An Analysis and Survey of the Development of Mutation Testing

Yue Jia, *Student Member*, *IEEE*, and Mark Harman, *Member*, *IEEE*

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. 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.

1 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. By carefully choosing the location and type of mutant, we can also simulate any test adequacy criteria. Such faults are deliberately seeded into the original program by simple syntactic changes 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. If the result of running a mutant is different from the result of running the original program for any test cases in the input test set, the seeded fault denoted by the mutant is detected. One outcome of the Mutation Testing process is the mutation score, which indicates the quality of the input test set. The mutation score is the ratio of the number of detected faults over the total number of the seeded faults.

The history of Mutation Testing can be traced back to 1971 in a student paper by Lipton [144]. The birth of the field can also be identified in papers published in the late 1970s by DeMillo et al. [66] and Hamlet [107].

Mutation Testing can be used for 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], [36], [40], [131], [145], [181], Ada programs [29], [192], C programs [6], [56], [97], [213], [214], [237], [239], Java programs [44], [45], [127], [128], [129], [130], [150], [151], C# programs [69], [70], [71], [72], [73], SQL code [43], [212], [233], [234], and AspectJ programs [12], [13], [17], [90]. Mutation Testing has also been used for integration testing [54], [55], [56], [58]. 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 [20], [28], [88], [111], Statecharts [95], [231], [260], Estelle Specifications [222], [223], Petri Nets [86], Network protocols [124], [202], [216], [238], Security Policies [139], [154], [165], [166], [201], and Web Services [140], [142], [143], [193], [245], [259].

Mutation Testing has been increasingly and 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 [62] 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) subarea of Strong, Weak, and Firm mutation techniques was presented by Woodward [253], [256]. An introductory chapter on Mutation Testing can be found in the book by Mathur [155] and also in the book by Ammann and Offutt [11]. The most recent survey work was conducted by Offutt and Untch [191] in 2000. They summarized the history of Mutation Testing and provide an overview of the existing optimization techniques for Mutation Testing. However, since then, there have been more than 230 new publications on Mutation Testing.

In order to provide a complete survey covering all the publications related to Mutation Testing since the 1970s, we constructed a Mutation Testing publication repository, which includes more than 390 papers from 1977 to 2009 [121]. We also searched for master's and PhD theses that have made a significant contribution to the development of Mutation Testing. These are listed in Table 1. We took four steps to build this repository. First, we searched the online repositories of the main technical publishers, including

• *The authors are with the Department of Computer Science, University College London, Malet Place, London WC1E 6BT, UK. E-mail: {yue.jia, mark.harman}@cs.ucl.ac.uk.*

Manuscript received 21 Sept. 2009; revised 15 Feb. 2010; accepted 1 Apr. 2010; published online 10 June 2010.
Recommended for acceptance by P. Frankl.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-09-0232.
Digital Object Identifier no. 10.1109/TSE.2010.62.

TABLE 1
A List of PhD and Master's 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 [34]	Mutation Analysis of Program Test Data	PhD	Yale University	1980
Tanaka [228]	Equivalence Testing for Fortran Mutation System Using Data Flow Analysis	PhD	Georgia Institute of Technology	1981
Morell [164]	A Theory of Error-Based Testing	PhD	University of Maryland at College Park	1984
Offutt [194]	Automatic Test Data Generation	PhD	Georgia Institute of Technology	1988
Craft [48]	Detecting Equivalent Mutants Using Compiler Optimization Techniques	Master	Clemson University	1989
Choi [46]	Software Testing Using High-performance Computers	PhD	Purdue University	1991
Krauser [132]	Compiler-Integrated Software Testing	PhD	Purdue University	1991
Fichter [91]	Parallelizing Mutation on a Hypercube	Master	Clemson University	1991
Lee [141]	Weak vs. Strong: An Empirical Comparison of Mutation Variants	Master	Clemson University	1991
Zapf [261]	A Distributed Interpreter for the Mothra Mutation Testing System	PhD	Clemson University	1993
Delamaro [52]	Proteum - A Mutation Analysis Based Testing Environment	PhD	University of São Paulo	1993
Wong [248]	On Mutation and Data Flow	PhD	Purdue University	1993
Pan [197]	Using Constraints to Detect Equivalent Mutants	Master	George Mason University	1994
Untch [236]	Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method	PhD	Clemson University	1995
Ghosh [98]	Testing Component-Based Distributed Applications	PhD	Purdue University	2000
Ding [74]	Using Mutation to Generate Tests from Specifications	Master	George Mason University	2000
Okun [195]	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 [161]	Test Data Generation: Two Evolutionary Approaches to Mutation Testing	PhD	University of Kent	2007
Hussain [116]	Mutation Clustering	Master	King's College London	2008
Adamopoulos [4]	Search Based Test Selection and Tailored Mutation	Master	King's College London	2009

IEEE XPlore, ACM Portal, Springer Online Library, Wiley InterScience, and Elsevier Online Library, collecting papers which have either "mutation testing," "mutation analysis," "mutants + testing," "mutation operator + testing," "fault injection," and "fault-based testing" keywords in their title or abstract. Then, we went through the references for each paper in our repository to find missing papers using the same keyword rules. In this way, we performed a "transitive closure" on the literature. Mutation Testing work which was not concerned with software, for example, hardware, was removed and we also filtered out papers not written in English. Finally, we sent a draft of this paper to all cited authors asking them to check our citations. We have made the repository publicly available at http://www.dcs.kcl.ac.uk/pg/jiayue/repository/ [121]. Overall growth trend of all papers in Mutation Testing can be found in Fig. 1.

The rest of the paper is organized as follows: Section 2 introduces the fundamental theory of Mutation Testing, including the hypotheses, the process, and the problems of Mutation Testing. Section 3 explains the techniques for the reduction of the computational cost. Section 4 introduces the techniques for detecting equivalent mutants. The applications of Mutation Testing are introduced in Section 5. Section 6 summarizes the empirical experiments of the research work on Mutation Testing. Section 7 describes the development work on mutation tools. Section 8 discusses the evidences for the increasing importance of Mutation Testing. Section 9 discusses the unresolved problems, barriers, and the areas of success in Mutation Testing. The paper concludes in Section 10.

2 The Theory of Mutation Testing

This section will first introduce the two fundamental hypotheses of Mutation Testing. It then discusses the general process of Mutation Testing and the problems from which it suffers.

2.1 Fundamental Hypotheses

Mutation Testing promises to be effective in identifying adequate test data which can be used to find real faults [96]. However, the number of such potential faults for a given program is enormous; it is impossible to generate mutants representing all of them. Therefore, traditional Mutation Testing targets only a subset of these faults, those which are close to the correct version of the program, with the hope that these will be sufficient to simulate all faults. This theory is based on two hypotheses: the Competent Programmer Hypothesis (CPH) [3], [66] and the Coupling Effect [66].

The CPH was first introduced by DeMillo et al. in 1978 [66]. It states that programmers are competent, which implies that they tend to develop programs close to the

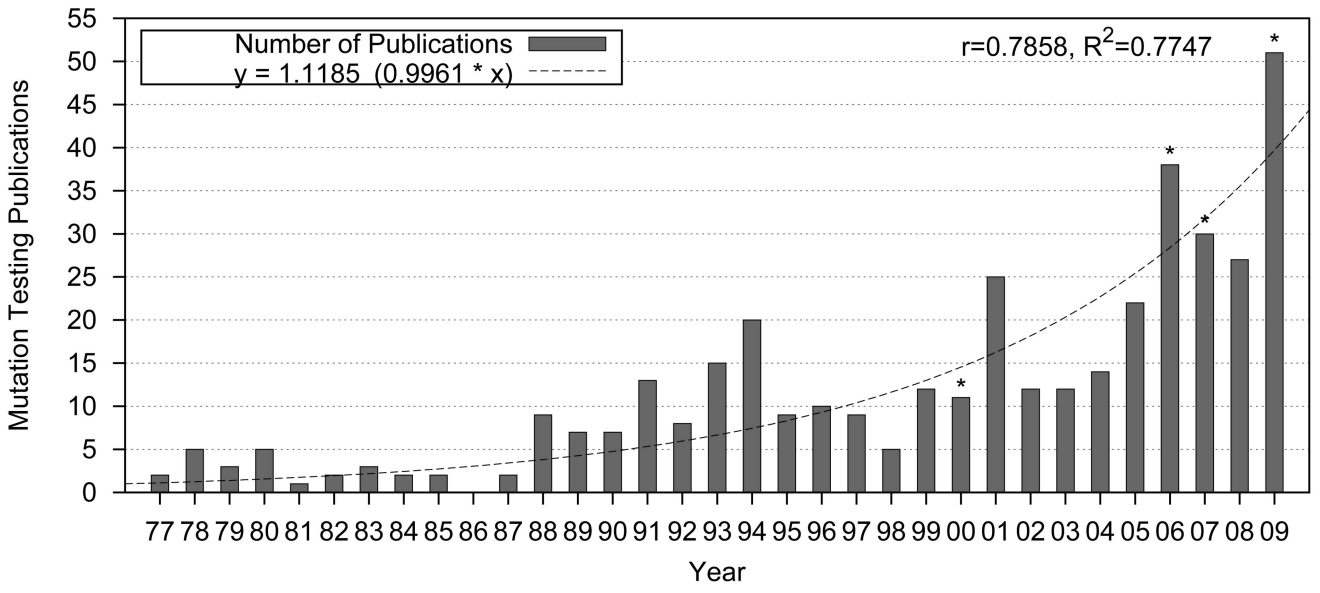

Fig. 1. Mutation testing publications from 1978-2009 (* indicates years in which a mutation workshop was held).

correct version. As a result, although there may be faults in the program delivered by a competent programmer, we assume that these faults are merely a few simple faults which can be corrected by a few small syntactical changes. Therefore, in Mutation Testing, only faults constructed from several simple syntactical changes are applied, which represent the faults that are made by "competent programmers." An example of the CPH can be found in Acree et al.'s work [3]. A theoretical discussion using the concept of program neighborhoods can also be found in Budd et al.'s work [37].

The Coupling Effect was also proposed by DeMillo et al. in 1978 [66]. Unlike the CPH concerning a programmer's behavior, 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 [174], [175] extended this into the Coupling Effect Hypothesis and the Mutation Coupling Effect Hypothesis with a precise definition of simple and complex faults (errors). In his definition, a simple fault is represented by a simple mutant which is created by making a single syntactical change, while a complex fault is represented as a complex mutant which is created by making more than one change.

According to Offutt, the Coupling Effect Hypothesis is that "complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults" [175]. 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" [175]. 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], [164], [174], [175]. Lipton and Sayward [145] conducted an empirical study using a small program, FIND. In their experiment, a small sample of second order, third order, and fourth order mutants is investigated. The results suggested that an adequate test set generated from first order mutants was also adequate for the samples of kth order mutants ($k = 2, \ldots, 4$). Offutt [174], [175] extended this experiment using all possible second order mutants with two more programs, MID and TRITYP. The results suggested that test data developed to kill first order mutants killed over 99 percent second order and third order mutants. This study implied that the mutation coupling effect hypothesis does indeed manifest itself in practice. Similar results were found in the empirical study by Morell [164].

The validity of the mutation coupling effect has also been considered in the theoretical studies of Wah [242], [243], [244] and Kapoor [125]. In Wah's work [243], [244], 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 first order and the second order model. Empirical results indicated that the average survival ratio of first order mutants and second order mutants is $1/n$ and $1/n^2$, respectively, where n is the order of the domain [243]. This result is also similar to the estimated results of the empirical studies mentioned above. A formal proof of the coupling effect on the boolean logic faults can be also found in Kapoor's work [125].

2.2 The Process of Mutation Analysis

The traditional process of mutation analysis is illustrated in Fig. 2. In mutation analysis, from a program p, a set of faulty programs p', called mutants, is generated by a few single syntactic changes to the original program p. As an illustration, Table 2 shows the mutant p', generated by changing the and operator (&&) of the original program p, into the or operator (||), thereby producing the mutant p'.

A transformation rule that generates a mutant from the original program is known as a mutation operator.[1] Table 2

1. In the literature of Mutation Testing, mutation operators are also known as mutant operators, mutagenic operators, mutagens, and mutation rules [191].

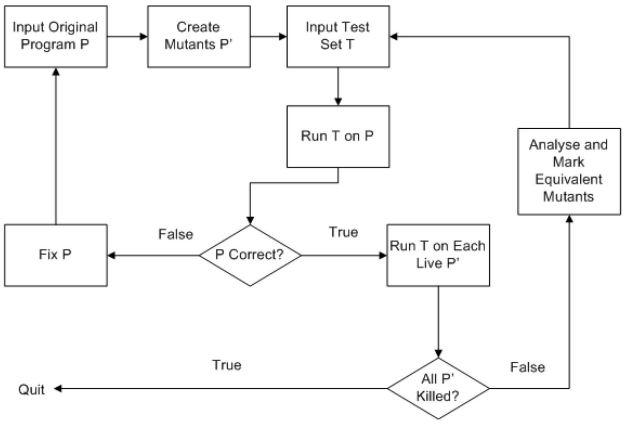

Fig. 2. Generic process of mutation analysis [191].

contains only one example of a mutation operator; there are many others. Typical mutation operators are designed to modify variables and expressions by replacement, insertion, or deletion operators. Table 3 lists the first set of formalized mutation operators for the Fortran programming language. These typical mutation operators were implemented in the Mothra mutation system [131].

To increase the flexibility of Mutation Testing in practical applications, Jia and Harman introduced a scripting language, the Mutation Operator Constraint Script (MOCS) [123]. 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. [217] also proposed a transformation language, MUDEL, used to specify the description of mutation operators. Besides modifying program source, mutation operators can also be defined as rules to modify the grammar used to capture the syntax of a software artifact. A much more detailed account of these grammar-based mutation operators can be found in the work of Offutt et al. [177].

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

After all test cases have been executed, there may still be a few "surviving" mutants. To improve the test set T, the program tester can provide additional test inputs to kill these surviving mutants. However, there are some mutants that can never be killed because they always produce the same output as the original program. These mutants are called Equivalent Mutants. They are syntactically different but functionally equivalent to the original program. Automatically detecting all equivalent mutants is impossible [35], [187] 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 4.

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 nonequivalent 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.

TABLE 2
An Example of Mutation Operation

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

TABLE 3
The First Set of Mutation Operators: The 22 "Mothra" Fortran Mutation Operators (Adapted from [131])

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

2.3 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 [247] and the equivalent mutant problem [35].

