SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–51
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

A Systematic Literature Review of How Mutation Testing Supports Test Activities

Qianqian Zhu*, Annibale Panichella*, Andy Zaidman*

Software Engineering Research Group, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands.

SUMMARY

Mutation testing has been very actively investigated by researchers since the 1970s and remarkable advances have been achieved in its concepts, theory, technology and empirical evidence. While the latest realisations have been summarised by existing literature review, we lack insight into how mutation testing is *actually* applied. Our goal is to identify and classify the main applications of mutation testing and analyse the level of replicability of empirical studies related to mutation testing. To this aim, this paper provides a systematic literature review on the *application perspective* of mutation testing based on a collection of 159 papers published between 1981 and 2015. In particular, we analysed in which testing activities mutation testing is used, which mutation tools and which mutation operators are employed. Additionally, we also investigated how the core inherent problems of mutation testing, i.e. the equivalent mutant problem and the high computational cost, are addressed during the actual usage. The results show that most studies use mutation testing as an assessment tool targeting unit tests, and many of the supporting techniques for making mutation testing applicable in practice are still underdeveloped. Based on our observations, we made nine recommendations for the future work, including an important suggestion on how to report mutation testing in testing experiments in an appropriate manner. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: mutation testing; systematic literature review;application

1. INTRODUCTION

Mutation testing is defined by Jia and Harman [1] as a fault-based testing technique which provides a testing criterion called the *mutation adequacy score*. This score can be used to measure the effectiveness of a test set in terms of its ability to detect faults [1]. The principle of mutation testing is to introduce syntactic changes into the original program to generate faulty versions (called *mutants*) according to well-defined rules (mutation operators) [2]. Mutation testing originated in the 1970s with works from Lipton [3], DeMillo et al. [4] and Hamlet [5] and has been a very active research field over the last few decades. The activeness of the field is in part evidenced by the extensive

*Correspondence to: Software Engineering Research Group, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands. Email address:{qianqian.zhu,a.panichella,a.e.zaidman}@tudelft.nl

Copyright © 2010 John Wiley & Sons, Ltd.

survey of more than 390 papers on mutation testing that Jia and Harman published in 2011 [1]. Jia and Harman's survey highlights the research achievements that have been made over the years, including the development of tools for a variety of languages and empirical studies performed [1]. Additionally, they highlight some of the actual and inherent problems of mutation testing, amongst others: (1) the high computational cost caused by generating and executing the numerous mutants and (2) the tremendous time-consuming human investigation required by test oracle problem and equivalent mutants detection.

While Jia and Harman's survey provides us with a great overview of the latest realisations in research, we lack insight into how mutation testing is actually *applied*. Specifically, we are interested in analysing in which testing activities mutation testing is used, which mutation tools are employed and which mutation operators are used. Additionally, we want to investigate how the aforementioned problems of the high computational cost and the considerable human effort required are dealt with when applying mutation testing. In order to steer our research, we aim to fulfil the following objectives:

- to identify and classify the applications of mutation testing in testing activities;
- to analyse how the main problems are coped with when applying mutation testing;
- to provide guidelines for applying mutation testing in testing experiments;
- to identify gaps in current research and to provide recommendations for future work.

As systematic literature reviews have been shown to be good tools to summarise existing evidence concerning a technology and identify gaps in current research [6], we will follow this approach for reaching our objectives. We only consider the articles which provide sufficient details on how mutation testing is used in their studies, which require at least brief specification about the adopted mutation tool, mutation operators or mutation score. Moreover, we selected only papers that use Mutation Testing as a tool for evaluating or improving other testing activities rather than focusing on the development of mutation tools, operators or challenges and open issues for mutation analysis. This resulted in a collection containing 159 papers published from 1981 to 2015. We analysed this collection in order to answer the following two research questions:

RQ1: *How mutation testing is used in testing activities?*

This research question aims to identify and classify the main software testing tasks where mutation testing is applied. In particular, we are interested in the following key aspects: (1) in which circumstances mutation testing is used (e.g. assessment tool), (2) which testing activities are involved (e.g. test data generation, test case prioritisation), (3) which test level it targets (e.g. unit level) and (4) which testing strategies it supports (e.g. structural testing). The above four detailed aspects are defined to characterise the essential features related to usage of mutation testing and the testing activities involved. With these elements in place, we can provide an in-depth analysis of the applications of mutation testing.

