
Jumble	Java	2007	General (Open Source)	Yes	[222]
Testooj	Java	2007	General	Yes	[202]
ESPT	C/C++	2008	Tabular	Yes	[90]
MUFORMAT	C	2008	Format String Bugs	No	[215]
CREAM	C#	2008	General	No	[72]
MUSIC	SQL(JSP)	2008	Weak Mutation, SQL Vulnerabilities	No	[213]
MILU	C	2008	Higher Order Mutation, Search-based technique, Test harness embedding	No	[124]
Javalanche	Java	2009	Invariant and Impact analysis	Yes	[106], [210]
Gamera	WS-BPEL	2009	Genetic algorithm	Yes	[78]
MutateMe	PHP	2009	General (Open Source)	Yes	[32]
AjMutator	AspectJ	2009	General	Yes	[50]
JDAMA	SQL(JDBC)	2009	Byte code translation	Yes	[259]

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. [215] using TCL script and provided 8 mutation operators representing String Format faults.

Kim et al. was the first to implement Java mutation tool [131]. 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 JAVA-MUT 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 [151], [152], [187]. 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 [151].

MUGAMMA is a Java mutation extension of the GAMMA framework, developed by Kim et al. [128]. 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. TESTOOJ is an test data generation tool for Java developed by Polo et al. [202]. TESTOOJ integrates several existing testing tools including MuJava. It can be used to generate JUnit tests, construct and execute mutants and reduce test suites using mutation analysis.

TDS is a mutation tool for testing Distributed Component-Based applications, developed by Ghosh et al. [101]. TDS applies Interface Mutation techniques to test the interfaces between two components which defined using CORBA IDL. SESAME is

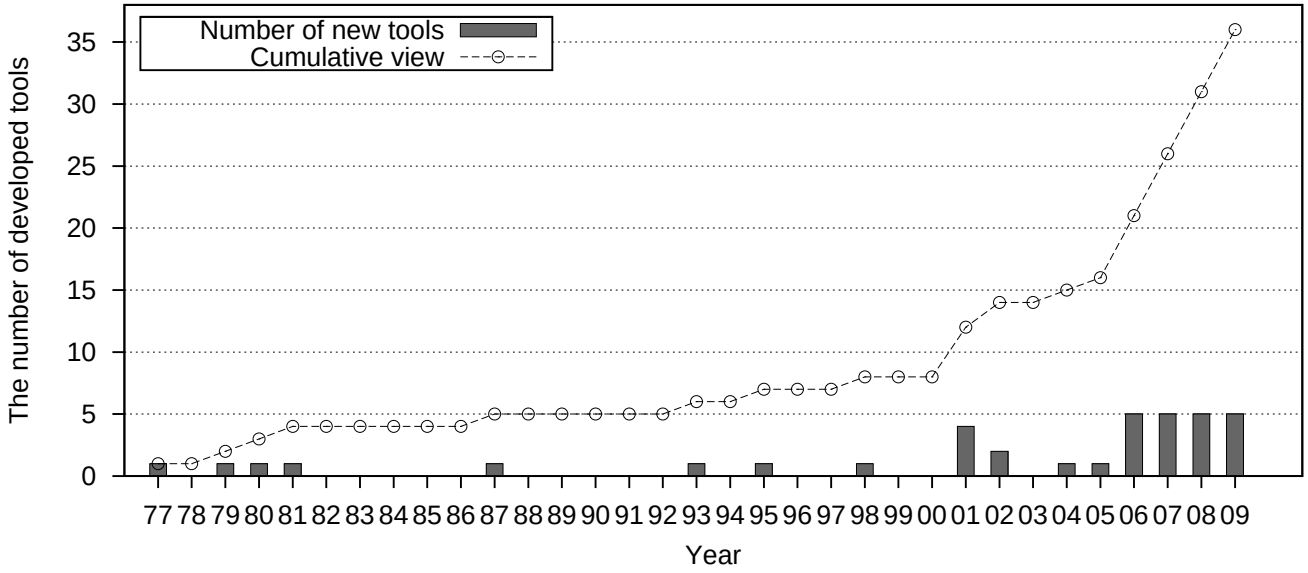

Fig. 13. The number of tools introduced for each year

multi-language mutation tool developed at LAAS-CNRS group [47]. It supports to inject faults into assembly languages, procedural languages, such as Pascal, C and data-flow, such as Lustre. SESAME has been coupled to 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. [29]. 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 [233]. 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. [213]. It was designed to detect SQL Vulnerabilities by injecting faults into SQL statements in JSP applications. It implemented 9 SQL mutation operators proposed by the Shahriar et al. [213]. Weak Mutation was adopted to reduce the computational cost by checking the internal results returned after query executions immediately. JDAMA was developed by Zhou and Frankl to mutate the JDBC interface in Java database programs [259]. JDAMA uses SQLMUTATION as a mutant generator for abstract database queries and applies the Bytecode Translation Technique.