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 set of inputs has been arrived at, there remains the problem of checking output

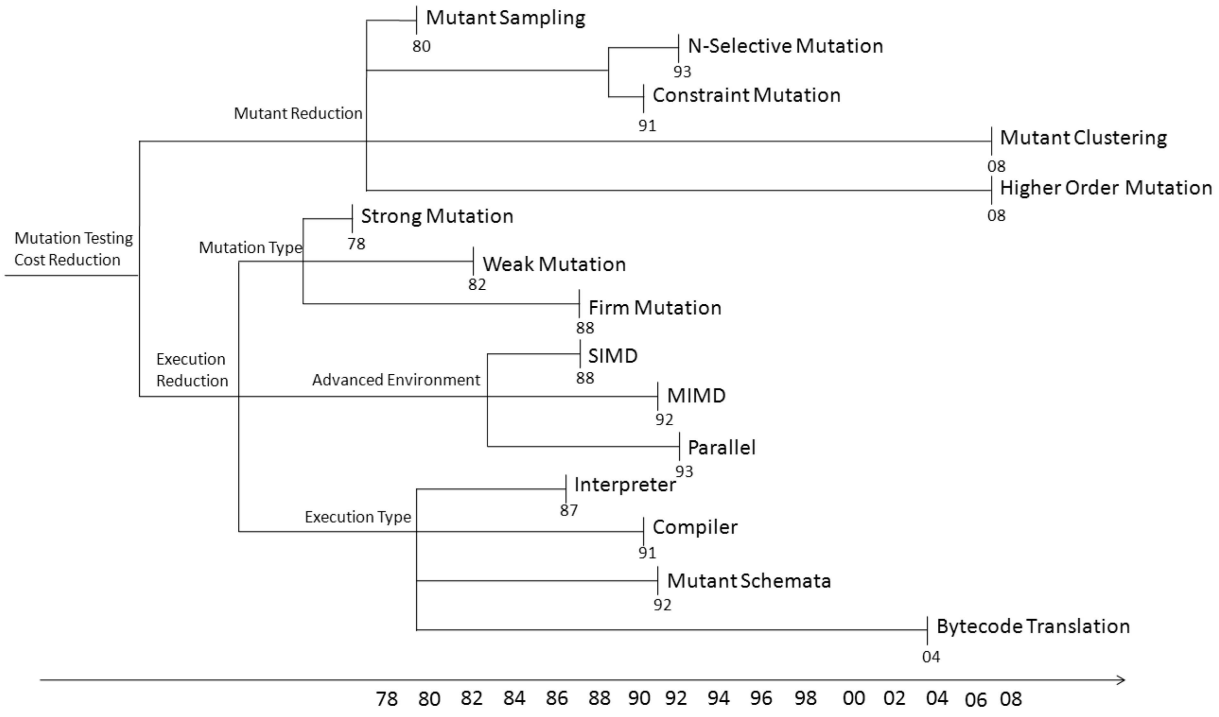

Fig. 3. Overview of the chronological development of mutant reduction techniques.

[247]. 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 typically 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 runtime 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.

3 Cost Reduction Techniques

Mutation Testing is widely believed to be a computationally expensive testing technique. However, this belief is partly based on the outdated assumption that all mutants in the traditional Mothra set need to be considered. In order to turn Mutation Testing into a practical testing technique, many cost reduction techniques have been proposed. In the survey work of Offutt and Untch [191], cost reduction techniques are divided into three types: "do fewer," "do faster," and "do smarter." In this paper, these techniques are classified into two types, reduction of the generated mutants (which corresponds to "do fewer") and reduction of the execution cost (which combines do faster and do smarter). Fig. 3 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 Fig. 4). From this figure, it is clear that Selective Mutation and Weak Mutation are the most widely studied cost reduction techniques. Each of the other techniques is studied in no more than five papers to date. The rest of the section will introduce each cost reduction technique in detail. Section 3.1 will present work on mutant reduction techniques, while Section 3.2 will cover execution reduction techniques.

3.1 Mutant Reduction Techniques

One of the major sources of computational cost in Mutation Testing is the inherent running cost in executing the large

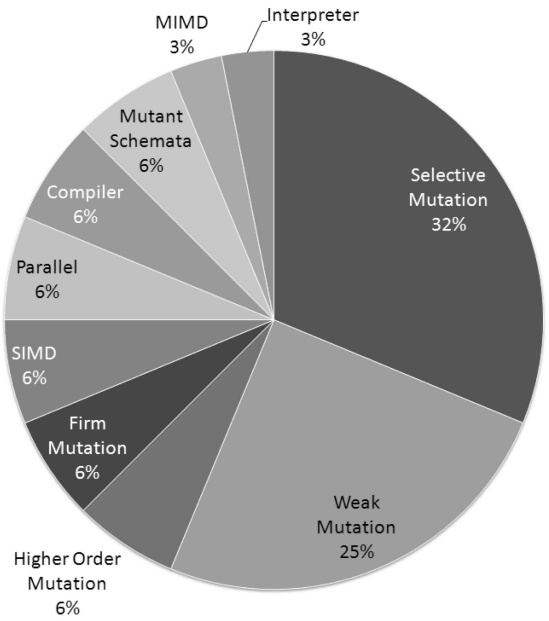

Fig. 4. Percentage of publications using each mutant reduction technique.

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

3.1.1 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 [34]. In this approach, all possible mutants are generated first as in traditional Mutation Testing. x percent 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 [159], [248], the authors conducted an experiment using a random selection rate x percent from 10 to 40 percent in steps of 5 percent. The results suggested that random selection of 10 percent of mutants is only 16 percent less effective than a full set of mutants in terms of mutation score. This study implied that Mutant Sampling is valid with a x percent value higher than 10 percent. This finding also agreed with the empirical studies by DeMillo et al. [64] and King and Offutt [131]. Instead of fixing the sample rate, Sahinoglu and Spafford [207] proposed an alternative sampling approach based on the Bayesian sequential probability ratio test (SPRT). In their approach, the mutants are randomly selected until a statistically appropriate sample size has been reached. The result suggested that their model is more sensitive than the random selection because it is self-adjusting based on the available test set.

3.1.2 Mutant Clustering

The idea of Mutant Clustering was first proposed in Hussain's master's thesis [116]. 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. [120]. Ji et al. use a domain reduction technique to avoid the need to execute all mutants.

3.1.3 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. [190] subsequently extended this idea, calling it Selective Mutation.

Mutation operators generate different numbers of mutants and some mutation operators generate far more mutants than others, many of which may turn out to be redundant. For example, two mutation operators of the 22 Mothra operators, ASR and SVR, were reported to generate approximately 30 to 40 percent of all mutants [131]. 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. [190].

Offutt et al. [190] 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 percent with a 24 percent reduction in the number of mutants reduced. 4-selective mutation achieved a mean mutation score of 99.84 percent with a 41 percent reduction in the number of mutants. 6-selective mutation achieved a mean mutation score of 88.71 percent with a 60 percent reduction in the number of mutants.

Wong and Mathur adopted another type of selection strategy, selection based on test effectiveness [248], [252], known as constraint mutation. Wong and Mathur suggested using only two mutation operators: ABS and RAR. The motivation for the ABS operator is that killing the mutants generated from ABS requires test cases from different parts of the input domain. The motivation for the ROR operator is that killing the mutants generated from ROR requires test cases which "examine" the mutated predicate [248], [252]. Empirical results suggest that these two mutation operators achieve an 80 percent reduction in the number of mutants and only 5 percent reduction in the mutation score in practice.

Offutt et al. [182] extended their six-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 five operators from the operands and expressions class became the key operators. These five operators are ABS, UOI, LCR, AOR, and ROR. These key operators achieved 99.5 percent mutation score.

Mresa and Bottaci [167] 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. [19] defined a guideline for selecting a sufficient set of mutation operators from all possible mutation operators. They applied this guideline to Proteum's 77 C mutation operators [6] and obtained a set of 10 selected mutation operators, which achieved a mean mutation score of 99.6 percent with a 65.02 percent 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 and Andrews [168], [169], [170]. 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 percent of all generated mutants. According to their results, this approach achieved the highest rate of reduction compared with other approaches.

3.1.4 Higher Order Mutation

Higher Order Mutation is a comparatively new form of Mutation Testing introduced by Jia and Harman [122]. The underlying motivation was 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 (FOMs) and higher order mutants (HOMs). FOMs are created by applying a mutation operator only once. HOMs 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.

This idea has been partly proved by Polo et al.'s work [199]. 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 suggest that applying second order mutants reduced test effort by approximately 50 percent, without much loss of test effectiveness. More recently, Langdon et al. have applied multi-object genetic programming to the generation of higher order mutants [136], [137]. In their experiment, they have found realistic higher order mutants that are harder to kill than any first order mutant.

3.2 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 types of techniques used to optimize the execution process that have been considered in the literature.

3.2.1 Strong, Weak, and Firm Mutation

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

Strong Mutation is often referred to as traditional Mutation Testing. That is, it is the formulation originally proposed by DeMillo et al. [66]. 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 [115] proposed Weak Mutation. In Weak Mutation, a program p is assumed to be constructed from a set of components $C = \{c_1, \ldots, c_n\}$. Suppose mutant m is made by changing component c_m; mutant m is said to be killed if any execution of component c_m is different from mutant m. As a result, in Weak Mutation, instead of checking mutants after the execution of the entire program, the mutants need only to be checked immediately after the execution point of the mutant or mutated component.

In Howden's work [115], the component C referred to one of the following five types: variable reference, variable assignment, arithmetic expression, relational expression, and boolean expression. This definition of components was later refined by Offutt and Lee [183], [184]. Offutt and Lee defined four types of execution: evaluation after the first execution of an expression (Ex-Weak/1), the first execution of a statement (St-Weak/1), the first execution of a basic block (BB-Weak/1), and after N iterations of a basic block in a loop ($\text{BB-Weak}/N$).

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 [253]. However, as different components of the original program may give different outputs from the original execution, weak mutation test sets can be less effective than strong mutation test sets. In this way, weak mutation sacrifices test effectiveness for improvements in test effort. This raises the question as to what kind of trade-off can be achieved.

There were many empirical studies on the Weak Mutation trade-off. Girgis and Woodward [103] implemented a weak mutation system for Fortran 77 programs. Their system is an analytical type of weak mutation system in which the mutants are killed by examining the program's internal state. In their experiment, four of Howden's five program components were considered. The results suggested that weak mutation is less computationally expensive than strong mutation. Marick [153] drew similar conclusions from his experiments.

A theoretical proof of Weak Mutation by Horgan and Mathur [113] showed that under certain conditions, test sets generated by weak mutation can also be expected to be as effective as strong mutation. Offutt and Lee [183], [184] presented a comprehensive empirical study using a weak mutation system named Leonardo. In their experiment, they used the 22 Mothra mutation operators as fault models

instead of Howden's five component set. The results from their experiments indicated that Weak Mutation is an alternative to Strong Mutation in most common cases, agreeing with the probabilistic results of Horgan and Mathur [113] and experimental results of Girgis and Woodward [103] and Marick [153].

Firm Mutation was first proposed by Woodward and Halewood [257]. 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 [119] proposed a parallel Firm Mutation approach for Java programs. Unfortunately, to date there is no publicly available firm mutation tool.

3.2.2 Runtime Optimization Techniques

The Interpreter-Based Technique is one of the optimization techniques used in the first generation of Mutation Testing tools [131], [181]. In traditional Interpreter-Based Techniques, the result of a mutant is interpreted from its source code directly. The main cost of this technique is determined by the cost of interpretation. To optimize the traditional Interpreter-Based approach, Offutt and King [131], [181] translated the original program into an intermediate form. Mutation and interpretation are performed at this intermediate code level. Interpreter-Based tools provide additional flexibility and are sufficiently efficient for mutating small programs. However, due to the nature of interpretation, it becomes slower as the scale of programs under test increases.

The Compiler-Based Technique is the most common approach to achieve program mutation [52], [53]. In a Compiler-Based Technique, each mutant is first compiled into an executable program; the compiled mutant is then executed by a number of test cases. Compared to source code interpretation techniques, this approach is much faster because execution of compiled binary code takes less time than interpretation. However, there is also a speed limitation, known as compilation bottleneck, due to the high compilation cost for programs whose runtime is much longer than the compilation/link time. [47].

DeMillo et al. proposed the Compiler-Integrated Technique [65] to optimize the performance of the traditional Compiler-Based Techniques. Because there is only a minor syntactic difference between each mutant and the original program, compiling each mutant separately in the Compiler-Based technique will result in redundant compilation cost. In the Compiler-Integrated technique, an instrumented compiler is designed to generate and compile mutants.

The instrumented compiler generates two outputs from the original program: an executable object code for the original program and a set of patches for mutants. Each patch contains instructions which can be applied to convert the original executable object code image directly to executable code for a mutant. As a result, this technique can effectively reduce the redundant cost from individual compilation. A much more detailed account can be found in the Krauser's PhD thesis [132].

The Mutant Schema Generation approach is also designed to reduce the overhead cost of the traditional interpreter-based techniques [235], [236], [237]. 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 [237] 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 Untch's PhD dissertation [236]. 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 was first proposed by Ma et al. [151], [185]. 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], [185], [208]. 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 of the mutation operators can be represented at the Bytecode level [208].

Bogacki and Walter introduced an alternative approach to reduce compilation cost, called Aspect-Oriented Mutation [26], [27]. In their approach, an aspect patch is generated to capture the output of a method on the fly. Each aspect patch will run programs twice. The first execution obtains the results and context of the original program and mutants are generated and executed in the second execution. As a result, there is no need to compile each mutant. Empirical evaluation between a prototype tool and Jester can be found in the work of Bogacki and Walter [26].

3.2.3 Advanced Platforms Support for Mutation Testing

Mutation Testing has also been applied to many advanced computer architectures to distribute the overall computational cost among many processors. In 1988, Mathur and Krauser [158] were the first to perform Mutation Testing on a vector processor system. Krauser et al. [133], [134] proposed an approach for concurrent execution mutants under SIMD machines. Fleyshgakker and Weiss [92], [246] proposed an algorithm that significantly improved techniques for parallel Mutation Testing. Choi and Mathur [47] and Offutt et al. [189] have distributed the execution cost of Mutation Testing through MIMD machines. Zapf [261] extended this idea in a network environment, where each mutant is executed independently.

TABLE 4
An Example of Equivalent Mutation

Program p	Equivalent Mutant m

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

4 Equivalent Mutant Detection Techniques

To detect if a program and one of its mutants programs are equivalent is undecidable, as proved in the work of Budd and Angluin [35]. As a result, the detection of equivalent mutants alternatively may have to be carried out by humans. This has been a source of much theoretical interest. For a given program p, m denotes a mutant of program p. Recall that m is an equivalent mutant if m is syntactically different from p, but has the same behavior as p. Table 4 shows an example of equivalent mutant generated by changing the operator $<$ of the original program into the operator $!=$. If the statements within the loop do not change the value of i, program p and mutant m will produce identical output.

An equivalent mutant is created when a mutation leads to no possible observable change in behavior; the mutant is syntactically different but semantically identical to the original program from which it is created. Grün et al. [106] manually investigated eight equivalent mutants generated from the JAXEN XPATH query engine program. They pointed out four common equivalent mutant situations: The mutant is generated from dead code, the mutant improves speed, the mutant only alters the internal states and the mutant cannot be triggered (i.e., no input test data can change the program's behavior at the mutation point). It is worth noticing that these four are not the only situations that lead to equivalent mutants. For example, none of it applies to the example in Table 4.

As the mutation score is counted based on nonequivalent mutants without a complete detection of all equivalent mutants, the mutant score can never be 100 percent, 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 percent of mutants which are equivalent [178], [187]. Fortunately, there has been much research work on the detection of the equivalent mutants.

Baldwin and Sayward [18] 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 "optimization" or a "deoptimization process." Baldwin and Sayward [18] proposed six types of compiler optimization rules that can be used for the detection of equivalent mutants. These six were implemented and empirically studied by Offutt and Craft [178]. The empirical results showed that, generally, 10 percent of all mutants were equivalent mutants for 15 subject programs.

Based on the work of constraint test data generation, Offutt and Pan [186], [187], [197] 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 analyzing the path condition of a mutant. A mutant is equivalent if and only if the input constraint is unsatisfiable. Empirical evaluation of a prototype has shown that this technique is able to detect a significant percentage of equivalent mutants (47.63 percent among 11 subject programs) for most of the programs. Their results suggest that the constraint satisfaction formulation is more powerful than the compiler optimization technique [178].

The program slicing technique has also been proposed to assist in the detection of equivalent mutants [109], [110], [241]. Voas and McGraw [241] were the first to suggest the application of program slicing to Mutation Testing. Hierons et al. [110] demonstrated an approach using slicing to assist the human analysis of equivalent mutants. This is achieved by the generation of a sliced program that denotes the answer to an equivalent mutant. This work was later extended by Harman et al. [109] using dependence analysis.