RQ2: *How are empirical studies related to the mutation testing designed and reported?*

The objective of this question is to synthesise empirical evidence related to mutation testing. The case studies or experiments play an inevitable role in a research study. The design and demonstration of the evaluation methods should ensure the replicability. The replicability means that the subject, the basic methodology, as well as the result, should be clearly pointed out in the article. In particular, we are interested in how the articles report the following information related to mutation testing:

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

(1) mutation tools, (2) mutation operators, (3) mutant equivalence problem, (4) techniques for reduction of computational cost and (5) subject programs used in the case studies. After gathering this information, we can draw conclusions from the distributions of related techniques adopted under the above five facets and thereby provide guidelines for applying mutation testing.

The remainder of this review is organised as follows: Section 2 provides an overview on background notions on Mutation testing. Section 3 details the main procedures we followed to conduct the systematic literature review and describes our inclusion and exclusion criteria. Section 4 presents the discussion of the our findings, particularly Section 4.3 summarises the answers to the research questions while Section 4.4 provides the recommendation for the future research. Section 5 discusses the threats to validity, and the Section 6 concludes the paper.

2. BACKGROUND

In order to level the playing field, we first provide the basic concepts related to mutation testing, i.e., its fundamental hypothesis and generic process, including the *Competent Programmer Hypothesis*, the *Coupling Effect*, *mutation operators* and the *mutation score*. Subsequently, we discuss the benefits and limitations of mutation testing. After that, we present a historical overview of mutation testing where we mainly address the studies that concern the application of mutation testing.

2.1. Basic Concepts

2.1.1. Fundamental Hypothesis Mutation testing starts with the assumption of the *Competent Programmer Hypothesis* (introduced by Demillo et al. [4] in 1978): “The competent programmers create programs that are close to being correct.” This hypothesis implies that the potential faults in the programs delivered by the competent programmers are just very simple mistakes; these defects can be corrected by a few simple syntactical changes. Inspired by the above hypothesis, mutation testing typically applies small syntactical changes to original programs, thus implying that the faults that are seeded resemble faults made by “competent programmers”.

At first glance, it seems that the programs with complex errors cannot be explicitly generated by mutation testing. However, the *Coupling Effect*, which was coined by Demillo et al. [4] 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”. This means complex faults are *coupled* to simple faults. This hypothesis was later supported by Offutt [7, 8] through empirical investigations over the domain of mutation testing. In his experiments, he used first order mutants, which are created by applying the mutation operator to the original program once, to represent simple faults. Conversely, higher-order mutants, which are created by applying the mutation operator to the original program more than once, stand for complex faults. The results showed that the test data generated for 1-order mutants killed a higher percentage of mutants when applied to higher-order mutants thus yielding positive empirical evidence about the *Coupling Effect*. Besides, there has been a considerable effort in validating the coupling effect hypothesis, amongst others the theoretical studies of Wah [9–11] and Kapoor [12].

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

2.1.2. The Generic Mutation Testing Process After introducing the fundamental hypotheses of mutation testing, we are going to give a detailed description of the *generic process* of mutation testing:

> *Given a program P and a test suite T, a mutation engine makes syntactic changes (defined as mutation operators) to the program P, thereby generating a set of mutant programs $\mathbb{M}$. After that, each mutant $P_m \in \mathbb{M}$ is executed against T to verify whether tests fail or not.*

Here is an example of a mutation operator, i.e. Arithmetic Operator Replacement (AOR), on a statement "X = a + b". The produced mutants include "X = a - b", "X = a × b" and "X = a ÷ b".

The execution results of T on $P_m \in \mathbb{M}$ are compared with P: (1) if the output of P_m is different from P, then P_m is *killed* by T; (2) otherwise, i.e. the output of P_m is same as P, this leads to either (2.1) P_m is *equivalent* to P, which means that they are syntactically different but functionally equivalent; or (2.2) T is not adequate to detect the mutants, which requires test case augmentation.