The CREAM system is a mutation testing tool for C# programs. It was proposed by Derezinska and Szustek [72]. The CREAM system implemented five object-oriented mutation operators (EOC, IHD, IPC, IOP, and JID). The ESTP testing platform was designed by Feng et al. [90]. It can be used to generate test data from tabular specifications as well as to measure the effectiveness of new 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 [124]. MILU was the first mutation tool which supports Higher Order Mutation Testing [123], [125]. To reduce the inherited computational cost of the traditional Mutation Testing, MILU provides a set of search based optimization 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 [124]. MILU is also the first tool to use Search Based Software Engineering (SBSE) to optimize Mutation Testing, although SBSE has been widely used to optimize other aspects of testing [109], [163].

AJMUTATOR is a mutation tool for AspectJ programs. It was developed by Delamare et al. [50]. Delamare et al. have implemented three kinds of operators: operators for PCDs, operators for AspectJ declarations, and operators for advice definitions and implementations. An empirical study of using AjMutator can be found in the work of Delamare et al. [49].

JAVALANCHE is a Mutation Testing tool for Java, developed by Schuler et al. [106], [210]. The design of JAVALANCHE 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, JAVALANCHE applies a new form of Weak Mutation approach which analyses the mutants using local invariants [210]. This approach has also been used to prioritize the execution of the non-equivalent mutants [106].

2) Industry and Open Source Tools: JESTER is the first open source tool for Java [165]. It only provides two mutation operators. One changes 0 to 1 and the other is to replace predicates with TRUE and FALSE [164]. A Python version of Jester, PESTER is available from the JESTER's website [165], and a C# version, NESTER is also available [221].

HECKLE is an open source unit mutation tool for Ruby at the RubyForge [208]. 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 [208].

JUMBLE is a class level Java Mutation Testing tool, developed by a commercial company called ReelTwo [205] from 2003 to 2006, and it has been released as an open source project under the

GPL licence since 2007 [222]. 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 an automatic mutant generation system for WS-BPEL compositions based in genetic algorithms. It was first designed as an academic tool by Domínguez et al. [78], [77] and has also been released as an open-source tool. It uses 26 WS-BPEL mutation operators [85] and is composed of three different elements: an analyzer, a mutant generator and a system that executes and evaluates the mutants. GAmera is based on genetic algorithms and attempts to minimize the number of mutants generated, independently of the number and type of mutation operators, without losing relevant information. It can also detect potentially equivalent 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 [32].

INSURE++ was the earliest commercial automatic testing tool for C and C++ using mutation analysis technique [200]. 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 optimize the performance [200].

PLEXTEST is a commercial Mutation Testing tool for C/C++ programs [119]. 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 [41]. 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.

VIII. FUTURE TREND

It would not be possible to conclude this survey, without spending a little time discussing the possible future development of Mutation Testing. Naturally, what follows is very much the authors' own personal view of this possible future, based on the analysis presented in this paper. We see five important avenues for research: a need for high quality higher order mutants, a need to reduce the equivalent mutant problem, a preference for semantics over syntax, an interest in achieving a better balance between cost and value and a pressing need to generate test cases to kill mutants. By addressing these concerns, Mutation Testing will be transformed from a useful but passive test suite assessment technique into a much more dynamic part of the whole process of *improving* test effectiveness while reducing test effort.

As this paper has shown, work on mutation testing is growing at a rapid rate and tools and techniques are reaching an level of maturity not previously witness in this field. It is clear that the future may bring exciting new developments in the field of Mutation Testing. Recent work has tended to focus on more elaborate forms of mutation than on the relatively simply faults that have been previously considered. There is an interest in the semantic effects of mutation, rather than the syntactic achievement of a mutation and this is very much to be welcomed. This migration from the syntactic achievement of mutation to the semantic effect desired has raised interest in higher order mutation to generate subtle faults and to find those mutations that denote real faults. This work can be expected to continue and it is to be hoped that the next ten years will see Mutation Testing further coming of age, with a focus on more realistic mutants that more closely resemble real faults.

One of the barrier to wider application of mutation testing centres on the problems associated with Equivalent Mutants. As the survey shows, there has been a sustained interest in techniques for reducing the impact of equivalent mutants. We see several possible developments along this line. Past work has concentrated on techniques to detect equivalent mutants once they have been produced. In future, Mutation Testing approaches may seek to avoid their initial creation or to reduce their likelihood. Mutation Testing may be applied to languages in which a way that equivalent mutation is not possible. Where equivalent mutants are a possibility, there will be a focus on designing operators and analyzing code so that their likelihood is reduced. Of course, we should be careful not to 'throw the baby out with the bath water'; we seek to retain the highly valuable, so-called stubborn mutants, while filtering out those that are equivalent. However, behaviourally these two classes of mutants are highly similar.