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

Ellims et al. [83] reported that mutants with syntactic difference and the same output can be also semantically different in terms of running profile. These mutants often have the same output as the original programs but have different execution time or memory usage. Ellims et al. suggested that "resource-aware" might be used to kill the potential mutants.

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.

5 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) [60] 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. 5 shows the chronological development of research work on Program Mutation and Specification Mutation. Fig. 6 shows the percentage of the publications addressing each language to which Mutation Testing has been applied.

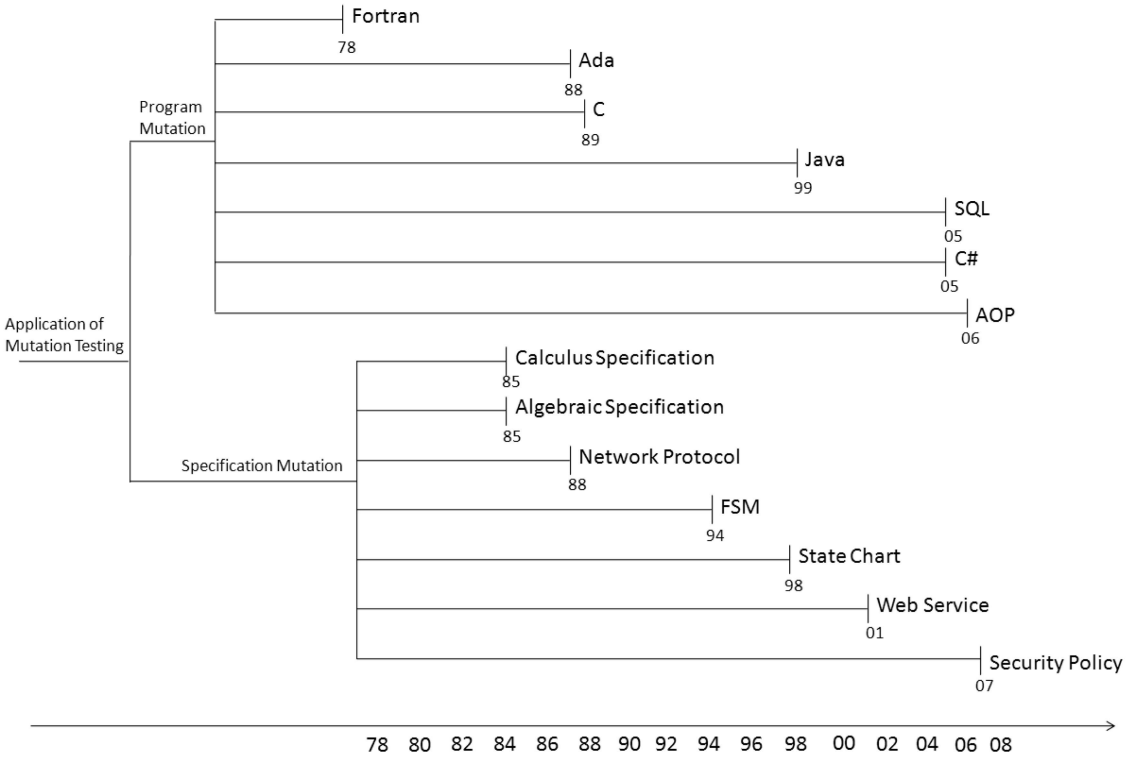

Fig. 5. Publications of the applications of mutation testing.

As Fig. 5 shows, there has been more work on Program Mutation than Specification Mutation. Notably more than 50 percent of the work has been applied to Java, Fortran, and C. Fortran features highly because a lot of the earlier work on Mutation Testing was carried out on Fortran programs. In the following section, the applications of Program Mutation and Specification Mutation are summarized by the programming language targeted.

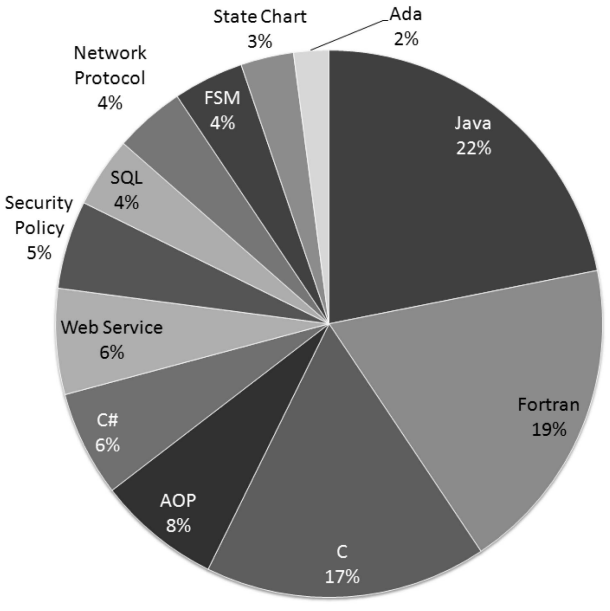

Fig. 6. Percentage of publications addressing each language to which Mutation Testing has been applied.

5.1 Program Mutation

Program Mutation has been applied to both the unit level [66] and the integration level [55] 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 [240]. Applying Program Mutation at the integration level is also known as Interface Mutation, which was first introduced by Delamaro et al. [55] in 1996. Interface Mutation has been applied to C programs by Delamaro and Maldonado [54], Delamaro et al. [55], [56] and also to CORBA programs by Ghosh [98], Ghosh et al. [100], Ghosh and Mathur [101], [102]. Empirical evaluations of Interface Mutation can be found in Vincenzei et al.'s work [240] and Delamaro et al.'s work [57], [58].

5.1.1 Mutation Testing for Fortran

In the earliest days of Mutation Testing, most of the experiments on Mutation Testing targeted Fortran. Budd et al. [36], [40] were 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], [36], [145]. However, there were no formal definitions of mutation operators for Fortran until 1987. In 1987, Offutt and King [131], [181] summarized the results from previous work and proposed 22 mutation operators for Fortran 77. This set of mutation operators became the first set of formalized mutation operators and consequently had greater influence on later definitions of mutation operators for applying Mutation Testing to the

other programming languages. These mutation operators are divided into three groups; the Statement analysis group, the Predicate analysis group, and the Coincidental correctness group.

5.1.2 Mutation Testing for Ada

Ada mutation operators were first proposed by Bowser [29] in 1988. In 1997, based on previous work of Bowser's Ada mutation operators [29], Agrawal et al.'s C mutation operators [6], and the design of Fortran 77 mutation operators for MOTHRA [131], Offutt et al. [192] redesigned mutation operators for Ada programs to produce a proposed set of 65 Ada mutation operators. According to the semantics of Ada, this set of Ada mutation operators is divided into five groups: the Operand Replacement Operators group, Statement Operators group, Expression Operators group, Coverage Operators group, and Tasking Operators group.

5.1.3 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 and Maldonado [54], Delamaro et al. [55], [56], [58] investigated the application of Mutation Testing at the integration level. They selected 10 mutation operators from Agrawal et al.'s 77 mutation operators to test interfaces of C programs. These mutation operators focus on injecting faults into the signature of public functions. More recently, Higher Order Mutation Testing has also been applied to C programs by Jia and Harman [122].

There are also mutation operators that target specific C program defects or vulnerabilities. Shahriar and Zulkernine [214] proposed 8 mutation operators to generate mutants that represent Format String Bugs (FSBs). Vilela et al. [239] proposed 2 mutation operators representing faults associated with static and dynamic memory allocations, which were used to detect Buffer Overflows (BOFs). This work was subsequently extended by Shahriar and Zulkernine [213], who proposed 12 comprehensive mutation operators to support the testing of all BOF vulnerabilities, targeting vulnerable library functions, program statements, and buffer size. Ghosh et al. [97] have applied Mutation Testing to an Adaptive Vulnerability Analysis (AVA) to detect BOFs.

5.1.4 Mutation Testing for Java

Traditional mutation operators are not sufficient for testing Object-Oriented (OO) programming languages like Java [130], [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. [128] were the first to design mutation operators for the Java programming language. They proposed 20 mutation operators for Java using Hazard and Operability Studies (HAZOP). 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. [127] introduced Class Mutation, which applies mutation to OO (Java) programs targeting faults related to OO-specific features. In Class Mutation, 3 mutation operators representing Java OO-features were selected from the 20 Java mutation operators. In 2000, Kim et al. [129] 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 [130]. A similar approach was also adopted by Chevalley and Thevenod-Fosse in their work [44], [45].

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 [127]). They suggested that the selected mutation operators should be obtained from empirical results of the effectiveness of all mutation operators. Therefore, instead of continuing Kim et al.'s work [129], Ma et al. [150] proposed 24 comprehensive Java mutation operators based on previous studies of OO Fault models. These are classified into six groups: the 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 class mutation (Reference assignment and content assignment replacement) and the EOC class mutation (reference comparison and content comparison replacement) cannot 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], [24] 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, Bradbury et al. [31] introduced an extension to the concurrent Java environment.

5.1.5 Mutation Testing for C#

Based on previous proposed Java mutation operators, Derezińska introduced an extension to a set of C# specialized mutation operators [70], [71] and implemented them in a C# mutation tool named CREAM [72]. Empirical results for this set of C# mutation operators using CREAM were reported by Derezińska [71], [73].

5.1.6 Mutation Testing for SQL

Mutation Testing has also been applied to SQL code to detect faults in database applications. The first attempt to

design mutation operators for SQL was done by Chan et al. [43] in 2005. They proposed 7 SQL mutation operators based on the enhanced entity-relationship model. Tuya et al. [234] proposed another set of mutant operators for SQL query statements. This set of mutation operators is organized into four categories, including mutation of SQL clauses, mutation of operators in conditions and expressions, mutation handling NULL values, and mutation of identifiers. They also developed a tool named SQLMutation that implements this set of SQL mutation operators and an empirical evaluation concerning results using SQLMutation [233]. A development of this work targeting Java database applications can be found in the work of Zhou and Frankl [264]. Shahriar and Zulkernine [212] have also proposed a set of mutation operators to handle the full set of SQL statements from connection to manipulation of the database. They introduced 9 mutation operators and implemented them in an SQL mutation tool called MUSIC.

5.1.7 Mutation Testing for Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm that aids programmers in separation of cross-cutting concerns. Ferrari et al. [90] 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 [90]. 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 [51].

AspectJ is a widely studied aspect-oriented extension of the Java language, which provides many special constructs such as aspects, advice, join points, and pointcuts [13]. Baekken and Alexander [17] summarized previous research work on the fault model associated with AspectJ pointcuts. They proposed a complete AspectJ fault model based on the incorrect pointcut pattern, which was used as a set of mutation operators for AspectJ programs. Based on this work, Anbalagan and Xie [12], [13] proposed a framework to generate mutants for pointcuts and to detect equivalent mutants. To reduce the total number of mutants, a classification and ranking approach based on the strength of the pointcuts was also introduced in their framework.

5.1.8 Other Program Mutation Applications

Besides these programming languages, Mutation Testing has also been applied to Lustre programs [80], [81], PHP programs [215], Cobol programs [108], Matlab/Simulink [262], and spreadsheets [1]. There is also research work investigating the design of mutation operators for real-time systems [96], [171], [172], [227] and concurrent programs [8], [31], [41], [99], [147].

5.2 Specification Mutation

Although Mutation Testing was originally proposed as a white box testing technique at the implementation level, it has also been applied at the software design level. Mutation Testing at design level is often referred to as "Specification Mutation," which was first introduced by Gopal and Budd in 1983 [38], [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 [195].

5.2.1 Mutation Testing for Formal Specifications

The formal specifications can be presented in many forms, for example, calculus expressions, Finite State Machines (FSMs), Petri Nets, and Statecharts. The earlier research work on Specification Mutation considered specifications of simple logical expressions. Gopal and Budd [38], [105] considered specifications in predicate calculus targeting the predicate structure of the program under test. A similar work applied to the refinement calculus specification can be found in the work of Aichernig [7]. Woodward [254], [257] 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 [255].

More recently, many formal techniques have been proposed to specify the dynamic aspects of a software system, for example, FSMs, Petri Nets, and Statecharts. Fabbri et al. [88] 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 [85]. An empirical evaluation of these mutation operators was reported by them [85]. Hierons and Merayo [111], [112] 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. [20], Bombieri et al. [28] and Belli et al. [23].

Statecharts are widely used for the formal specification of complex reactive systems. Statecharts can be considered as an extension of FSMs, so the first set of mutation operators for Statecharts was also proposed by Fabbri et al. [87], based on their previous work on FSM mutation operators. Using Fabbri et al.'s Statecharts mutation operators, Yoon et al. [260] introduced a new test criterion, the 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 implementation level as well as the model level. Other work on Statechart mutation can be found in the work of Fraser and Wotawa [95].

Besides FSMs and Statecharts, Specification Mutation has been also applied to a variety of specification languages. For example, Souza et al. [222], [223] investigated the application of Mutation Testing to the Estelle Specification language. Fabbri et al. [86] proposed mutation operators for Petri Nets. Srivatanakul et al. [225] performed an empirical study using Specification Mutation to CSP Specifications. Olsson and Runeson [196] and

Sugeta et al. [226] proposed mutation operators for SDL. Definitions of mutation operators for formal specification language can be found in the work of Black et al. [25] and the work of Okun [195].

5.2.2 Mutation Testing for Running Environment

During the process of implementing specifications, bugs might be introduced by programmers due to insufficient knowledge of the final target environment. These bugs are called "environment bugs" and they can be hard to detect. Examples are the bugs caused by memory limitations, numeric limitations, value initialization, constant value interpretation, exception handling, and system errors [224]. Mutation Testing was first applied to the detection of such bugs by Spafford [224] in 1990. In his work, environment mutants were generated to detect integer arithmetic environmental bugs.

The idea of environment bugs was extended in the 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" [78]. As a result, Mutation Testing was also applied to the validation of security vulnerabilities. Du and Mathur [78] 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 [79].

5.2.3 Mutation Testing for Web Services

Lee and Offutt [142] were the first to apply Mutation Testing to Web Services. In 2001, they introduced an Interaction Specification Model to formalize the interactions between web components [142]. Based on this specification model, a set of generic mutation operators was proposed to mutate the XML data model. This work was later extended by Offutt and Xu [193] and Xu et al. [259] targeting the mutation of XML data and they renamed it XML perturbation. Instead of mutating XML data directly, they perturbed XML schemas to create invalid XML data using seven XML schema mutation operators. A constraint-based test case generation approach was also proposed and the results of empirical studies were reported [259]. Another set of XML schema mutation operators was proposed by Li and Miller [143].

There is also Web Service mutation work targeting specific XML-based language features, for example, the OWL-S specification language [140], [245] and WS-BPEL specification language [84]. 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. [140], the authors propose mutation operators for detection of semantic errors caused by the misuse of the ontology classes.

5.2.4 Mutation Testing for Networks

Protocol robustness is an important aspect of any network system. Sidhu and Leung [216] investigated fault coverage of network protocols. Based on this work, Probert and Guo proposed a set of mutation operators to test network protocols [202]. Vigna et al. [238] applied Mutation Testing to network-based intrusion detection signatures, which are used to identify malicious traffic. Jing et al. [124] built an 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. [263].

5.2.5 Mutation Testing for Security Policy

Mutation Testing has also been applied to security policies [139], [154], [165], [166], [201]. Much of this research work sought to design mutation operators that inject common flaws into different types of security policies. For example, Martin and Xie [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. [166]. Le Traon et al. [139] introduced eight mutation operators for the Organization-Based Access Control OrBAC policy. Mouelhi et al. [165] proposed a generic metamodel for security policy formalisms. Based on this formalism, a set of mutation operators was introduced to apply to all rule-based formalisms. Hwang et al. proposed an approach that applies Mutation Testing to test firewall policies [117].