The result of mutation testing can be represented as the *mutation score* (also referred as mutation coverage or mutation adequacy), which is defined as:

$$mutation\ score = \frac{\#\ killed\ mutants}{\#\ nonequivalent\ mutants} \tag{1}$$

From the above equation, we can see that mutant equivalence detection is done before calculating the mutation score, as the denominator explicitly mentions nonequivalent mutants. Budd and Angluin [13] have theoretically proven that deciding the equivalence of two programs is not measurable. Meanwhile, in their systematic literature survey Madeyski et al. [14] have also indicated that the equivalent mutant problem takes an enormous amount of time in practice.

A mutation testing system in large part can be regarded as a language system [15] since the programs under test must be parsed, modified and executed. The main components of mutation testing consist of the mutant creation engine, the equivalent mutant detector and the test execution runner. The first prototype of a mutation testing system for Fortran was proposed by Budd and Sayward [16] in 1977. Since then, numerous mutation tools have been developed for different languages, such as Mothra [17] for Fortran, Proteum [18] for C, Mujava [19] for Java, and SQLMutation [20] for SQL.

2.1.3. Benefits & Limitations Mutation testing is widely considered as a "high end" test criterion [15]. This is in part due to the fact that mutation testing is extremely hard to satisfy because of the massive number of mutants. However, many empirical studies found that it is much stronger than other test adequacy criteria in terms of fault exposing capability, e.g. Mathur and Wong [21], Frankl et al. [22] and Li et al. [23]. In addition to comparing mutation testing with other test criteria, there have also been empirical studies comparing real faults and mutants. The most well-known research work on such a topic is by Andrews et al. [24]: they suggest that when using carefully selected mutation operators and after removing equivalent mutants, mutants can provide a good indication of the fault detection ability of a test suite. As a result, we consider the benefits of mutation testing to be:

- better fault exposing capability compare to other test coverage criteria, e.g. all-use

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

- a good alternative to real faults which can provide a good indication of the fault detection ability of a test suite

The limitations of mutation testing are inherent. Firstly, both the generation and execution of a vast number of mutants are computationally expensive. Secondly, the equivalent mutants detection is also an inevitable stage of mutation testing which is a prominent undeterminable problem, thereby requiring human effort to investigate. We thus consider the major limitations of mutation testing to be:

- the high computational cost caused by the tremendous amount of mutants
- the undecidable Equivalent Mutant Problem resulting in the difficulty of fully automating the equivalent mutant analysis

In order to deal with the above two limitations, many efforts have been made to reduce the computational cost and propose heuristic methods to detect equivalent mutants. As for the high computational cost, Offutt and Untch [25] performed a literature review in which they summarised the approaches to reduce computational cost into three strategies: *do fewer*, *do smarter* and *do faster*. These three types were later classified into two classes by Jia and Harman [1]: reduction of the generated mutants and reduction of the execution cost. Mutant sampling (e.g. [26, 27]), mutant clustering (e.g. [28, 29]) and selective mutation (e.g. [30–32]) are the most well-known techniques for reducing the number of mutants while maintaining efficacy of mutation testing to an acceptable degree. For reduction of the execution expense, researchers have paid much attention to weak mutation (e.g. [33–35]) and mutant schemata (e.g. [36, 37]).

To overcome the Equivalent Mutant Problem, there are mainly three categories classified by Madeyski et al. [14]: (1) detecting equivalent mutants, such as Baldwin and Sayward [38] (using compiler optimisations), Hierons et al. [39] (using program slicing), Martin and Xie [40] (through change-impact analysis), Ellims et al. [41] (using running profile), and du Bousquet and Delaunay [42] (using model checker); (2) avoiding equivalent mutant generation, such as Mresa and Bottaci [31] (through selective mutation), Harman et al. [43] (using program dependence analysis), and Adamopoulos et al. [44] (using co-evolutionary search algorithm); (3) suggesting equivalent mutants, such as bayesian learning [45], dynamic invariants analysis [46], and coverage change examination (e.g. [47]).

2.2. *Historical Overview*

In this part, we are going to present a chronological overview of important research in the area of mutation testing. As the focus of our review is the application perspective of mutation testing, we mainly address the studies that concern the application of mutation testing. In the following paragraphs, we will first give a brief summary of the development of mutation testing, and — due to the sheer size of the research body — we will then highlight some notable studies on applying mutation testing.