There has also been a great deal of work to extend Mutation Testing to new languages, paradigms and to find new domains of application. Many of these have been covered in this survey; a testament to the wide variety of topics to which Mutation Testing has be applied. We can expect that the future will bring many more applications, and that there will shortly be few widely used programming languages to which Mutation Testing has yet to be applied.

In all aspects of testing there is a trade off to be arrived at that balances the cost of test effort and the value of fault finding ability; a classic tension between effort and effectiveness. Traditionally, Mutation Testing has been seen to be a rather expensive technique that offers high value. However, more recently, authors have started to develop techniques that reduce cost, without over-compromising on quality. It is to be expected that the new paradigms of higher order mutation and more accurate fault modeling will further improve this situation. Perhaps it might even be possible to have much less effort with very specifically tailored mutants, constructed to mimic a known fault model. At the same time, the ability to tailor mutants to fault models may help to improve the value. With Mutation Testing, in the future, it may even be possible to 'have our cake and to eat it'; lower cost and higher value.

Finally, we expect that there will be a shift of emphasis from mutant generation to test case generation. Much of the existing work on Mutation Testing has focussed on techniques to generate mutants and to select and refine those mutants to be passed on to the tester. This work on Mutation Testing has provided a way to asses the quality of test suites, but there has been comparatively little work on *improving* the test suites, based on the associated mutation analysis. We expect that, in future, there will be much more work that seeks to use high quality mutants as a basis for generating high quality test data.

IX. Conclusion

This paper has provided a detailed survey and analysis of trends and results on Mutation Testing. The paper covers theories, optimization techniques, equivalent mutant detection, applications, empirical studies and mutation tools. There has been much optimization 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. Of course, this is also an additional post-hoc motivation for the comprehensive survey provided by this paper.

Acknowledgement

The authors benefitted from many discussions with researchers and practitioners in the mutation testing community, approximately fifty of whom kindly provided very helpful comments and feedback on an earlier draft of this analytical survey. We are very grateful to these colleagues for their time and expertise though we are not able to name them all individually. This work is part funded by EPSRC grants EP/G060525, EP/F059442 and EP/D050863 and by EU grant IST-33742. Yue Jia is additionally supported by a grant from the ORSA scheme.

TABLE IX: 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
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
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
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

Continued on next page

Table IX – continued from previous page

Name	Size	Description	First Use	No. of Uses
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
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
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

Continued on next page

Table IX – continued from previous page

Name	Size	Description	First Use	No. of Uses
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
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
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
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

Continued on next page

Table IX – continued from previous page

Name	Size	Description	First Use	No. of Uses
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

REFERENCES

[1] R. Abraham and M. Erwig, "Mutation Operators for Spreadsheets," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 94–108, January-February 2009.

[2] A. T. Acree, "On Mutation," PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.

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

[4] K. Adamopoulos, "Search Based Test Selection and Tailored Mutation," Masters Thesis, King's College London, London, Uk, 2009.

[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)*, ser. LNCS, vol. 3103. Seattle, Washington, USA: Springer, 26th-30th, June 2004, pp. 1338–1349.

[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," Purdue University, West Lafayette, Indiana, Technique Report SERC-TR-41-P, March 1989.

[7] B. K. Aichernig, "Mutation Testing in the Refinement Calculus," *Formal Aspects of Computing*, vol. 15, no. 2-3, pp. 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)*, ser. LNCS, vol. 3922. Vienna, Austria: Springer, 27-28 March 2006, pp. 324–338.

[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)*. Annapolis, Maryland: IEEE Computer Society, 12-15 November 2002, pp. 341–351.

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

[11] 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)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 3.

[12] 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)*. Redmond, Washingto: IEEE Computer Society, 11-14 November 2008, pp. 239–248.

[13] 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)*, St Louis, Missouri, 15-21 May 2005, pp. 402 – 411.

[14] 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*, vol. 32, no. 8, pp. 608–624, August 2006.

[15] 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)*, London, England, 7-11 July 2007, pp. 1074–1081.

[16] 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)*. Raleigh, North Carolina: IEEE Computer Society, 7-10 November 2006, pp. 169–178.

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

[18] 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*, vol. 11, no. 2, pp. 113–136, May 2001.

[19] 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)*, ser. LNCS, vol. 4574. Tallinn, Estonia: Springer, 26-29 June 2007, pp. 50–65.

[20] B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. Le Traon, "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment," in *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, Annapolis, Maryland, 12-15 November 2002, pp. 195–206.

[21] B. Baudry, V. Le Hanh, J.-M. Jézéquel, and Y. Le Traon, "Trustable Components: Yet Another Mutation-Based Approach," in *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, published in book form, as *Mutation Testing for the New Century*. San Jose, California, 6-7 October 2001, pp. 47–54.