5.3 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 summarize the main work on mutation as support to these testing activities.

5.3.1 Test Data Generation

The main idea of mutation-based test data generation is to generate test data that can effectively kill mutants. 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 [194]. 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 constraint. Test data that guarantee 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 [67] under the Mothra system. Godzilla applied control-flow analysis, symbolic evaluation, and a constraint satisfaction technique to generate and solve constraints for each mutant. Empirical results suggest that 90 percent of mutants can be killed using the CBT technique for most programs [68]. However, the CBT technique also suffers from some of the drawbacks associated with symbolic evaluation. Offutt et al. [179], [180] 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 [22]. 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

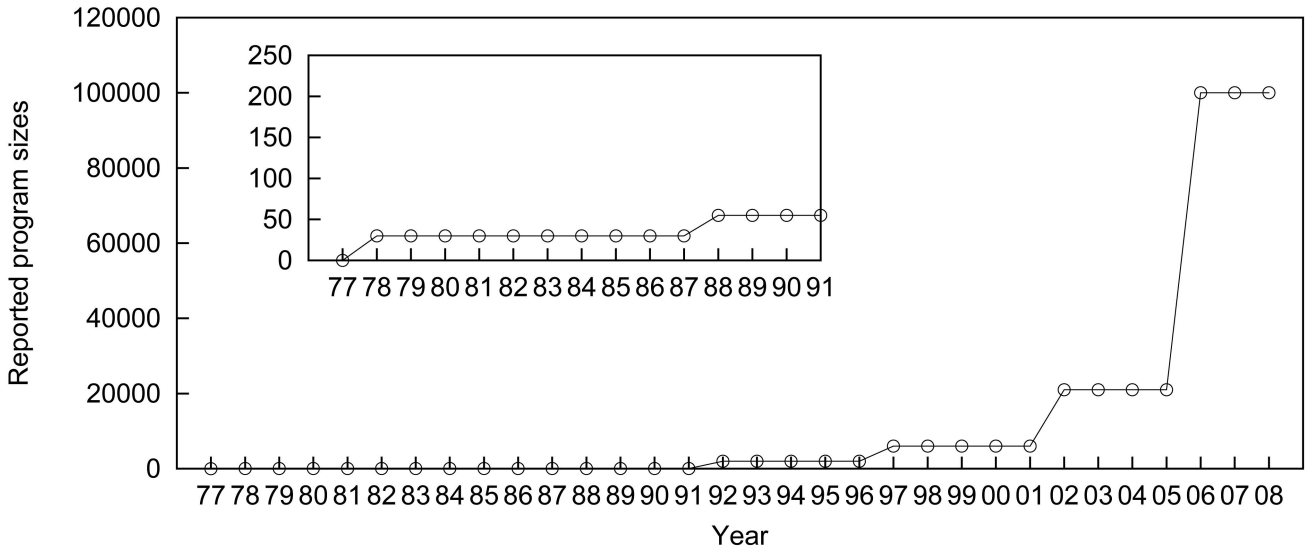

Fig. 7. The largest program applied for each year.

test data. The generated test datum 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 percent of mutants 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. [21] proposed an approach to improve the quality of test data using Mutation Testing with a Bacteriological Algorithm. Smith and Williams applied Mutation Testing as guidance to test data augmentation [219]. Le Traon et al. [138] use mutation analysis to improve component contract. Xie et al. [258] applied Mutation Testing to assist programmers in writing parameterized unit tests.

5.3.2 Regression Testing

Test case prioritization techniques are one way to assist regression testing. Mutation Testing has been applied as a test case prioritization technique by Do and Rothermel [75], [76]. Do and Rothermel 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. [173] 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 percent without loss of test effectiveness.

In addition to the previously 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 [210], [211]. Mutation analysis has also been used to assist the evaluation of software clone detection tools [204], [205].

6 Empirical Evaluation

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

6.1 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 9 (Table 9 is located in the end of the paper). Table 9 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 9 is sorted by the number of papers that use the subject program, so the first 10 programs are the most studied subject programs in the literature on Mutation Testing. These wildly studied programs are all laboratory programs under 50 LoC, but we also noticed that the 11th program is SPACE, a nontrivial real program.

To provide an overview of the trend of empirical studies on Mutation Testing to attack more challenging programs, we calculated the size of the largest subject program for each year. For each year on the horizontal axis, the data point in Fig. 7 shows the size of the largest program considered in a mutation study up to that point in time. Clearly, the definition of "program size" can be problematic, so the figure is merely intended to be used as a rough indicator. There is evidence to indicate that the size of the subject programs that can be handled by Mutation Testing is increasing. However, caution is required. We found that although some empirical experiments were reported to

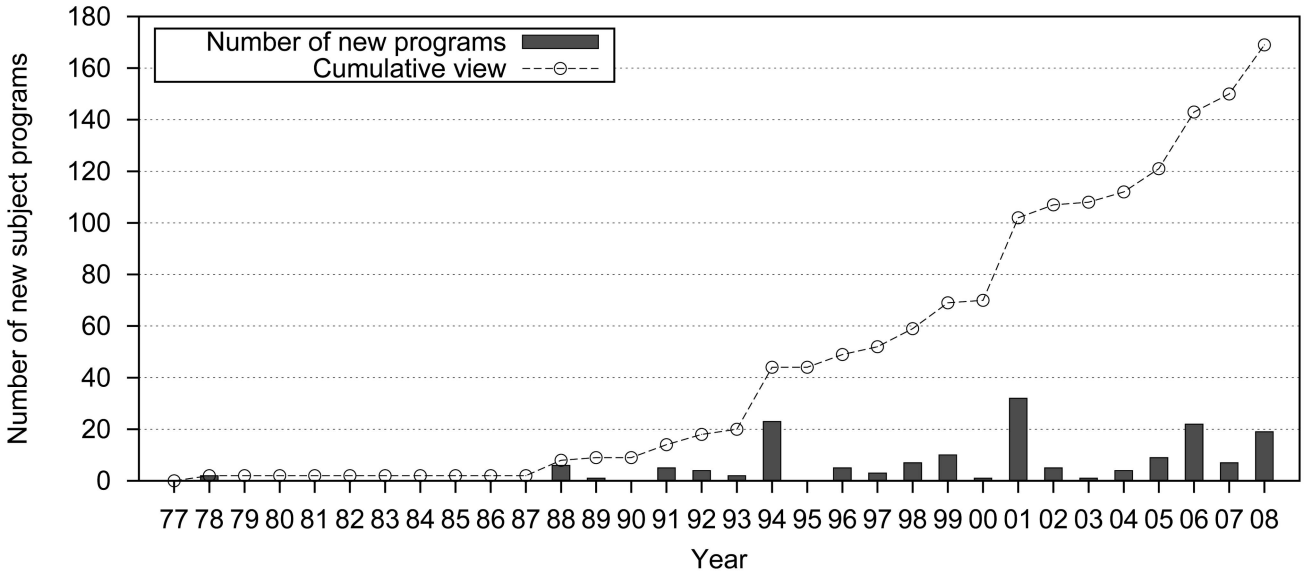

Fig. 8. New programs applied for each year.

handle large programs, some studies applied only a few mutation operators. We also counted the number of newly introduced subject programs for each year. The results are shown in Fig. 8. The dashed line in the figure is the cumulative view of the results. The number of newly used subject programs is gradually increasing, which suggests a growth in practical work.

In the empirical studies, it may be more indicative to use a real-world program rather than laboratory program. To understand the relationship between the use of laboratory programs and real-world programs in mutation experiments, we have counted each type by year. The results are shown in Fig. 9. In this study, we consider a real-world program to be either an open source or an industry program. In Fig. 9, 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. Fig. 9 also shows the number of laboratory and real programs introduced into studies each year as bars. This clearly indicates that, while there are correctly more laboratory programs overall, since 2002 far more new real programs than laboratory programs have been introduced. This finding provides some evidence to support the claim that the development of Mutation Testing is maturing.

In our study, we found that for each research area of Mutation Testing, there is a different set of subject programs used as benchmarks. In Table 5, we have summarized these benchmark programs. We chose five active research areas based on our studies: Coupling effect, Selective Mutation, Weak, Strong, and Firm Mutation, Equivalent Mutant Detection, and experiments supporting testing, including

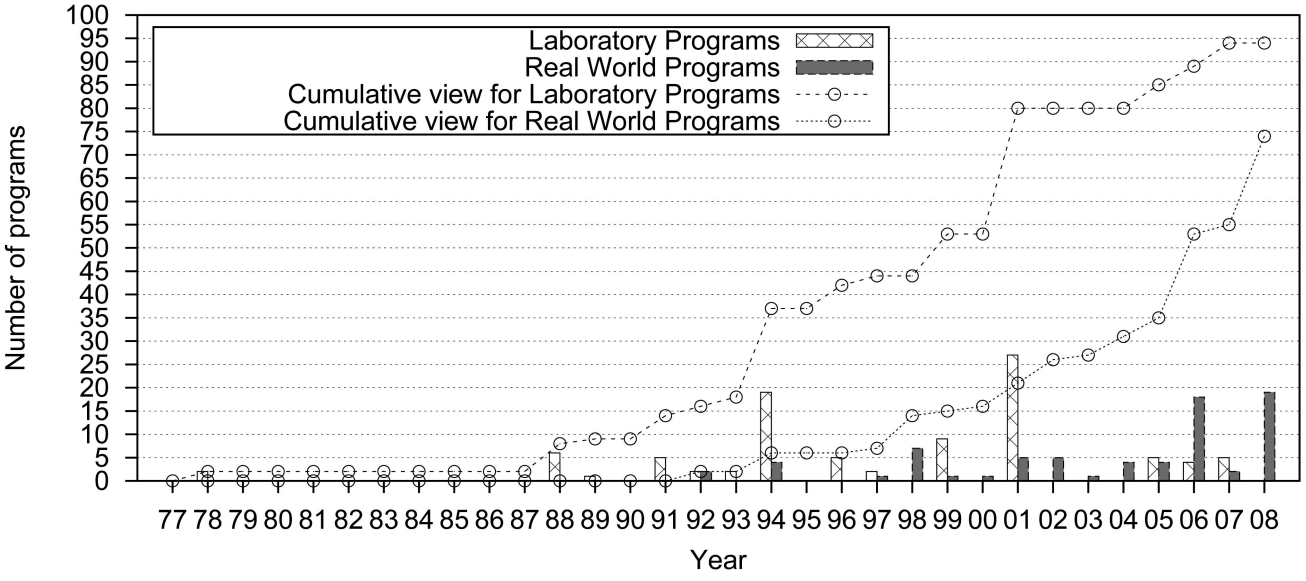

Fig. 9. Laboratory programs versus real programs.

TABLE 5
Subject Programs by Application

Application	Subject Programs	Reference
Coupling Effect	Triangle, Find, MID	[174], [175]
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	[19], [167], [168], [170], [182], [190]
Weak, Strong, Firm Mutation	Triangle, Find, Bubble, MID, Calendar, Euclid, Quad, Insert, Warshall, Pat, Gcd, Sort, Max_index	[183], [184], [257]
Equivalent Mutant	Triangle, Find, Bubble, MID, Calendar, Euclid, Quad, Insert, Warshall, Pat, Bsearch, Max, Banker, Deadlock, Count, Dead	[178], [186], [187]
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	[16], [67], [68], [75], [76], [114], [146], [173], [179], [180], [250]

TABLE 6
Empirical Evaluation of Mutation Testing

Research	Evaluation Type	Subject Programs
DeMillo and Mathur [61]	real faults vs mutants	Tex
Mathur and Wong [160], [248]	all-use vs mutation criteria	Find, Strmat1, Strmat2 and Textfmt
Offutt et al. [188]	all-use vs mutation criteria	Bub, Cal, Euclid, Find, Insert, Mid, Pat, Quad, Trityp and Warshall
Daran and Thévenod-Fosse [50]	real faults vs mutants	Nuclear Reactor Safety Shutdown System
Frankl et al. [93], [94]	all-use vs mutation criteria	Determinant, Find1, Find2, Matinv1, Matinv2, Strmatch1, Strmatch2, Textformat.r and Transpose
Andrews et al. [14]	hand seeded faults vs mutants	Space, Printtokens, Printtokens2, Replace, Schedule, Schedule2, Tcas and Totinfo
Do and Rothermel [75], [76]	hand seeded faults vs mutants	Ant, Xml-security, Jmeter, Jtopas, galileo and nanoxml

the use of mutation analysis to select, minimize, prioritize, and generate test data.

6.2 Empirical Results

Many researchers have conducted experiments to evaluate the effectiveness of Mutation Testing [14], [50], [61], [93], [94], [160], [188], [248]. 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 6 summarizes the evaluation type and the subject programs used in each of these experiments.

Mathur and Wong have conducted experiments to compare the "all-use" criterion with mutation criteria [160], [248], [251]. In their experiment, Mathur and Wong manually generated 30 sets of test cases satisfying each criterion for each subject program. Empirical results suggested that mutation adequate test sets more easily satisfy the "all-use" criteria than all-use test sets satisfy mutation criteria. This result indicates mutation criteria "probsubsumes"[2] the "all-use" criteria in general.

Offutt et al. conduced a similar experiment using 10 different programs [188]. The "cross scoring" result also provides evidence for Mathur and Wong's probsubsumes relationship [160], [248]. In addition to comparing the two criteria with each other, Offutt et al. also compared the two criteria in terms of the fault detection rate. This result showed that 16 percent more faults can be detected using mutation adequate test sets than "all-use" test sets, indicating that mutation criteria is "probbetter"[3] than the "all-use" data flow. This conclusion also agreed with the results of the experiment of Frankl et al. [93], [94].

2. If a test criterion C_1 probsumes a test criterion C_2, a test set which is adequate to C_1 is likely to be adequate to C_2 [188].

3. If a test criterion C_1 probbetter than a test criterion C_2, then a randomly selected test set which satisfies C_1 is more likely to detect a fault than a randomly selected test set which satisfies C_2 [188].

In addition to comparing mutation analysis with other testing criteria, there have also been empirical studies comparing real faults and mutants. In the work of Daran and Thévenod-Fosse [50], the authors conducted an experiment comparing real software errors with first 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 percent of the errors caused by mutants were also produced by real faults, thereby providing evidence for the Mutation Coupling Effect Hypothesis. This result also agreed with DeMillo and Mathur's experiment [61]. DeMillo and Mathur carried out an extensive study of the errors in TeX reported by Knuth [61] and they demonstrated how simple mutants could detect real complex errors from TeX.

Andrews et al. [14] conducted an experiment comparing manually instrumented faults generated by experienced developers with mutants automatically generated by four carefully selected mutation operators. In the experiment, the Siemens suite (Printtokens, Printtokens2, Replace, Schedule, Schedule2, Tcas, and Totinfo) and the Space program were used as subjects. Empirical results suggested that, after filtering out equivalent mutants, the remaining nonequivalent 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 autogenerated faults are needed for the detection of real faults.

Do and Rothermel [75], [76] studied the effect of both hand seeded faults and machine generated mutants on fault detection ability and the test prioritization order. In the test data prioritization study, Do and Rothermel considered several prioritization techniques to improve the fault detection rate. Their analysis showed that for noncontrol test case prioritization, the use of mutation can improve fault detection rates. However, the results are affected by the number of mutation faults applied. In the fault detection ability studies, Do and Rothermel followed Andrews et al.'s experimental procedure [14]. Results from four out of the six subject programs revealed a similar data spread to the work of Andrews et al. The effect of test set minimization using mutation can be found in the work of Wong et al. [249].