Mutation testing was initially introduced as a fault-based testing method which was regarded significantly better in detecting errors than the *covering measure* approach [48]. Since then mutation testing has been actively investigated and studied thereby resulting in remarkable advances in its concepts, theory, technology and empirical evidence. The main interests on mutation testing

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

includes (1) defining mutation operator [49], (2) developing mutation testing systems [17,19,33], (3) reducing the cost of mutation testing [30, 36], (4) overcoming the equivalent mutant detection [14], and (5) empirical studies with mutation testing [24]. For more literature on mutation testing, we refer to the existing surveys of DeMillo [50], Offutt and Untch [25], Jia and Harman [1] and Offutt [2].

In the meanwhile, mutation testing has also been applied to support other testing activities, such as test data generation and test strategy evaluation. The early application of mutation testing can be traced back to the 1980s [51–54]). Ntafos is one of the very first researchers to use mutation analysis as a measure of test set effectiveness. Ntafos applied mutation operators (e.g. constant replacement) to the source code of 14 Fortran programs [52]. The generated test suites were based on three test strategies, i.e. random testing, branch testing and data-flow testing, and were evaluated regarding mutation score.

DeMillo and Offutt [35] are the first to automate test data generation guided by fault-based testing criteria. Their method is called Constraint-based testing (CBT). They transformed the conditions under which mutants will be killed (necessity and sufficiency condition) to the corresponding algebraic constraints (using constraint template table). The test data then was automatically generated by solving the constraint satisfaction problem using heuristics. Their proposed constraint-based automatic test data generator is limited and was only validated on five laboratory-level Fortran programs. Other remarkable approaches of the automatic test data generation includes a paper by Zhang et al. [55], who adopted Dynamic Symbolic Execution, and a framework by Papadakis and Malevris [56] in which three techniques, i.e. Symbolic Execution, Concolic testing and Search-based testing, were used to support the automatic test data generation.

Apart from test data generation, mutation testing is widely adopted to assess the cost-effectiveness of different test strategies. The work above by Ntafos [52] is one of the early studies on applying mutation testing. Recently, there has been a considerable effort in the empirical investigation of structural coverage and fault-finding effectiveness, including Namin and Andrews [57] and Inozemtseva et al. [58]. Also of interest is assertion coverage proposed by Zhang and Mesbah [59] and observable modified condition/decision coverage (OMC/DC) presented by Whalen et al. [60]; these novel test criteria were also evaluated via mutation testing.

Test case prioritisation is one of the practical approaches to reducing the expense of regression testing by rescheduling test cases to expose the faults more quickly. Mutation testing has also been applied in supporting test case prioritisation. Among these studies are influential papers by Rothermel et al. [61] and Elbaum et al. [62] who proposed a new test case prioritisation method based on the rate of mutants killing. Moreover, Do and Rothermel [63, 64] measured the effectiveness of different test case prioritisation strategies via mutation faults, since Andrews et al. [24]'s empirical study suggested that mutation faults can be representative of real faults.

The test-suite reduction is another test activity we identified which is supported by mutation testing. The research work of Offutt et al. [65] is the first to target test-suite reduction strategies especially for mutation testing. They proposed *Ping-Pong* reduction heuristics to select test cases based on their mutation scores. Another notable work is Zhang et al. [66] that investigated test-suite reduction techniques on Java programs with real-world JUnit test suites via mutation testing.

Another portion of the application of mutation testing is debugging, such as fault localisation. Influential examples include an article by Zhang et al. [67] in which mutation analysis is adopted to investigate the effect of coincidental correctness upon coverage-based fault localiser, and a novel