[22] 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)*. Raleigh, North Carolina: IEEE Computer Society, 2006, p. 9.

[23] 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)*, San Diego, California, 26-29 November 2001, p. 337.

[24] 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)*, published in book form, as *Mutation Testing for the New Century*. San Jose, California, 6-7 October 2001, pp. 14–20.

[25] 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)*, ser. LNCS, vol. 4044, 2006, Oulu, 17-22 June 2006, pp. 202–204.

[26] B. Bogacki and B. Walter, "Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing," in *Software Engineering Techniques: Design for Quality*, ser. IFIP, vol. 227, 2007, pp. 273–282.

[27] 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)*, Munich, Germany, 10-14 March 2008, pp. 396–401.

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

[29] 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)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, pp. 57–62.

[30] 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)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, pp. 83–92.

[31] 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)*, published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*. Windsor, UK: IEEE Computer Society, 2007, pp. 210–222.

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

[33] T. A. Budd, "Mutation Analysis of Program Test Data," PhD Thesis, Yale University, New Haven, Connecticut, 1980.

[34] T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, March 1982.

[35] 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*, vol. 74. Anaheim, New Jersey: ACM, 5-8 June 1978, pp. 623–627.

[36] 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)*, Las Vegas, Nevada, 28-30 January 1980, pp. 220–233.

[37] T. A. Budd and A. S. Gopal, "Program Testing by Specification Mutation," *Computer Languages*, vol. 10, no. 1, pp. 63–73, 1985.

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

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

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

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

[42] 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)*, Melbourne, Australia, 19 -20 September 2005, pp. 187–196.

[43] 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)*, Macau, China, 4-7 December 2001, p. 267.

[44] P. Chevalley and P. Thévenod-Fosse, "A Mutation Analysis Tool for Java Programs," *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 1, pp. 90–103, November 2002.

[45] B. Choi and A. P. Mathur, "High-performance Mutation Testing," *Journal of Systems and Software*, vol. 20, no. 2, pp. 135–152, February 1993.

[46] W. M. Craft, "Detecting Equivalent Mutants Using Compiler Optimization Techniques," Masters Thesis, Georgia Institute of Technology, 1989.

[47] 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)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 7.

[48] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 158–177, May 1996.

[49] R. Delamare, B. Baudry, , S. Ghosh, and Y. Le Traon, "A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ," in *Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST'09)*, Davor Colorado, 01-04 April 2009, pp. 376–385.

[50] 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)*, published with *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops*. Denver, Colorado: IEEE Computer Society, 1-4 April 2009, pp. 200–204.

[51] M. E. Delamaro, "Proteum - A Mutation Analysis Based Testing Environmen," Masters Thesis, University of São Paulo, Sao Paulo, Brazil, 1993.

[52] 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)*, New Brunswick, New Jersey, July 1996, pp. 79–95.

[53] 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)*, Talca, Chile, 11-13 November 1999, pp. 78–86.

[54] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Integration Testing Using Interface Mutation," in *Proceedings of the seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, White Plains, New York, 30 October - 02 November 1996, pp. 112–121.

[55] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface Mutation: An Approach for Integration Testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, May 2001.

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

[57] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur, "Interface Mutation Test Adequacy Criterion: An Empirical Evaluation," *Empirical Software Engineering*, vol. 6, no. 2, pp. 111–142, June 2001.

[58] 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)*, published in book form, as *Mutation Testing for the New Century*. San Jose, California, 6-7 October 2001, pp. 91–101.

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

[60] 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," Purdue University, West Lafayette, Indiana, Technique Report SERC-TR-92-P, 1992.

[61] R. A. DeMillo, "Test Adequacy and Program Mutation," in *Proceedings of the 11th International Conference on Software Engineering (ICSE'89)*, Pittsburgh, Pennsylvania, 15-18 May 1989, pp. 355–356.

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

[63] 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)*. Banff Alberta,Canada: IEEE Computer society, July 1988, pp. 142–151.

[64] 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)*. Tokyo, Japan: IEEE Computer Society Press, September 1991, pp. 351–356.

[65] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[66] R. A. DeMillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[67] R. A. DeMillo and A. J. Offutt, "Experimental Results From an Automatic Test Case Generator," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 2, pp. 109–127, April 1993.

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

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

[70] 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.

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

[72] 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)*, Szklarska Porêba, Poland, 26-28 June 2008, pp. 261–268.

[73] W. Ding, "Using Mutation to Generate Tests from Specifications," Masters Thesis, George Mason University, Fairfax, VA, 2000.

[74] H. V. Do, C. Robach, and M. Delaunay, "Mutation Analysis for Reactive System Environment Properties," in *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 2.

[75] 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)*, Budapest, Hungary, 25-30 September 2005, pp. 411–420.

[76] H. Do and G. Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, September 2006.