Despite evaluating Mutation Testing against other testing approaches, there are also experiments that use mutation analysis to evaluate different testing approaches. For example, Andrews et al. [15] conducted an experiment to compare test data generation using control flow and data flow. Thevenod-Fosse et al. [229] applied mutation analysis to compare random and deterministic input generation techniques. Bradbury et al. [32] used mutation analysis to evaluate traditional testing and model checking approaches on concurrent programs.

7 Tools for Mutation Testing

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

Since the idea of Mutation Testing was first proposed in the 1970s, many mutation tools have been built to support automated mutation analysis. In our study, we have collected information concerning 36 implemented mutation tools, including the academic tools reported in our repository as well as the tools from the open source and the industrial domains. Table 7 summarizes the application, publication time, and any notable characteristics for each tool. The detailed description of the tools can be found in the references cited in the final column of the table.

Fig. 10 shows the growth in the number of tools introduced. In Fig. 10, the development work can be classified into three stages. The first stage was from 1977 to 1981. In this early stage, in which the idea of Mutation Testing was first proposed, four prototype experimental mutation tools were built and used to support the establishment of the fundamental theory of mutation analysis, such as the Competent Programmer Hypothesis [3] and the Coupling Effect Hypothesis [66]. The second stage was from 1982 to 1999. There were four tools built in this period, three academic tools, MOTHRA for Fortran [63], [64], PROTEUM, TUMS for C [52], [53], [236], and one industry tool called INSURE++. Engineering effort had been put into MOTHRA and PROTEUM so that they were able to handle small real programs not just laboratory programs. As a result, these two academic tools were widely used. Most of the advanced mutation techniques were experimented on using these two tools, for example, Weak Mutation [183], [184], Selective Mutation [182], [190], Mutant Sampling [159], [248], and Interface Mutation [54], [55]. 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 Fig. 10, the dashed line shows a cumulative view of this development work. We can see that the tool development trend is rapidly increasing since year 2000, indicating that research work on Mutation Testing remains active and increasingly practical.

In order to explore the impact of Mutation Testing within the open source and industrial domains, we have classified tools into three classes: academic, open sources, and industrial. Table 8 shows the number of each class over two periods; one is before the year 2000, the other is from the year 2000 to the present. As can be seen, there are more open source and industrial tools implemented recently, indicating that Mutation Testing has gradually become a practical testing technique, embraced by both the open source and industrial communities.

8 Evidence for the Increasing Importance of Mutation Testing

To understand the general trend for the Mutation Testing research area, we analyzed the number of publications by year from 1977 to 2009. Consider again the results in Fig. 1; there are five apparent outliers in years 1994, 2001, 2006, 2007, and 2009. The reason for the last four years, is that there were four Mutation Testing workshops held in 2000 (with proceedings published in 2001), 2006, 2007, and 2009. However, there is no direct evidence to explain the spike in

TABLE 7
Summary of Published Mutation Testing Tools

Name	Application	Year	Character	Available	Reference
PIMS	Fortran	1977	General	No	[36], [40], [145]
EXPER	Fortran	1979	General	No	[3], [34], [39]
CMS.1	Cobol	1980	General	No	[2], [108]
FMS.3	Fortran	1981	General	No	[228]
Mothra	Fortran	1987	General	Yes	[63], [64]
Proteum 1.4	C	1993	Interface Mutation, Finite State Machines	No	[52], [53]
TUMS	C	1995	Mutant Schemata Generation	No	[235]–[237]
Insure++	C/C++	1998	Source Code Instrumentation (Commercial)	Commercially	[198]
Proteum/IM 2.0	C	2001	Interface Mutation, Finite State Machines	Yes	[59]
Jester	Java	2001	General (Open Source)	Yes	[163]
Pester	Python	2001	General (Open Source)	Yes	[163]
TDS	CORBA IDL	2001	Interface Mutation	No	[100]
Nester	C#	2002	General (Open Source)	Yes	[220]
JavaMut	Java	2002	General	Yes	[45]
MuJava	Java	2004	Mutant Schemata, Reflection Technique	Yes	[151], [152], [185]
Plextest	C/C++	2005	General (Commercial)	Commercially	[118]
SQLMutation	SQL	2006	General	Yes	[233]
Certitude	C/C++	2006	General (Commercial)	Commercially	[42]
SESAME	C, Lustre, Pascal	2006	Assembler Injection	No	[49]
ExMAn	C, Java	2006	TXL	Yes	[30]
MUGAMMA	Java	2006	Remote Monitoring	Yes	[126]
MuClipse	Java	2007	Weak Mutation, Mutant Schemata, Eclipse plug-in	Yes	[218]
CSAW	C	2007	Variable type optimization	Yes	[82], [83]
Heckle	Ruby	2007	General (Open Source)	Yes	[206]
Jumble	Java	2007	General (Open Source)	Yes	[221]
Testooj	Java	2007	General	Yes	[200]
ESPT	C/C++	2008	Tabular	Yes	[89]
MUFORMAT	C	2008	Format String Bugs	No	[214]
CREAM	C#	2008	General	No	[73]
MUSIC	SQL(JSP)	2008	Weak Mutation, SQL Vulnerabilities	No	[212]
MILU	C	2008	Higher Order Mutation, Search-based technique, Test harness embedding	Yes	[123]
Javalanche	Java	2009	Invariant and Impact analysis	Yes	[106], [208]
GAmera	WS-BPEL	2009	Genetic algorithm	Yes	[77]
MutateMe	PHP	2009	General (Open Source)	Yes	[33]
AjMutator	AspectJ	2009	General	Yes	[51]
JDAMA	SQL(JDBC)	2009	Byte code translation	Yes	[264]

year 2004; this just appears to be an anomalous productive year for Mutation Testing. The reader will also notice that 1986 is unique as no publications were found. An interesting explanation was provided by Offutt [176]: "1986 was when we were maximally devoted to programming Mothra."

We performed a regression analysis on these data and found there is a strong positive correlation between year and the number of publications ($r = 0.7858$). In order to predict the trend of publications in the future, we have tried to find a trend line for these data using several common regression models: Linear, Logarithmic, Polynomial, Power, Exponential, and Moving average. The dashed line in Fig. 1 is the best fit line we found. It uses a quadratic model, which achieves the highest coefficient of determination ($R^2 = 0.7747$). To put the Mutation Testing growth trend into a wider context, we also collected and plotted the publication data from DBLP for the subject of computer science as a whole [232]. According to DBLP, the general growth in computer science is also exponential. From this analysis it is clear that Mutation Testing remains at least as healthy as computer science itself.

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

The goal of this separation of papers into theoretical and practical work is to allow us to analyze the temporal relationship between the development of theoretical and practical research effort by the community. Fig. 11 shows

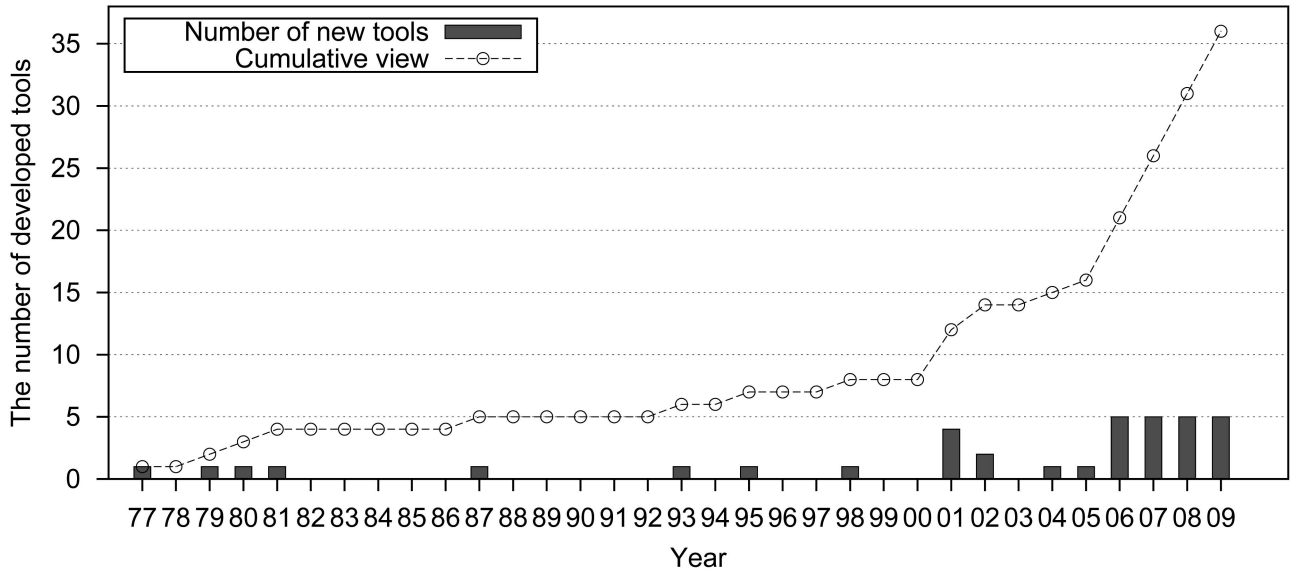

Fig. 10. The number of tools introduced for each year.

the overall cumulative result. It is clear that both theoretical and practical work is increasing. In 2006, for the first time, the total number of practical publications surpasses the number of theoretical publications. To take a closer look at this relationship, Fig. 12 shows the number of publications per year. From 1977 to 2000, there were fewer practical publications than theoretical. From 2000 to 2009, most of the research work appears to shift to the application area. This provides some evidence to suggest that the field is starting to move from foundational theory to practical application, possibly a sign of increasing maturity.

In the Redwine-Riddle maturation model [203], there is a trend that indicates that a technology takes about 15 to 20 years to reach a level of maturity at which time industrial uptake takes place. Suppose we cast our attention back by 15 years to the mid 1990s. We reach a point where only approximately 25 percent of the current volume of output had then been published in the literature. (see Fig. 12). The ideas found in this early Mutation Testing literature have now been implemented in practical commercial Mutation Testing tools, as shown in Table 7. This observation suggests that the development of Mutation Testing is in line with Redwine and Riddle's findings.

Furthermore, the set of Mutation Testing systems developed in the laboratory now provides tooling for a great many different programming language paradigms (as shown in Table 7). This provides further evidence of maturity and offers hope that, as these tools mature, following the Redwine and Riddle model, we can expect a future state-of-practice in which a wide coverage of popular programming paradigms will be covered by real-world Mutation Testing tools.

Finally, an increasing level of maturity can also be seen in the development of the empirical studies reported on Mutation Testing. For example, there is a noticeable trend for empirical studies to involve more programs and to also involve bigger and more realistic programs, as can be seen in the chronological data on empirical studies presented in Figs. 7 and 8. However, it should also be noted that more work is required on real-world programs and that many of our empirical evidence still rests on studies of what would now be regarded as "toy programs." There also appears to be an increasing degree of corroboration and replication of the results reported (see Table 6).

TABLE 8
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

9 Discussion of Unresolved Problems, Barriers, and Areas of Success

This section discusses some of the findings and conclusions that can be drawn from this survey of the literature concerning the current state of Mutation Testing. Naturally, this account is, to some extent, influenced by the authors' own position on Mutation Testing. However, we have attempted to take a step back and to summarize unresolved problems, barriers, and areas of success in an objective manner, based on the available literature and the trends we have found within it.

9.1 Unresolved Problems

One barrier to wider application of Mutation Testing centers 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. This remains an unresolved problem. We see several possible developments along this line. Past work has concentrated on techniques to detect equivalent mutants once they have been produced. In the future, Mutation Testing approaches may seek to avoid their initial creation or to reduce their likelihood. Mutation Testing may

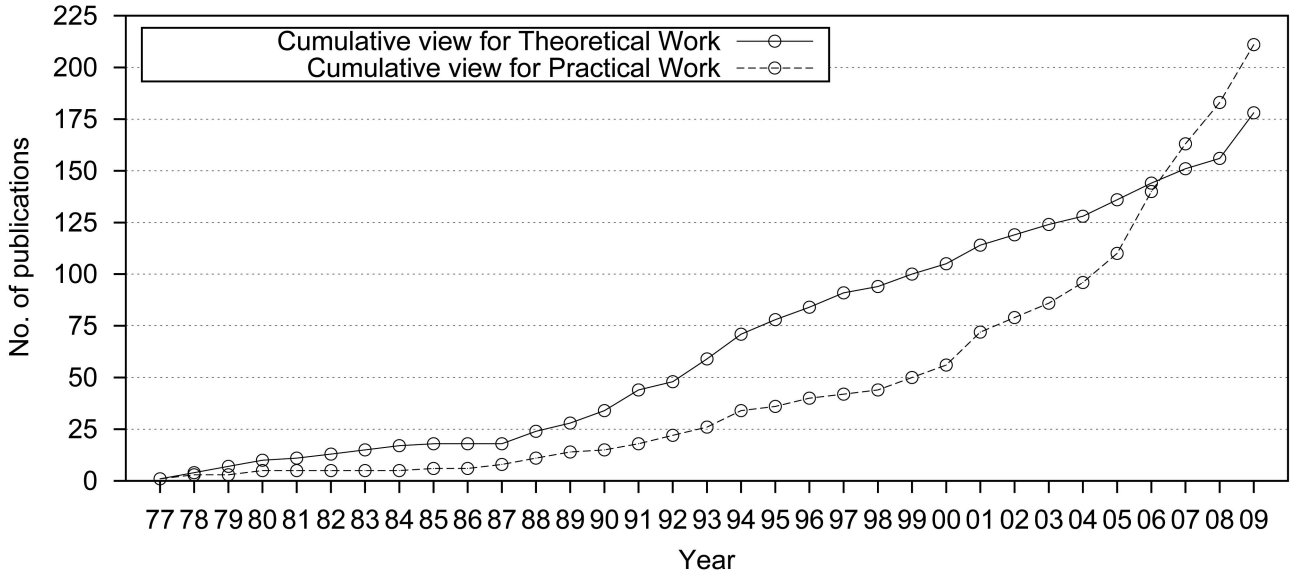

Fig. 11. Theoretical publications versus practical publications (cumulative view).

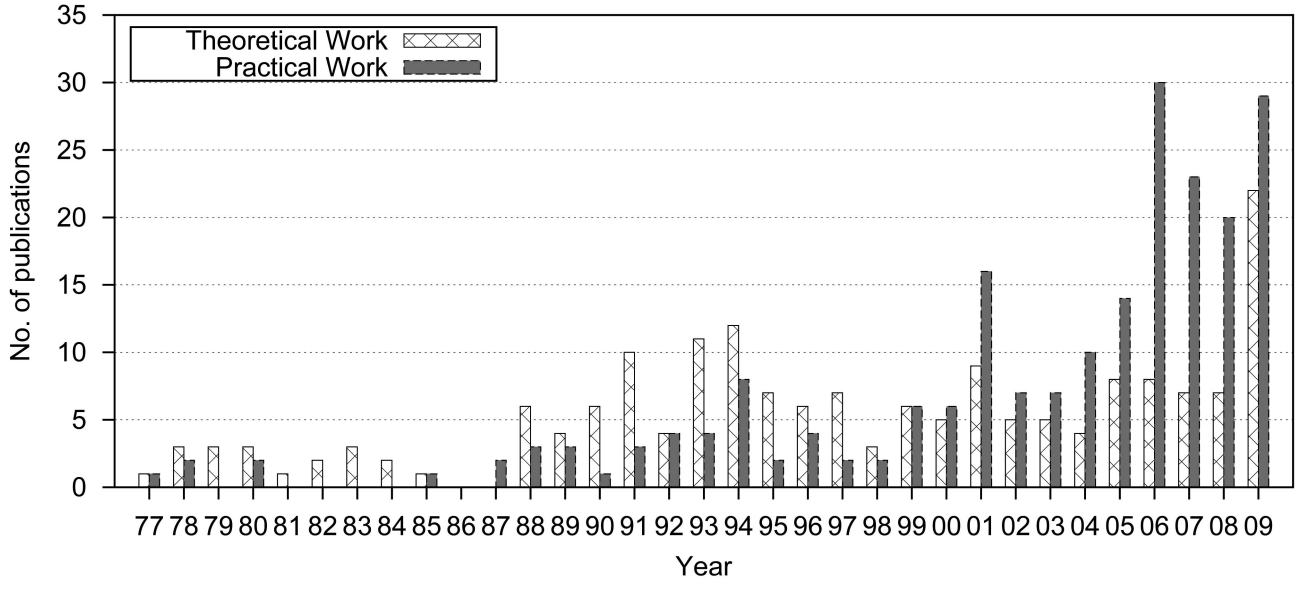

Fig. 12. Theoretical publications versus practical publications.

be applied to languages that do not have equivalent mutants. 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, behaviorally these two classes of mutants are highly similar.