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

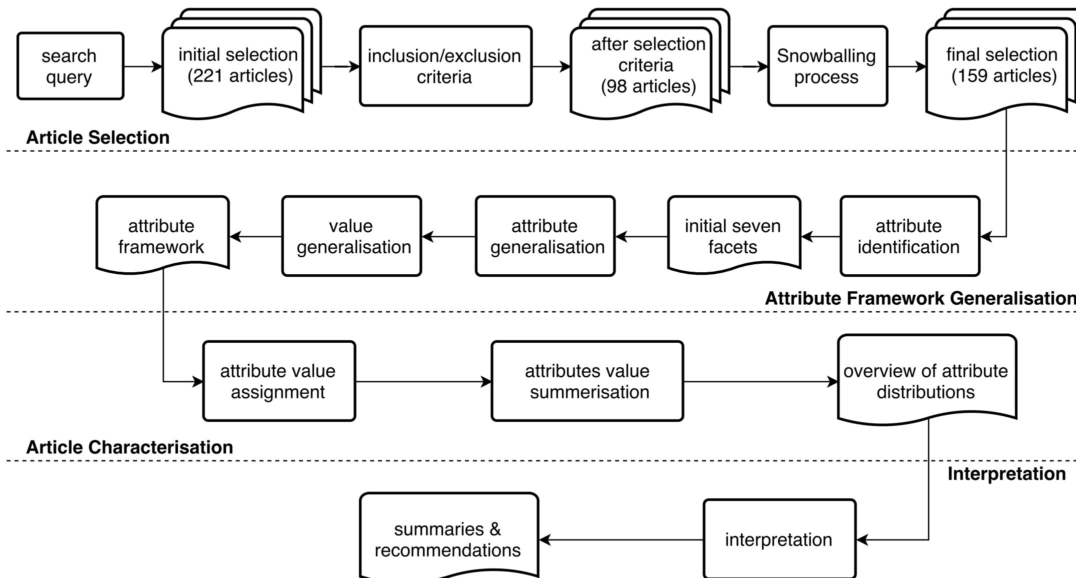

Figure 1. Overview of Systematic Review Process [72]

fault localisation method by Papadakis et al. [68], [69] who used mutants to identify the faulty program statements.

3. RESEARCH METHOD

In this section, we describe the main procedures we took to conduct this review. We adopted the methodology of the systematic literature review. A systematic literature review [6] is a means of aggregating and evaluating all the related primary studies under a research scope in an unbiased, thorough and trustworthy way. Unlike the general literature review, systematic literature review aims to eliminate bias and incompleteness through a systematic mechanism [70]. Kitchenham [6] presented comprehensive and reliable guidelines for applying the systematic literature review to the field of software engineering. The guidelines cover three main phases: (i) planning the review, (ii) conducting the review, and (iii) reporting the review. Each step is well-defined and well-structured. By following these guidelines, we can reduce the likelihood of generating biased conclusions and sum all the existing evidence in a manner that is fair and seen to be fair.

The principle of the systematic literature review [71] is to convert the information collection into a systematic research study; this research study first defines several specific research questions and then searches for the best answers accordingly. These research questions and search mechanisms (consisting of study selection criteria and data extraction strategy) are included in a review protocol, a detailed plan to perform the systematic review. After developing the review protocol, the researchers need to validate this protocol for further resolving the potential ambiguity.

Following the main stages of the systematic review, we will introduce our review procedure in four parts: we will first specify the research questions, and then present the study selection strategy and data extraction framework. In the fourth step, we will show the validation results of the review protocol. The overview of our systematic review process is shown in Figure 1.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

3.1. Research Questions

The research questions are the most critical part of the review protocol. The research questions determine study selection strategy and data extraction strategy. In this review, our objective is to examine the primary applications of mutation testing and identify the problems and gaps, therefore, we can provide guidelines for applying mutation testing and recommendations for future work. To achieve these goals and starting with our most vital interest, the application perspective of mutation testing, we naturally further divide it into two aspects: (1) how mutation testing is used and (2) how the relevant empirical studies are reported. For the first aspect, we aim to identify and classify the main applications of mutation testing:

RQ1: **How is mutation testing used in testing activities?**

In order to understand how mutation testing is used, we should first determine in which circumstances it is used. The usages might range from using mutation testing as a way to assess how other testing approaches perform or mutation testing might be a building block of an approach altogether. This leads to RQ1.1:

RQ1.1: *Which role does mutation testing play in testing activities?*

There is a broad range of specific testing activities in which mutation testing can be of help, e.g. fault localisation, test data generation, etc. RQ1.2 seeks to uncover these activities.

RQ1.2: *Which testing activities does mutation testing usually support?*

In Jia and Harman's survey [1] of mutation testing, they found that most approaches work at the unit testing level. In RQ1.3 we will investigate whether the application of mutation testing is also mostly done at the unit testing level, or whether other levels of testing are also considered important.