[77] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "GAmera: An Automatic Mutant Generation System for WS-BPEL Compositions," in *Proceedings of the 7th European Conference on Web Services (ECOWS'09)*, Eindhoven, Netherlands, 9-11 November 2009.

[78] 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*, ser. LNCS, vol. 5404. Spindleruv Mlyn, Czech Republic: Springer, January 2009, pp. 229 – 240.

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

[80] 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)*, New York, NY, 25-28 June 2000, pp. 603–612.

[81] 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)*, published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*. Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 176–184.

[82] 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)*, Cap Esterel, French Riviera, France, 25-31 August 2007, p. 40.

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

[84] 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)*, published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*. Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 185–192.

[85] 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.

[86] 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)*, Talca, Chile, 11-13 November 1999, p. 96.

[87] 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*, vol. 43, 1995, pp. 329–337.

[88] 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)*, Boca Raton, Florida, 1-4 November 1999, p. 210.

[89] 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*, Monterey, California, 6-9 November 1994, pp. 220–229.

[90] 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)*, Windsor, UK, 29-31 August 2008, pp. 59–63.

[91] 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)*. Lillehammer, Norway: IEEE Computer Society, 9-11 April 2008, pp. 52–61.

[92] S. Fichter, "Parallelizing Mutation on a Hypercube," Masters Thesis, Clemson University, Clemson, SC, 1991.

[93] 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)*. Seattle, Washington: ACM Press, August 1994, pp. 185–195.

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

[95] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs Mutation Testing: an Experimental Comparison of Effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, September 1997.

[96] 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)*, published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*. Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 161–168.

[97] R. Geist, A. J. Offutt, and F. C. Harris, "Estimation and Enhancement of Real-Time Software Reliability Through Mutation Analysis," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 550–558, May 1992.

[98] 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)*, Oakland, California, 3-6 May 1998, pp. 104–114.

[99] S. Ghosh, "Testing Component-Based Distributed Applications," PhD Thesis, Purdue University, West Lafayette, Indiana, 2000.

[100] 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)*, Los Alamitos, California, 2002, p. 7.

[101] 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)*, published in book form, as *Mutation Testing for the New Century*. San Jose, California, 6-7 October 2001, pp. 103–112.

[102] 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)*, Santa Barbara, California, 30 July - 4 August 2000, p. 37.

[103] S. Ghosh and A. P. Mathur, "Interface Mutation," *Software Testing, Verification and Reliability*, vol. 11, no. 3, pp. 227–247, March 2001.

[104] 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)*. London, England: IEEE Computer Society Press, August 1985, pp. 313–319.

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

[106] 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)*, published with *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops*. Denver, Colorado: IEEE Computer Society, 1-4 April 2009, pp. 192–199.

[107] R. G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, July 1977.

[108] J. M. Hanks, "Testing Cobol Programs by Mutation," PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.

[109] 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.

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

[111] R. M. Hierons, M. Harman, and S. Danicic, "Using Program Slicing to Assist in the Detection of Equivalent Mutants," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233–262, December 1999.

[112] 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)*, published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*. Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 141–150.

[113] R. M. Hierons and M. G. Merayo, "Mutation Testing from Probabilistic and Stochastic Finite State Machines," *Journal of Systems and Software*, To appear.

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

[115] 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)*, Paris, France, 2-5 October 2007, pp. 174–183.

[116] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 371–379, July 1982.

[117] S. Hussain, "Mutation Clustering," Masters Thesis, King's College London, Strand, London, 2008.

[118] 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)*, Napoli, Italy, 6-8 October 2008, pp. 105–114.

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

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

[121] 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: Knowledge Systems Institute Graduate School, 1-3 July 2009.

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

[123] 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)*, Beijing, China, 28-29 September 2008, pp. 249–258.

[124] 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)*. Windsor, UK: IEEE Computer Society, 29-31 August 2008, pp. 94–98.

[125] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Journal of Information and Software Technology*, To appear.

[126] 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)*, Okinawa, Japan, 25-28 March 2008, pp. 667–674.

[127] K. Kapoor, "Formal Analysis of Coupling Hypothesis for Logical Faults," *Innovations in Systems and Software Engineering*, vol. 2, no. 2, pp. 80–87, July 2006.

[128] 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)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 10.

[129] 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.

[130] 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.

[131] 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.

[132] 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)*, published in book form, as *Mutation Testing for the New Century*. San Jose, California, 6-7 October 2001, pp. 207–225.

[133] K. N. King and A. J. Offutt, "A Fortran Language System for Mutation-Based Software Testing," *Software:Practice and Experience*, vol. 21, no. 7, pp. 685–718, October 1991.

[134] E. W. Krauser, "Compiler-Integrated Software Testing," PhD Thesis, Purdue University, West Lafyette, 1991.

[135] E. W. Krauser, A. P. Mathur, and V. J. Rego, "High Performance Software Testing on SIMD Machines," *IEEE Transactions on Software Engineering*, vol. 17, no. 5, pp. 403–423, May 1991.