Most work on Mutation Testing has been concerned with the generation of mutants. Comparatively, less work has concentrated on the generation of test cases to kill mutants. Though there are existing tools for mutant generation that are mature enough for commercial application, there is currently no tool that offers test cases generation to kill mutants at a similar level of maturity. The state of the art is therefore one in which Mutation Testing has provided a way to assess 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. However, at present, practical software test data generation for mutation test adequacy remains an unresolved problem.

9.2 Barriers to be Overcome

There remains a perception—perhaps misplaced, but nonetheless widely held—that Mutation Testing is costly and impractical. This remains a barrier to wider academic interest in the subject and also to a wider uptake within industry. We hope that this survey will go some way toward addressing the remaining doubts of academics. There is plenty of

TABLE 9a
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 9b

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 9c

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 9d

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

evidence in this survey to show that Mutation Testing is on the cusp of a rising trend of maturity and that it is making a transition from academic to industrial application.

The barriers to industrial uptake are more significant and will take longer to fully overcome. The primary barriers appear to be those that apply to many other emergent software technologies as they make their transition from laboratory to wider practical application. That is, a need for reliable tooling and compelling evidence to motivate the necessary investment of time and money in such tooling.

As the survey shows, there is an increasingly practical trend in empirical work. That is, as shown in Section 6, empirical studies are increasingly focussing on nontrivial industrial subjects, rather than laboratory programs. In order to provide a compelling body of evidence, sufficient to overcome remaining practitioner doubts, this trend will need to continue. There is also evidence that Mutation Testing tools are starting to emerge as practical commercial products (see Section 7). However, more tooling is required to ensure widespread industrial uptake. Furthermore, there is a pressing need to address the, currently unresolved, problem of test case generation. An automated practical tool that offered test case generation would be a compelling facilitator for industrial uptake of Mutation Testing. No such tool currently exists for test data generation, but recent developments in dynamic symbolic execution [104], [209], [230] and search-based test data generation [10], [135], [162] indicates that such a tool cannot be far off. The Mutation Testing community will need to ensure that it does not lag behind in this trend.

9.3 Areas of Success

As this paper has shown (see Figs. 1, 3, 11, and 12 and Tables 7 and 8), work on Mutation Testing is growing at a rapid rate and tools and techniques are reaching a level of maturity not previously witnessed in this field. There has also been a great deal of work to extend Mutation Testing to new languages, paradigms and to find new domains of application (see Figs. 5, 7, 8, and 9 and Tables 5 and 9). Based on this existing success, we can expect that the future will bring many more applications. There may 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 costs, without overcompromising on quality. This has led to successful techniques for reducing mutation effort without significant reduction in test effectiveness (as described in Section 3).

10 Conclusion and Future Work

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.

Recent work has tended to focus on more elaborate forms of mutation than on the relatively simple faults that have been previously considered. There is interest in the semantic effects of mutation, rather than the syntactic achievement of a mutation. This migration from the syntactic achievement of mutation to the desired semantic effect has raised interest in higher order mutation to generate subtle faults and to find those mutations that denote real faults. We hope the future will see a further coming of age, with the generation of more realistic mutants and the test cases to kill them and with the provision of practical tooling to support both.

Acknowledgments

The authors benefitted from many discussions with researchers and practitioners in the Mutation Testing community, approximately 50 of whom kindly provided very helpful comments and feedback on an earlier draft of this analytical survey. The authors are very grateful to these colleagues for their time and expertise though they are not able to name them all individually. This work is partly 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. The authors are also grateful to Lorna Anderson and Kathy Harman for additional proof reading.

References

[1] R. Abraham and M. Erwig, "Mutation Operators for Spreadsheets," *IEEE Trans. Software Eng.*, vol. 35, no. 1, pp. 94-108, Jan./Feb. 2009.

[2] A.T. Acree, "On Mutation," PhD thesis, Georgia Inst. of Technology, 1980.
[3] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Mutation Analysis," Technical Report GIT-ICS-79/08, Georgia Inst. of Technology, 1979.
[4] K. Adamopoulos, "Search Based Test Selection and Tailored Mutation," master's thesis, King's College London, 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," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1338-1349, June 2004.
[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," Technical Report SERC-TR-41-P, Purdue Univ., Mar. 1989.
[7] B.K. Aichernig, "Mutation Testing in the Refinement Calculus," *Formal Aspects of Computing,* vol. 15, nos. 2-3, pp. 280-295, Nov. 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," *Proc. Ninth Int'l Conf. Fundamental Approaches to Software Eng.,* pp. 324-338, Mar. 2006.
[9] R.T. Alexander, J.M. Bieman, S. Ghosh, and B. Ji, "Mutation of Java Objects," *Proc. 13th Int'l Symp. Software Reliability Eng.,* pp. 341-351, Nov. 2002.
[10] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test-Case Generation," *IEEE Trans. Software Eng.,* vol. 36, no. 6, pp. 742-762, Nov./Dec. 2010.
[11] P. Ammann and J. Offutt, *Introduction to Software Testing.* Cambridge Univ. Press, 2008.
[12] P. Anbalagan and T. Xie, "Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs," *Proc. Second Workshop Mutation Analysis,* pp. 51-56, Nov. 2006.
[13] P. Anbalagan and T. Xie, "Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs," *Proc. 19th Int'l Symp. Software Reliability Eng.,* pp. 239-248, Nov. 2008.
[14] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" *Proc. 27th Int'l Conf. Software Eng.,* pp. 402-411, May 2005.
[15] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Trans. Software Eng.,* vol. 32, no. 8, pp. 608-624, Aug. 2006.
[16] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic Mutation Test Input Data Generation via Ant Colony," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1074-1081, July 2007.
[17] J.S. Baekken and R.T. Alexander, "A Candidate Fault Model for AspectJ Pointcuts," *Proc. 17th Int'l Symp. Software Reliability Eng.,* pp. 169-178, Nov. 2006.
[18] D. Baldwin and F.G. Sayward, "Heuristics for Determining Equivalence of Program Mutations," Research Report 276, Yale Univ., 1979.
[19] 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.
[20] S.S. Batth, E.R. Vieira, A.R. Cavalli, and M.U. Uyar, "Specification of Timed EFSM Fault Models in SDL," *Proc. 27th IFIP WG 6.1 Int'l Conf. Formal Techniques for Networked and Distributed Systems,* pp. 50-65, June 2007.
[21] B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. Le Traon, "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment," *Proc. 13th Int'l Symp. Software Reliability Eng.,* pp. 195-206, Nov. 2002.
[22] B. Baudry, V. Le Hanh, J.-M. Jézéquel, and Y. Le Traon, "Trustable Components: Yet Another Mutation-Based Approach," *Proc. First Workshop Mutation Analysis,* pp. 47-54, Oct. 2000.
[23] F. Belli, C.J. Budnik, and W.E. Wong, "Basic Operations for Generating Behavioral Mutants," *Proc. Second Workshop Mutation Analysis,* p. 9, 2006.
[24] J. Bieman, S. Ghosh, and R.T. Alexander, "A Technique for Mutation of Java Objects," *Proc. 16th IEEE Int'l Conf. Automated Software Eng.,* p. 337, Nov. 2001.
[25] P.E. Black, V. Okun, and Y. Yesha, "Mutation of Model Checker Specifications for Test Generation and Evaluation," *Proc. First Workshop Mutation Analysis,* pp.14-20, Oct. 2000.
[26] B. Bogacki and B. Walter, "Evaluation of Test Code Quality with Aspect-Oriented Mutations," *Proc. Seventh Int'l Conf. eXtreme Programming and Agile Processes in Software Eng.,* pp. 202-204, June 2006.
[27] B. Bogacki and B. Walter, "Aspect-Oriented Response Injection: An Alternative to Classical Mutation Testing," *Software Eng. Techniques: Design for Quality,* pp. 273-282, Springer, 2007.
[28] N. Bombieri, F. Fummi, and G. Pravadelli, "A Mutation Model for the SystemC TLM2.0 Communication Interfaces," *Proc. Conf. Design, Automation and Test in Europe,* pp. 396-401, Mar. 2008.
[29] J.H. Bowser, "Reference Manual for Ada Mutant Operators," Technical Report GIT-SERC-88/02, Georgia Inst. of Technology, 1988.
[30] J.S. Bradbury, J.R. Cordy, and J. Dingel, "ExMAn: A Generic and Customizable Framework for Experimental Mutation Analysis," *Proc. Second Workshop Mutation Analysis,* pp. 57-62, Nov. 2006.
[31] J.S. Bradbury, J.R. Cordy, and J. Dingel, "Mutation Operators for Concurrent Java (J2SE 5.0)," *Proc. Second Workshop Mutation Analysis,* pp. 83-92, Nov. 2006.
[32] J.S. Bradbury, J.R. Cordy, and J. Dingel, "Comparative Assessment of Testing and Model Checking Using Program Mutation," *Proc. Third Workshop Mutation Analysis, published with Proc. Second Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 210-222, 2007.
[33] P. Brady, "MutateMe," http://github.com/padraic/mutateme/tree/master, 2007.
[34] T.A. Budd, "Mutation Analysis of Program Test Data," PhD thesis, Yale Univ., 1980.
[35] T.A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica,* vol. 18, no. 1, pp. 31-45, Mar. 1982.
[36] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "The Design of a Prototype Mutation System for Program Testing," *Proc. Am. Fed. of Information Processing Soc. Nat'l Computer Conf.,* vol. 74, pp. 623-627, June 1978.
[37] 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," *Proc. Seventh ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 220-233, Jan. 1980.
[38] T.A. Budd and A.S. Gopal, "Program Testing by Specification Mutation," *Computer Languages,* vol. 10, no. 1, pp. 63-73, 1985.
[39] T.A. Budd, R. Hess, and F.G. Sayward, "EXPER Implementor's Guide," technical report, Yale Univ., 1980.
[40] T.A. Budd and F.G. Sayward, "Users Guide to the Pilot Mutation System," Technical Report 114, Yale Univ., 1977.
[41] R.H. Carver, "Mutation-Based Testing of Concurrent Programs," *Proc. IEEE Int'l Test Conf. Designing, Testing, and Diagnostics,* pp. 845-853, Oct. 1993.
[42] Cetress, "Certitude," http://www.certess.com/product/, 2006.
[43] W.K. Chan, S.C. Cheung, and T.H. Tse, "Fault-Based Testing of Database Application Programs with Conceptual Data Model," *Proc. Fifth Int'l Conf. Quality Software,* pp. 187-196, Sept. 2005.
[44] P. Chevalley, "Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach," *Proc. Eighth Asia-Pacific Software Eng. Conf.,* p. 267, Dec. 2001.
[45] P. Chevalley and P. Thévenod-Fosse, "A Mutation Analysis Tool for Java Programs," *Int'l J. Software Tools for Technology Transfer,* vol. 5, no. 1, pp. 90-103, Nov. 2002.
[46] B. Choi, "Software Testing Using High Performance Computers," PhD thesis, Purdue Univ., July 1991.
[47] B. Choi and A.P. Mathur, "High-Performance Mutation Testing," *J. Systems and Software,* vol. 20, no. 2, pp. 135-152, Feb. 1993.
[48] W.M. Craft, "Detecting Equivalent Mutants Using Compiler Optimization Techniques," master's thesis, Clemson Univ., Sept. 1989.
[49] Y. Crouzet, H. Waeselynck, B. Lussier, and D. Powell, "The SESAME Experience: From Assembly Languages to Declarative Models," *Proc. Second Workshop Mutation Analysis,* p. 7, Nov. 2006.
[50] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *ACM SIGSOFT Software Eng. Notes,* vol. 21, no. 3, pp. 158-177, May 1996.
[51] R. Delamare, B. Baudry, and Y. Le Traon, "AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors," *Proc. Fourth Int'l Workshop Mutation Analysis, published with Proc. Second Int'l Conf. Software Testing, Verification, and Validation Workshops,* pp. 200-204, Apr. 2009.

[52] M.E. Delamaro, "Proteum—A Mutation Analysis Based Testing Environmen," master's thesis, Univ. of São Paulo, 1993.

[53] M.E. Delamaro and J.C. Maldonado, "Proteum—A Tool for the Assessment of Test Adequacy for C Programs," *Proc. Conf. Performability in Computing Systems,* pp. 79-95, July 1996.

[54] M.E. Delamaro and J.C. Maldonado, "Interface Mutation: Assessing Testing Quality at Interprocedural Level," *Proc. 19th Int'l Conf. Chilean Computer Science Soc.,* pp. 78-86, Nov. 1999.

[55] M.E. Delamaro, J.C. Maldonado, and A.P. Mathur, "Integration Testing Using Interface Mutation," *Proc. Seventh Int'l Symp. Software Reliability Eng.,* pp. 112-121, Oct.-Nov. 1996.

[56] M.E. Delamaro, J.C. Maldonado, and A.P. Mathur, "Interface Mutation: An Approach for Integration Testing," *IEEE Trans. Software Eng.,* vol. 27, no. 3, pp. 228-247, Mar. 2001.

[57] M.E. Delamaro, J.C. Maldonado, A. Pasquini, and A.P. Mathur, "Interface Mutation Test Adequacy Criterion: An Empirical Evaluation," technical report, State Univ. of Maringá, 2000.

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

[59] M.E. Delamaro, J.C. Maldonado, and A. Vincenzi, "Proteum/IM 2.0: An Integrated Mutation Testing Environment," *Proc. First Workshop Mutation Analysis,* Oct. 2000.

[60] R.A. DeMillo, "Program Mutation: An Approach to Software Testing," technical report, Georgia Inst. of Technology, 1983.

[61] 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," Technical Report SERC-TR-92-P, Purdue Univ., 1992.

[62] R.A. DeMillo, "Test Adequacy and Program Mutation," *Proc. 11th Int'l Conf. Software Eng.,* pp. 355-356, May 1989.

[63] R.A. DeMillo, D.S. Guindi, K.N. King, and W.M. McCracken, "An Overview of the Mothra Software Testing Environment," Technical Report SERC-TR-3-P, Purdue Univ., 1987.

[64] 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," *Proc. Second Workshop Software Testing, Verification, and Analysis,* pp. 142-151, July 1988.

[65] R.A. DeMillo, E.W. Krauser, and A.P. Mathur, "Compiler-Integrated Program Mutation," *Proc. Fifth Ann. Computer Software and Applications Conf.,* pp. 351-356, Sept. 1991.

[66] 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, Apr. 1978.

[67] R.A. DeMillo and A.J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Trans. Software Eng.,* vol. 17, no. 9, pp. 900-910, Sept. 1991.

[68] R.A. DeMillo and A.J. Offutt, "Experimental Results from an Automatic Test Case Generator," *ACM Trans. Software Eng. and Methodology,* vol. 2, no. 2, pp. 109-127, Apr. 1993.

[69] A. Derezińska, "Object-Oriented Mutation to Assess the Quality of Tests," *Proc. 29th Euromicro Conf.,* pp. 417-420, Sept. 2003.

[70] A. Derezińska, "Advanced Mutation Operators Applicable in C# Programs," technical report, Warsaw Univ. of Technology, 2005.

[71] A. Derezińska, "Quality Assessment of Mutation Operators Dedicated for C# Programs," *Proc. Sixth Int'l Conf. Quality Software,* Oct. 2006.

[72] A. Derezińska and A. Szustek, "CREAM—A System for Object-Oriented Mutation of C# Programs," technical report, Warsaw Univ. of Technology, 2007.

[73] A. Derezińska and A. Szustek, "Tool-Supported Advanced Mutation Approach for Verification of C# Programs," *Proc. Third Int'l Conf. Dependability of Computer Systems,* pp. 261-268, June 2008.

[74] W. Ding, "Using Mutation to Generate Tests from Specifications," master's thesis, George Mason Univ., 2000.

[75] H. Do and G. Rothermel, "A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults," *Proc. 21st IEEE Int'l Conf. Software Maintenance,* pp. 411-420, Sept. 2005.

[76] H. Do and G. Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," *IEEE Trans. Software Eng.,* vol. 32, no. 9, pp. 733-752, Sept. 2006.

[77] J.J. Domínguez-Jiménez, A. Estero-Botaro, and I. Medina-Bulo, "A Framework for Mutant Genetic Generation for WS-BPEL," *Proc. 35th Conf. Current Trends in Theory and Practice of Computer Science,* pp. 229-240, Jan. 2009.

[78] W. Du and A.P. Mathur, "Vulnerability Testing of Software System Using Fault Injection," Technical Report COAST TR 98-02, Purdue Univ., 1998.

[79] W. Du and A.P. Mathur, "Testing for Software Vulnerability Using Environment Perturbation," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 603-612, June 2000.

[80] L.du Bousquet and M. Delaunay, "Mutation Analysis for Lustre Programs: Fault Model Description and Validation," *Proc. Third Workshop Mutation Analysis, published with Proc. Second Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 176-184, Sept. 2007.

[81] L. du Bousquet and M. Delaunay, "Using Mutation Analysis to Evaluate Test Generation Strategies in a Synchronous Context," *Proc. Second Int'l Conf. Software Eng. Advances,* p. 40, Aug. 2007.

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

[83] M. Ellims, D.C. Ince, and M. Petre, "The Csaw C Mutation Tool: Initial Results," *Proc. Third Workshop Mutation Analysis, published with Proc. Second Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 185-192, Sept. 2007.

[84] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Mutation Operators for WS-BPEL 2.0," *Proc. 21st Int'l Conf. Software and Systems Eng. and Their Applications,* Dec. 2008.

[85] 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," *Proc. 19th Int'l Conf. Chilean Computer Science Soc.,* pp. 96-104, Nov. 1999.

[86] 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," *Proc. IFIP TC6 Eighth Int'l Conf. Formal Description Techniques VIII,* vol. 43, pp. 329-337, 1995.

[87] S.C.P.F. Fabbri, J.C. Maldonado, T. Sugeta, and P.C. Masiero, "Mutation Testing Applied to Validate Specifications Based on Statecharts," *Proc. 10th Int'l Symp. Software Reliability Eng.,* pp. 210-219, Nov. 1999.

[88] S.P.F. Fabbri, M.E. Delamaro, J.C. Maldonado, and P. Masiero, "Mutation Analysis Testing for Finite State Machines," *Proc. Fifth Int'l Symp. Software Reliability Eng.,* pp. 220-229, Nov. 1994.

[89] X. Feng, S. Marr, and T. O'Callaghan, "ESTP: An Experimental Software Testing Platform," *Proc. Third Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 59-63, Aug. 2008.

[90] F.C. Ferrari, J.C. Maldonado, and A. Rashid, "Mutation Testing for Aspect-Oriented Programs," *Proc. First Int'l Conf. Software Testing, Verification, and Validation,* pp. 52-61, Apr. 2008.

[91] S. Fichter, "Parallelizing Mutation on a Hypercube," master's thesis, Clemson Univ., 1991.

[92] V.N. Fleyshgakker and S.N. Weiss, "Efficient Mutation Analysis: A New Approach," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 185-195, Aug. 1994.

[93] P.G. Frankl, S.N. Weiss, and C. Hu, "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness," technical report, Polytechnic Univ., 1994.

[94] P.G. Frankl, S.N. Weiss, and C. Hu, "All-Uses vs Mutation Testing: An Experimental Comparison of Effectiveness," *J. Systems and Software,* vol. 38, no. 3, pp. 235-253, Sept. 1997.

[95] G. Fraser and F. Wotawa, "Mutant Minimization for Model-Checker Based Test-Case Generation," *Proc. Third Workshop Mutation Analysis, published with Proc. Second Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 161-168, Sept. 2007.

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

[97] A.K. Ghosh, T. O'Connor, and G. McGraw, "An Automated Approach for Identifying Potential Vulnerabilities in Software," *Proc. IEEE Symp. Security and Privacy,* pp. 104-114, May 1998.

[98] S. Ghosh, "Testing Component-Based Distributed Applications," PhD thesis, Purdue Univ., 2000.

[99] S. Ghosh, "Towards Measurement of Testability of Concurrent Object-Oriented Programs Using Fault Insertion: A Preliminary Investigation," *Proc. Second IEEE Int'l Workshop Source Code Analysis and Manipulation,* pp. 17-25, 2002.

[100] S. Ghosh, P. Govindarajan, and A.P. Mathur, "TDS: A Tool for Testing Distributed Component-Based Applications," *Proc. First Workshop Mutation Analysis,* pp. 103-112, Oct. 2000.

[101] S. Ghosh and A.P. Mathur, "Interface Mutation to Assess the Adequacy of Tests for Components and Systems," *Proc. 34th Int'l Conf. Technology of Object-Oriented Languages and Systems,* pp. 37-46, July-Aug. 2000.

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

[103] M.R. Girgis and M.R. Woodward, "An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis," *Proc. Eighth Int'l Conf. Software Eng.,* pp. 313-319, Aug. 1985.

[104] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 213-223, June 2005.

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

[106] B.J.M. Grün, D. Schuler, and A. Zeller, "The Impact of Equivalent Mutants," *Proc. Fourth Int'l Workshop Mutation Analysis, published with Proc. Second Int'l Conf. Software Testing, Verification, and Validation Workshops,* pp. 192-199, Apr. 2009.

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

[108] J.M. Hanks, "Testing Cobol Programs by Mutation," PhD thesis, Georgia Inst. of Technology, 1980.

[109] M. Harman, R. Hierons, and S. Danicic, "The Relationship between Program Dependence and Mutation Analysis," *Proc. First Workshop Mutation Analysis,* pp. 5-13, Oct. 2000.

[110] 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, Dec. 1999.

[111] R.M. Hierons and M.G. Merayo, "Mutation Testing from Probabilistic Finite State Machines," *Proc. Third Workshop Mutation Analysis, published with Proc. Second Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 141-150, Sept. 2007.

[112] R.M. Hierons and M.G. Merayo, "Mutation Testing from Probabilistic and Stochastic Finite State Machines," *J. Systems and Software,* vol. 82, no. 11, pp. 1804-1818, Nov. 2009.

[113] J.R. Horgan and A.P. Mathur, "Weak Mutation is Probably Strong Mutation," Technical Report SERC-TR-83-P, Purdue Univ., 1990.

[114] S.-S. Hou, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, "Applying Interface-Contract Mutation in Regression Testing of Component-Based Software," *Proc. 23rd Int'l Conf. Software Maintenance,* pp. 174-183, Oct. 2007.

[115] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Trans. Software Eng.,* vol. 8, no. 4, pp. 371-379, July 1982.

[116] S. Hussain, "Mutation Clustering," master's thesis, King's College London, 2008.

[117] J. Hwang, T. Xie, F. Chen, and A.X. Liu, "Systematic Structural Testing of Firewall Policies," *Proc. IEEE Symp. Reliable Distributed Systems,* pp. 105-114, Oct. 2008.

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

[119] D. Jackson and M.R. Woodward, "Parallel Firm Mutation of Java Programs," *Proc. First Workshop Mutation Analysis,* pp. 55-61, Oct. 2000.

[120] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A Novel Method of Mutation Clustering Based on Domain Analysis," *Proc. 21st Int'l Conf. Software Eng. and Knowledge Eng.,* July 2009.

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

[122] Y. Jia and M. Harman, "Constructing Subtle Faults Using Higher Order Mutation Testing," *Proc. Eighth Int'l Working Conf. Source Code Analysis and Manipulation,* pp. 249-258, Sept. 2008.

[123] Y. Jia and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," *Proc. Third Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 94-98, Aug. 2008.

[124] C. Jing, Z. Wang, X. Shi, X. Yin, and J. Wu, "Mutation Testing of Protocol Messages Based on Extended TTCN-3," *Proc. 22nd Int'l Conf. Advanced Information Networking and Applications,* pp. 667-674, Mar. 2008.

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

[126] S.-W. Kim, M.J. Harrold, and Y.-R. Kwon, "MUGAMMA: Mutation Analysis of Deployed Software to Increase Confidence Assist Evolution," *Proc. Second Workshop Mutation Analysis,* p. 10, Nov. 2006.

[127] S. Kim, J.A. Clark, and J.A. McDermid, "Assessing Test Set Adequacy for Object Oriented Programs Using Class Mutation," *Proc. Third Symp. Software Technology,* Sept. 1999.

[128] S. Kim, J.A. Clark, and J.A. McDermid, "The Rigorous Generation of Java Mutation Operators Using HAZOP," *Proc. 12th Int'l Conf. Software and Systems Eng. and Their Applications,* Nov.-Dec. 1999.

[129] S. Kim, J.A. Clark, and J.A. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs," *Proc. Net.ObjectDays Conf. Object-Oriented Software Systems,* 2000.

[130] S. Kim, J.A. Clark, and J.A. McDermid, "Investigating the Effectiveness of Object-Oriented Testing Strategies Using the Mutation Method," *Proc. First Workshop Mutation Analysis,* pp. 207-225, Oct. 2000.

[131] 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, Oct. 1991.

[132] E.W. Krauser, "Compiler-Integrated Software Testing," PhD thesis, Purdue Univ., 1991.

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

[134] E.W. Krauser, A.P. Mathur, and V.J. Rego, "High Performance Software Testing on SIMD Machines," *Proc. Second Workshop Software Testing, Verification, and Analysis,* pp. 171-177, July 1988.

[135] K. Lakhotia, P. McMinn, and M. Harman, "Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet?" *Proc. Testing: Academia & Industry Conf.—Practice And Research Techniques,* pp. 95-104, Sept. 2009.

[136] W.B. Langdon, M. Harman, and Y. Jia, "Multi Objective Higher Order Mutation Testing with Genetic Programming," *Proc. Fourth Testing: Academic and Industrial Conf.—Practice and Research,* Sept. 2009.

[137] W.B. Langdon, M. Harman, and Y. Jia, "Multi Objective Mutation Testing with Genetic Programming," *Proc. Genetic and Evolutionary Computation Conf.,* July 2009.

[138] Y. Le Traon, B. Baudry, and J.-M. Jézéquel, "Design by Contract to Improve Software Vigilance," *IEEE Trans. Software Eng.,* vol. 32, no. 8, pp. 571-586, Aug. 2006.

[139] Y.L. Traon, T. Mouelhi, and B. Baudry, "Testing Security Policies: Going Beyond Functional Testing," *Proc. 18th IEEE Int'l Symp. Software Reliability,* pp. 93-102, Nov. 2007.

[140] S. Lee, X. Bai, and Y. Chen, "Automatic Mutation Testing and Simulation on OWL-S Specified Web Services," *Proc. 41st Ann. Simulation Symp.,* pp. 149-156, Apr. 2008.

[141] S.D. Lee, "Weak versus Strong: An Empirical Comparison of Mutation Variants," master's thesis, Clemson Univ., 1991.

[142] S.C. Lee and A.J. Offutt, "Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis," *Proc. 12th Int'l Symp. Software Reliability Eng.,* pp. 200-209, Nov. 2001.

[143] J.B. Li and J. Miller, "Testing the Semantics of W3C XML Schema," *Proc. 29th Ann. Int'l Computer Software and Applications Conf.,* pp. 443-448, July 2005.

[144] R. Lipton, "Fault Diagnosis of Computer Programs," student report, Carnegie Mellon Univ., 1971.

[145] R.J. Lipton and F.G. Sayward, "The Status of Research on Program Mutation," *Proc. Workshop Software Testing and Test Documentation,* pp. 355-373, Dec. 1978.

[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," *Proc. 22nd IEEE Int'l Conf. Software Maintenance,* pp. 113-122, Sept. 2006.

[147] B. Long, R. Duke, D. Goldson, P. Strooper, and L. Wildman, "Mutation-Based Exploration of a Method for Verifying Concurrent Java Components," *Proc. 18th Int'l Parallel and Distributed Processing Symp.,* p. 265, Apr. 2004.

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

[149] Y.-S. Ma, M.J. Harrold, and Y.-R. Kwon, "Evaluation of Mutation Testing for Object-Oriented Programs," *Proc. 28th Int'l Conf. Software Eng.,* pp. 869-872, May 2006.

[150] Y.-S. Ma, Y.-R. Kwon, and A.J. Offutt, "Inter-Class Mutation Operators for Java," *Proc. 13th Int'l Symp. Software Reliability Eng.,* pp. 352-363, Nov. 2002.

[151] Y.-S. Ma, A.J. Offutt, and Y.-R. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification, and 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," *Proc. 28th Int'l Conf. Software Eng.*, pp. 827-830, May 2006.

[153] B. Marick, "The Weak Mutation Hypothesis," *Proc. Fourth Symp. Software Testing, Analysis, and Verification* pp. 190-199, Oct. 1991.

[154] E.E. Martin and T. Xie, "A Fault Model and Mutation Testing of Access Control Policies," *Proc. 16th Int'l Conf. World Wide Web*, pp. 667-676, May 2007.

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

[156] A.P. Mathur, "Performance, Effectiveness, and Reliability Issues in Software Testing," *Proc. Fifth Int'l Computer Software and Applications Conf.*, pp. 604-605, Sept. 1991.

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

[158] A.P. Mathur and E.W. Krauser, "Mutant Unification for Improved Vectorization," Technical Report SERC-TR-14-P, Purdue Univ., 1988.

[159] A.P. Mathur and W.E. Wong, "An Empirical Comparison of Mutation and Data Flow Based Test Adequacy Criteria," technical report, Purdue Univ., 1993.

[160] 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.

[161] P.S. May, "Test Data Generation: Two Evolutionary Approaches to Mutation Testing," PhD thesis, Univ. of Kent, 2007.

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

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

[164] L.J. Morell, "A Theory of Error-Based Testing," PhD thesis, Univ. of Maryland at College Park, 1984.

[165] T. Mouelhi, F. Fleurey, and B. Baudry, "A Generic Metamodel For Security Policies Mutation," *Proc. IEEE Int'l Conf. Software Testing, Verification, and Validation Workshop*, pp. 278-286, Apr. 2008.

[166] T. Mouelhi, Y. Le Traon, and B. Baudry, "Mutation Analysis for Security Tests Qualification," *Proc. Third Workshop Mutation Analysis, published with Proc. Second Testing: Academic and Industrial Conf. Practice and Research Techniques*, pp. 233-242, Sept. 2007.

[167] 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, Dec. 1999.

[168] A.S. Namin and J.H. Andrews, "Finding Sufficient Mutation Operators via Variable Reduction," *Proc. Second Workshop Mutation Analysis*, p. 5, Nov. 2006.