[136] 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)*. Banff Alberta: IEEE Computer Society, July 1988, pp. 171 – 177.

[137] W. B. Langdon, M. Harman, and Y. Jia, "Multi Objective Higher Order Mutation Testing With Genetic Programming," in *Proceedings of the 4th Testing: Academic and Industrial Conference - Practice and Research (TAIC PART'09)*. Windsor, UK: IEEE Computer Society, 4-6 September 2009.

[138] W. B. Langdon, M. Harman, and Y. Jia, "Multi Objective Mutation Testing With Genetic Programming," in *Proceedings of the Genetic and Evolutionary Computation Conference 2009 (GECCO'09)*, Montréal, Canada, 8-12 July 2009.

[139] Y. Le Traon, B. Baudry, and J.-M. Jézéquel, "Design by Contract to Improve Software Vigilance," *IEEE Transactions of Software Engineering*, vol. 32, no. 8, pp. 571–586, August 2006.

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

[141] 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)*, Ottawa, Canada., 14-16 April 2008, pp. 149–156.

[142] S. D. Lee, "Weak vs. Strong: An Empirical Comparison of Mutation Variants," Masters Thesis, Clemson University, Clemson, SC, 1991.

[143] 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)*, Hong Kong, China, November 2001, pp. 200–209.

[144] 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)*, Turku, Finland, 26-28 July 2005, pp. 443–448.

[145] 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*, December 1978, pp. 355–373.

[146] 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)*, Philadelphia, Pennsylvania, USA, 24-27 September 2006, pp. 113–122.

[147] 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)*, Santa Fe, New Mexico, 26-30 April 2004, p. 265.

[148] Y.-S. Ma, "Object-Oriented Mutation Testing for Java," PhD Thesis, KAIST University in Korea, 2005.

[149] 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)*, Shanghai, China, 20-28 May 2006, pp. 869–872.

[150] 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)*. Annapolis, Maryland: IEEE Computer Society, 12-15 November 2002, p. 352.

[151] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification & Reliability*, vol. 15, no. 2, pp. 97–133, June 2005.

[152] 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)*, Shanghai, China, 20-28 May 2006, pp. 827–830.

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

[154] 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*. Banff, Alberta, Canada: ACM, 8-12 May 2007, pp. 667–676.

[155] A. P. Mathur, *Foundations of Software Testing*. Pearson Education, 2008.

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

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

[158] A. P. Mathur, "Mutation Testing," in *Encyclopedia of Software Engineering*, J. J. Marciniak, Ed., 1994, pp. 707–713.

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

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

[161] A. P. Mathur and W. E. Wong, "An Empirical Comparison of Data Flow and Mutation-based Test Adequacy Criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9 – 31, 1994.

[162] P. S. May, "Test Data Generation: Two Evolutionary Approaches to Mutation Testing," PhD Thesis, University of Kent, Canterbury, Kent, 2007.