[169] A.S. Namin and J.H. Andrews, "On Sufficiency of Mutants," *Proc. 29th Int'l Conf. Software Eng.*, pp. 73-74, May 2007.

[170] A.S. Namin, J.H. Andrews, and D.J. Murdoch, "Sufficient Mutation Operators for Measuring Test Effectiveness," *Proc. 30th Int'l Conf. Software Eng.*, pp. 351-360, May 2008.

[171] R. Nilsson, A.J. Offutt, and S.F. Andler, "Mutation-Based Testing Criteria for Timeliness," *Proc. 28th Ann. Int'l Computer Software and Applications Conf.*, pp. 306-311, Sept. 2004.

[172] R. Nilsson, A.J. Offutt, and J. Mellin, "Test Case Generation for Mutation-Based Testing of Timeliness," *Proc. Second Workshop Model Based Testing*, pp. 97-114, Mar. 2006.

[173] A.J. Offutt, J. Pan, and J.M. Voas, "Procedures for Reducing the Size of Coverage-Based Test Sets," *Proc. 12th Int'l Conf. Testing Computer Software*, pp. 111-123, June 1995.

[174] A.J. Offutt, "The Coupling Effect: Fact or Fiction," *ACM SIGSOFT Software Eng. Notes*, vol. 14, no. 8, pp. 131-140, Dec. 1989.

[175] A.J. Offutt, "Investigations of the Software Testing Coupling Effect," *ACM Trans. Software Eng. and Methodology*, vol. 1, no. 1, pp. 5-20, Jan. 1992.

[176] A.J. Offutt, private communication, July 2008.

[177] A.J. Offutt, P. Ammann, and L.L. Liu, "Mutation Testing Implements Grammar-Based Testing ," *Proc. Second Workshop Mutation Analysis*, p. 12, Nov. 2006.

[178] 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, Sept. 1994.

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

[180] 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, Feb. 1999.

[181] 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.

[182] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 2, pp. 99-118, Apr. 1996.

[183] A.J. Offutt and S. Lee, "An Empirical Evaluation of Weak Mutation," *IEEE Trans. Software Eng.*, vol. 20, no. 5, pp. 337-344, May 1994.

[184] A.J. Offutt and S.D. Lee, "How Strong is Weak Mutation?" *Proc. Fourth Symp. Software Testing, Analysis, and Verification*, pp. 200-213, Oct. 1991.

[185] A.J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "An Experimental Mutation System for Java," *ACM SIGSOFT Software Eng. Notes*, vol. 29, no. 5, pp. 1-4, Sept. 2004.

[186] A.J. Offutt and J. Pan, "Detecting Equivalent Mutants and the Feasible Path Problem," *Proc. Ann. Conf. Computer Assurance*, pp. 224-236, June 1996.

[187] 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, Sept. 1997.

[188] 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, Feb. 1996.

[189] A.J. Offutt, R.P. Pargas, S.V. Fichter, and P.K. Khambekar, "Mutation Testing of Software Using a MIMD Computer," *Proc. Int'l Conf. Parallel Processing*, pp. 255-266, Aug. 1992.

[190] A.J. Offutt, G. Rothermel, and C. Zapf, "An Experimental Evaluation of Selective Mutation," *Proc. 15th Int'l Conf. Software Eng.*, pp. 100-107, May 1993.

[191] A.J. Offutt and R.H. Untch, "Mutation 2000: Uniting the Orthogonal," *Proc. First Workshop Mutation Analysis*, pp. 34-44, Oct. 2000.

[192] A.J. Offutt, J. Voas, and J. Payn, "Mutation Operators for Ada," Technical Report ISSE-TR-96-09, George Mason Univ., 1996.

[193] A.J. Offutt and W. Xu, "Generating Test Cases for Web Services Using Data Perturbation," *Proc. Workshop Testing, Analysis and Verification of Web Services*, pp. 1-10, July 2004.

[194] A.J. Offutt, "Automatic Test Data Generation," PhD thesis, Georgia Inst. of Technology, 1988.

[195] V. Okun, "Specification Mutation for Test Generation and Analysis," PhD thesis, Univ. of Maryland, 2004.

[196] T. Olsson and P. Runeson, "System Level Mutation Analysis Applied to a State-Based Language," *Proc. Eighth Ann. IEEE Int'l Conf. and Workshop Eng. of Computer Based Systems*, pp. 222-228, Apr. 2001.

[197] J. Pan, "Using Constraints to Detect Equivalent Mutants," master's thesis, George Mason Univ., 1994.

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

[199] 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.

[200] 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, Mar. 2007.

[201] A. Pretschner, T. Mouelhi, and Y.L. Traon, "Model-Based Tests for Access Control Policies," *Proc. First Int'l Conf. Software Testing, Verification, and Validation*, pp. 338-347, Apr. 2008.

[202] R. Probert and F. Guo, "Mutation Testing of Protocols: Principles and Preliminary Experimental Results," *Proc. Workshop Protocol Test Systems*, pp. 57-76, Oct. 1991.

[203] S.T. Redwine and W.E. Riddle, "Software Technology Maturation," *Proc. Eighth Int'l Conf. Software Eng.*, pp. 189-200, 1985.

[204] C.K. Roy and J.R. Cordy, "Towards a Mutation-Based Automatic Framework for Evaluating Code Clone Detection Tools ," *Proc. Canadian Conf. Computer Science and Software Eng.*, pp. 137-140, May 2008.

[205] C.K. Roy and J.R. Cordy, "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools," *Proc. Fourth Int'l Workshop Mutation Analysis, published with Proc. Second Int'l Conf. Software Testing, Verification, and Validation Workshops,* pp. 157-166, Apr. 2009.

[206] Rubyforge, "Heckle," http://seattlerb.rubyforge.org/heckle/, 2007.

[207] M. Sahinoglu and E.H. Spafford, "A Bayes Sequential Statistical Procedure for Approving Software Products," *Proc. IFIP Conf. Approving Software Products,* pp. 43-56, Sept. 1990.

[208] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient Mutation Testing by Checking Invariant Violations," *Proc. Int'l Symp. Software Testing and Analysis,* July 2009.

[209] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *Proc. 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 263-272, 2005.

[210] Y. Serrestou, V. Beroulle, and C. Robach, "Functional Verification of RTL Designs Driven by Mutation Testing Metrics," *Proc. 10th Euromicro Conf. Digital System Design Architectures, Methods and Tools,* pp. 222-227, Aug. 2007.

[211] Y. Serrestou, V. Beroulle, and C. Robach, "Impact of Hardware Emulation on the Verification Quality Improvement," *Proc. IFIP WG 10.5 Int'l Conf. Very Large Scale Integration of System-on-Chip,* pp. 218-223, Oct. 2007.

[212] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-Based SQL Injection Vulnerability Checking," *Proc. Eighth Int'l Conf. Quality Software,* pp. 77-86, Aug. 2008.

[213] H. Shahriar and M. Zulkernine, "Mutation-Based Testing of Buffer Overflow Vulnerabilities," *Proc. Second Ann. IEEE Int'l Workshop Security in Software Eng.,* pp. 979-984, July-Aug. 2008.

[214] H. Shahriar and M. Zulkernine, "Mutation-Based Testing of Format String Bugs," *Proc. 11th IEEE High Assurance Systems Eng. Symp.,* pp. 229-238, Dec. 2008.

[215] H. Shahriar and M. Zulkernine, "MUTEC: Mutation-Based Testing of Cross Site Scripting," *Proc. Fifth Int'l Workshop Software Eng. for Secure Systems,* pp. 47-53, May 2009.

[216] D.P. Sidhu and T.K. Leung, "Fault Coverage of Protocol Test Methods," *Proc. IEEE INFOCOM,* pp. 80-85, Mar. 1988.

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

[218] B.H. Smith and L. Williams, "An Empirical Evaluation of the MuJava Mutation Operators," *Proc. Third Workshop Mutation Analysis, published with Proc. Second Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 193-202, Sept. 2007.

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

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

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

[222] 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, Dec. 1999.

[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," *Proc. 33rd Hawaii Int'l Conf. System Sciences,* vol. 8, p. 8011, Jan. 2000.

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

[225] T. Srivatanakul, J.A. Clark, S. Stepney, and F. Polack, "Challenging Formal Specifications by Mutation: A CSP Security Example," *Proc. 10th Asia-Pacific Software Eng. Conf.,* pp. 340-350, Dec. 2003.

[226] T. Sugeta, J.C. Maldonado, and W.E. Wong, "Mutation Testing Applied to Validate SDL Specifications," *Proc. 16th IFIP Int'l Conf. Testing of Comm. Systems,* p. 2741, Mar. 2004.

[227] A. Sung, J. Jang, and B. Choi, "Fault-Based Interface Testing between Real-Time Operating System and Application," *Proc. Second Workshop Mutation Analysis,* p. 8, Nov. 2006.

[228] A. Tanaka, "Equivalence Testing for Fortran Mutation System Using Data Flow Analysis," master's thesis, Georgia Inst. of Technology, 1981.

[229] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet, "An Experimental Study on Software Structural Testing: Deterministic versus Random Input Generation," *Proc. 25th Int'l Symp. Fault-Tolerant Computing,* pp. 410-417, June 1991.

[230] N. Tillmann and J. de Halleux, "Pex-White Box Test Generation for .NET," *Proc. Second Int'l Conf. Tests and Proofs,* pp. 134-153, Apr. 2008.

[231] M. Trakhtenbrot, "New Mutations for Evaluation of Specification and Implementation Levels of Adequacy in Testing of Statecharts Models," *Proc. Third Workshop Mutation Analysis, published with Proc. Second Testing: Academic and Industrial Conf. Practice and Research Techniques,* pp. 151-160, Sept. 2007.

[232] U. Trier, "DBLP," http://www.informatik.uni-trier.de/ley/db/, 2009.

[233] J. Tuya, M.J.S. Cabal, and C. de la Riva, "SQLMutation: A Tool to Generate Mutants of SQL Database Queries," *Proc. Second Workshop Mutation Analysis,* p. 1, Nov. 2006.

[234] 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, Apr. 2007.

[235] R.H. Untch, "Mutation-Based Software Testing Using Program Schemata," *Proc. 30th Ann. Southeast Regional Conf.,* pp. 285-291, 1992.

[236] R.H. Untch, "Schema-Based Mutation Analysis: A New Test Data Adequacy Assessment Method," PhD thesis, Clemson Univ., Dec. 1995.

[237] R.H. Untch, A.J. Offutt, and M.J. Harrold, "Mutation Analysis Using Mutant Schemata," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 139-148, 1993.

[238] G. Vigna, W. Robertson, and D. Balzarotti, "Testing Network-Based Intrusion Detection Signatures Using Mutant Exploits," *Proc. 11th ACM Conf. Computer and Comm. Security,* pp. 21-30, 2004.

[239] P. Vilela, M. Machado, and W.E. Wong, "Testing for Security Vulnerabilities in Software," *Proc. Conf. Software Eng. and Applications,* 2002.

[240] 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, Nov. 2001.

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

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

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

[244] K.S.H.T. Wah, "An Analysis of the Coupling Effect I: Single Test Data," *Science of Computer Programming,* vol. 48, nos. 2-3, pp. 119-161, Aug.-Sept. 2003.

[245] R. Wang and N. Huang, "Requirement Model-Based Mutation Testing for Web Service," *Proc. Fourth Int'l Conf. Next Generation Web Services Practices,* pp. 71-76, Oct. 2008.

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

[247] E.J. Weyuker, "On Testing Non-Testable Programs," *The Computer J.,* vol. 25, pp. 456-470, 1982.

[248] W.E. Wong, "On Mutation and Data Flow," PhD thesis, Purdue Univ., 1993.

[249] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Software: Practice and Experience,* vol. 28, pp. 347-369, 1998.

[250] 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," *J. Systems and Software,* vol. 48, no. 2, pp. 79-89, Oct. 1999.

[251] W.E. Wong and A.P. Mathur, "Fault Detection Effectiveness of Mutation and Data Flow Testing," *Software Quality J.,* vol. 4, no. 1, pp. 69-83, Mar. 1995.

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

[253] M.R. Woodward, "Mutation Testing—An Evolving Technique," *Proc. IEE Colloquium on Software Testing for Critical Systems,* pp. 3/1-3/6, June 1990.

[254] M.R. Woodward, "OBJTEST: An Experimental Testing Tool for Algebraic Specifications," *Proc. IEE Colloquium on Automating Formal Methods for Computer Assisted Prototying,* p. 2, Jan. 1990.

[255] M.R. Woodward, "Errors in Algebraic Specifications and an Experimental Mutation Testing Tool," *Software Eng. J.*, vol. 8, no. 4, pp. 221-224, July 1993.
[256] M.R. Woodward, "Mutation Testing—Its Origin and Evolution," *J. Information and Software Technology*, vol. 35, no. 3, pp. 163-169, Mar. 1993.
[257] M.R. Woodward and K. Halewood, "From Weak to Strong Dead or Alive? An Analysis of Some Mutationtesting Issues," *Proc. Second Workshop Software Testing, Verification, and Analysis*, pp. 152-158, July 1988.
[258] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Mutation Analysis of Parameterized Unit Tests," *Proc. Fourth Int'l Workshop Mutation Analysis, published with Proc. Second Int'l Conf. Software Testing, Verification, and Validation Workshops*, pp. 177-181, Apr. 2009.
[259] W. Xu, A.J. Offutt, and J. Luo, "Testing Web Services by XML Perturbation," *Proc. 16th IEEE Int'l Symp. Software Reliability Eng.*, pp. 257-266, July 2005.
[260] H. Yoon, B. Choi, and J.O. Jeon, "Mutation-Based Inter-Class Testing," *Proc. Fifth Asia Pacific Software Eng. Conf.*, pp. 174-181, Dec. 1998.
[261] C.N. Zapf, "A Distributed Interpreter for the Mothra Mutation Testing System," master's thesis, Clemson Univ., 1993.
[262] Y. Zhan and J.A. Clark, "Search-Based Mutation Testing for Simulink Models," *Proc. Conf. Genetic and Evolutionary Computation*, pp. 1061-1068, June 2005.
[263] S. Zhang, T.R. Dean, and G.S. Knight, "Lightweight State Based Mutation Testing for Security," *Proc. Third Workshop Mutation Analysis, published with Proc. Second Testing: Academic and Industrial Conf. Practice and Research Techniques*, pp. 223-232, Sept. 2007.
[264] C. Zhou and P. Frankl, "Mutation Testing for Java Database Applications," *Proc. Second Int'l Conf. Software Testing Verification and Validation*, pp. 396-405, Apr. 2009.

Yue Jia received the BSc degree from Beijing Union University, China, and the MSc degree from King's College London, United Kingdom. He is a PhD student in the CREST Centre at University College London, under the supervision of Professor Mark Harman. His MSc work resulted in the KClone tool for lightweight dependence-based clone detection and the PhD work concerns Higher Order Mutation Testing, a topic on which he has published several papers, one of which received the Best Paper Award at the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation. His work on Higher Order Mutation is implemented in the Mutation Testing tool MiLu, a publicly available tool that offers configurable mutation testing for the C language. He also maintains the Mutation Testing Repository, from which the reader can find many resources, including papers and analysis of trends in research and practice of Mutation Testing. His interests are software testing and search-based software engineering. He is a student member of the IEEE.

Mark Harman is a professor of software engineering in the Department of Computer Science at University College London. He is widely known for work on source code analysis and testing and was instrumental in the founding of the field of search-based software engineering (SBSE). He has given 15 keynote invited talks on SBSE and its applications in the past four years. He is the author of more than 150 refereed publications, on the editorial board of six international journals, and has served on 90 program committees. He is director of the Crest Centre at the University College London. He is a member of the IEEE.