[163] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

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

[165] I. Moore, "Jester and Pester," http://jester.sourceforge.net/, 2001.
[166] L. J. Morell, "A Theory of Error-Based Testing," PhD Thesis, University of Maryland at College Park, College Park, Maryland, 1984.
[167] 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)*. Lillehammer, Norway: IEEE Computer Society, 9-11 April 2008, pp. 278–286.
[168] 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)*, published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*. Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 233–242.
[169] E. S. Mresa and L. Bottaci, "Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 205–232, December 1999.
[170] 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)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 5.
[171] A. S. Namin and J. H. Andrews, "On Sufficiency of Mutants," in *Proceedings of the 29th International Conference on Software Engineering (ICSE COMPANION'07)*, Minneapolis, Minnesota, 20-26 May 2007, pp. 73–74.
[172] 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)*, Leipzig, Germany, 10-18 May 2008, pp. 351–360.
[173] 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)*, Hong Kong, China, 28-30, September 2004, pp. 306–311.
[174] 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)*, ser. ENTCS, vol. 164, no. 4, Vienna, Austria, 25-26 March 2006, pp. 97–114.
[175] Offutt, "Mothra," http://jwww.cs.qmu.edu/ offutt/rsrch/mut.html, 2009.
[176] 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*, Washington, DC, June 1995, pp. 111–123.
[177] A. J. Offutt, "The Coupling Effect: Fact or Fiction," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, December 1989.
[178] A. J. Offutt, "Investigations of the Software Testing Coupling Effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, January 1992.
[179] A. J. Offutt, "Private Communication," July 2008.
[180] A. J. Offutt and W. M. Craft, "Using Compiler Optimization Techniques to Detect Equivalent Mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 131–154, September 1994.
[181] A. J. Offutt, Z. Jin, and J. Pan, "The Dynamic Domain Reduction Approach for Test Data Generation: Design and Algorithms," George Mason University, Fairfax, Virginia, Technical Report ISSE-TR-94-110, 1994.
[182] A. J. Offutt, Z. Jin, and J. Pan, "The Dynamic Domain Reduction Procedure for Test Data Generation," *Software:Practice and Experience*, vol. 29, no. 2, pp. 167–193, February 1999.
[183] A. J. Offutt and K. N. King, "A Fortran 77 Interpreter for Mutation Analysis," *ACM SIGPLAN Notices*, vol. 22, no. 7, pp. 177–188, July 1987.
[184] 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*, vol. 5, no. 2, pp. 99–118, April 1996.
[185] A. J. Offutt and S. Lee, "An Empirical Evaluation of Weak Mutation," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 337–344, May 1994.
[186] 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)*. Victoria, British Columbia, Canada: IEEE Computer Society, October 1991, pp. 200 – 213.
[187] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "An Experimental Mutation System for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 5, pp. 1–4, September 2004.
[188] A. J. Offutt and J. Pan, "Detecting Equivalent Mutants and the Feasible Path Problem," in *Proceedings of the 1996 Annual Conference on Computer Assurance*. Gaithersburg, Maryland: IEEE Computer Society Press, June 1996, pp. 224–236.
[189] A. J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, September 1997.
[190] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," *Software:Practice and Experience*, vol. 26, no. 2, pp. 165–176, February 1996.
[191] 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*, Chicago, Illinois, August 1992, pp. 255–266.
[192] 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)*. Baltimore, Maryland: IEEE Computer Society Press, May 1993, pp. 100–107.
[193] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," in *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, published in book form, as *Mutation Testing for the New Century*. San Jose, California, 6-7 October 2001, pp. 34–44.
[194] A. J. Offutt, J. Voas, and J. Payn, "Mutation Operators for Ada," George Mason University, Fairfax, Virginia, Technique Report ISSE-TR-96-09, 1996.
[195] 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)*, Boston, Massachusetts, 11-14 July 2004, pp. 1 – 10.
[196] A. J. Offutt, "Automatic Test Data Generation," PhD Thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1988.
[197] V. Okun, "Specification Mutation for Test Generation and Analysis," PhD Thesis, University of Maryland Baltimore County, Baltimore, Maryland, 2004.
[198] 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)*, Washington DC, 17-20 April 2001, p. 222.
[199] J. Pan, "Using Constraints to Detect Equivalent Mutants," Masters Thesis, George Mason University, Fairfax VA, 1994.
[200] Parasoft, "Parasoft Insure++," http://www.parasoft.com/jsp/products/home.jsp?product=Insure, 2006.
[201] M. Polo, M. Piattini, and I. Garcia-Rodriguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *Software Testing, Verification and Reliability*, vol. 19, no. 2, pp. 111 – 131, June 2008.
[202] M. Polo, S. Tendero, and M. Piattini, "Integrating techniques and tools for testing automation: Research Articles," *Software Testing, Verification and Reliability*, vol. 17, no. 1, pp. 3–39, March 2007.
[203] 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)*. Lillehammer, Norway: IEEE Computer Society, 9-11 April 2008, pp. 338–347.
[204] R. Probert and F. Guo, "Mutation Testing of Protocols: Principles and Preliminary Experimental Results," in *Proceedings of the Workshop on Protocol Test Systems*, Leidschendam, Netherland, 15-17 October 1991, pp. 57–76.
[205] ReelTwo, http://www.reeltwo.com, 2007.
[206] 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)*. Montreal, Quebec, Canada: ACM, 12-13 May 2008, pp. 137–140.
[207] 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)*, published with *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops*. Denver, Colorado: IEEE Computer Society, 1-4 April 2009, pp. 157–166.
[208] Rubyforge, "Heckle," http://seattlerb.rubyforge.org/heckle/, 2007.
[209] 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)*. Garmisch Partenkirchen, Germany: Elsevier Science, September 1990, pp. 43–56.

[210] 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.

[211] 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*, Lubeck, Germany, 29-31 August 2007, pp. 222–227.

[212] 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)*, Atlanta, GA, 15-17 October 2007, pp. 218–223.

[213] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL Injection Vulnerability Checking," in *Proceedings of the 8th International Conference on Quality Software (QSIC'08)*, Oxford, UK, 12-13 August 2008, pp. 77–86.

[214] 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*, 28 July -1 August, Turku, Finland 2008, pp. 979–984.

[215] 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)*, Nanjing, China, 3-5 Dec 2008, pp. 229–238.

[216] 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)*, Vancouver, Canada, 19 May 2009, pp. 47–53.

[217] 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)*, New Orleans, Louisiana, 27-31 March 1988, pp. 80–85.

[218] A. Simao, J. C. Maldonado, and R. da Silva Bigonha, "A Transformational Language for Mutant Description," *Computer Languages, Systems & Structures*, vol. 35, no. 3, pp. 322–339, October 2009.

[219] 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)*, published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*. Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 193–202.

[220] B. H. Smith and L. Williams, "On Guiding the Augmentation of an Automated Test Suite via Mutation Analysis," *Empirical Software Engineering*, vol. 14, no. 3, pp. 341–369, 2009.

[221] SourceForge, "Nester," http://nester.sourceforge.net/, 2002.

[222] SourceForge, "Jumble," http://jumble.sourceforge.net/, 2007.

[223] 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*, vol. 8, no. 4, pp. 285–301, December 1999.

[224] 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)*, vol. 8, Maui, Hawaii, 4-7 January 2000, p. 8011.

[225] E. H. Spafford, "Extending Mutation Testing to Find Environmental Bugs," *Software:Practice and Experience*, vol. 20, no. 2, pp. 181–189, February 1990.

[226] 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)*, Chiang Mai, Thailand, 10-12 December 2003, pp. 340–350.

[227] 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*, ser. LNCS, vol. 2978, Oxford, UK, 17-19 March 2004, p. 2741.

[228] 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)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 8.

[229] A. Tanaka, "Equivalence Testing for Fortran Mutation System Using Data Fow Analysis," Masters Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1981.

[230] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet, "An Experimental Study on Software Structural Testing: Deterministi cversus Random Input Generation," in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS'91)*, Montréal, Canada, 25-27 June 1991, pp. 410–417.

[231] 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)*, published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*. Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 151–160.

[232] 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)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 1.

[233] J. Tuya, M. J. S. Cabal, and C. de la Riva, "Mutating Database Queries," *Information and Software Technology*, vol. 49, no. 4, pp. 398–417, April 2007.

[234] R. H. Untch, "Mutation-based Software Testing Using Program Schemata," in *Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE'92)*, Raleigh, North Carolina, 1992, pp. 285–291.

[235] 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.

[236] 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)*, Cambridge, Massachusetts, 1993, pp. 139–148.

[237] 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*, Washington DC, USA, 2004, pp. 21–30.

[238] P. Vilela, M. Machado, and W. E. Wong, "Testing for Security Vulnerabilities in Software," in *Software Engineering and Applications*, 2002.

[239] 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*, vol. 11, no. 4, pp. 249–268, November 2001.

[240] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.

[241] K. S. H. T. Wah, "Fault Coupling in Finite Bijective Functions," *Software Testing, Verification and Reliability*, vol. 5, no. 1, pp. 3–47, 1995.

[242] K. S. H. T. Wah, "A Theoretical Study of Fault Coupling," *Software Testing, Verification and Reliability*, vol. 10, no. 1, pp. 3–46, April 2000.

[243] K. S. H. T. Wah, "An Analysis of the Coupling Effect I: Single Test Data," *Science of Computer Programming*, vol. 48, no. 2-3, pp. 119–161, August-September 2003.

[244] 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)*, Seoul, Republic of Korea, 20-22 October 2008, pp. 71–76.

[245] S. N. Weiss and V. N. Fleyshgakker, "Improved Serial Algorithms for Mutation Analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 3, pp. 149–158, July 1993.

[246] W. E. Wong, "On Mutation and Data Flow," PhD Thesis, Purdue University, West Lafayette, Indiana, 1993.

[247] 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*, vol. 48, no. 2, pp. 79–89, October 1999.

[248] W. E. Wong and A. P. Mathur, "Reducing the Cost of Mutation Testing: An Empirical Study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, December 1995.

[249] M. R. Woodward, "Mutation Testing-An Evolving Technique," in *Proceedings of the IEE Colloquium on Software Testing for Critical Systems*, London, UK, 19 June 1990, pp. 3/1–3/6.

[250] 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*, 14 Jan 1990, p. 2.

[251] M. R. Woodward, "Errors in Algebraic Specifications and an Experimental Mutation Testing Tool," *Software Engineering Journal*, vol. 8, no. 4, pp. 221–224, July 1993.

[252] 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). Banff Albert, Canada: IEEE Computer Society, July 1988, pp. 152–158.

[253] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Mutation Analysis of Parameterized Unit Tests," in *Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09)*, published with *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops*. Denver, Colorado: IEEE Computer Society, 1-4 April 2009, pp. 177–181.

[254] 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)*, Chicago Illinois, 14-16 July 2005, pp. 257–266.

[255] 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)*, Taipei, Taiwan, 2-4 December 1998, p. 174.

[256] C. N. Zapf, "A Distributed Interpreter for the Mothra Mutation Testing System," Masters Thesis, Clemson University, Clemson, South Carolina, 1993.

[257] 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)*, Washington DC, USA, 25-29 June 2005, pp. 1061–1068.

[258] 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)*, published with *Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07)*. Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 223–232.

[259] 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)*, Davor Colorado, 01-04 April 2009, pp. 396–405.