RQ1.3: *Which test level does mutation testing usually target?*

Jia and Harman [1] have also indicated that mutation testing is most often used in a white box testing context. In RQ1.4 we explore what other strategies can also benefit from the application of mutation testing.

RQ1.4: *Which testing strategies does mutation testing frequently support?*

For the second aspect, we are going to synthesise empirical evidence related to mutation testing:

RQ2: **How are empirical studies related to the mutation testing designed and reported?**

A plethora of mutation testing tools exist and have been surveyed by Jia and Harman [1]. Little is known which ones are most applied and why these are more popular. RQ2.1 tries to fill this knowledge gap by providing insight into which tools are used more frequently in a particular context.

RQ2.1: *Which mutation tools have been frequently used?*

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The mutation tools that we surveyed implement different mutation operators. Also, the various mutation approaches give different names to virtually the same mutation operators. RQ2.2 explores what mutation operators each method or tool has to offer and how mutation operators can be compared.

> **RQ2.2**: *Which mutation operators have been used more frequently?*

The equivalent mutant problem, i.e. the situation where a mutation leads to change that is not observable in behaviour, is one of the most significant open issues in mutation testing. Both Jia and Harman [1] and Madeyski et al. [14] highlighted some of the most remarkable achievements in the area, but we have a lack of knowledge when it comes to how the equivalent mutant problem is coped with during the actual application of mutation testing. RQ2.3 aims to seek answers for exactly this question.

> **RQ2.3**: *Which approaches are used to overcome the equivalent mutant problem more often when applying mutation testing?*

As mutation testing is computationally expensive, techniques to reduce costs are important. Selective Mutation and Weak Mutation are the most widely studied cost reduction techniques [1], but it is unclear which reduction techniques are actually used when applying mutation testing, which is the exact topic of RQ2.4.

> **RQ2.4**: *Which techniques are used to reduce the computational cost more frequently when applying mutation testing?*

In order to better understand in which context mutation testing is applied, we want to look into the programming languages that have been used in the experiments. But also the size of the case study systems is of interest, as it can be an indication of the maturity of certain tools. Finally, we are also explicitly looking at whether the case study systems are available for replication purposes (in addition to the check for availability of the mutation testing tool in RQ2.1).

> **RQ2.5**: *What are the most common subjects used in the experiments (in terms of programming language, size and data availability)?*

3.2. *Study Selection Strategy*

Initial Study Selection:

We started with searching queries in online platforms, including Google Scholar, Scopus, ACM Portal, IEEE explore as well as Springer, Wiley, Elsevier Online libraries, to collect papers containing the keywords “mutation testing” or “mutation analysis” in their titles, abstracts and keywords. Meanwhile, to ensure the high quality of the selected papers, we only considered the articles published in seven top journals and ten top conferences (as listed in Table I) dating from 1971 as data sources. The above 17 venues are chosen because we regard them to be the leaders in Software Engineering and the major venues that report a high proportion of research on software testing, debugging, software quality and validation. Moreover, we excluded article summaries, interviews, reviews, workshops, panels and poster sessions from the search. If the paper’s language is not English or its full-text is not available, we also excluded such a paper. After this step, 221 papers were initially selected.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Type	Venue Name	No. of papers After Applying Search queries	No. of papers After Applying In./Ex. Criteria	No. of papers After Snow-balling proce-dure
Journal	Transaction on Software Engineering (TSE)	19	9	19
	Journal of Empirical Software Engineering (EMSE)	4	3	5
	Journal on Software Testing, Verification and Reliability (STVR)	33	16	19
	Journal Software Maintenance and Evolution (JSME)	0	0	0
	Transaction on Reliability (TR)	1	1	1
	Transaction on Software Engineering and Methodology (TOSEM)	3	2	3
	Journal of Systems and Software (JSS)	17	8	8
	Information and Software Technology (IST)	0	0	2
	Software Quality Journal (JSQ)	0	0	2
Conference	International Conference on Software Engineering (ICSE)	29	9	16
	European Software Engineering Conference / International Symposium on the Foundations of Software Engineering (ESEC/FSE)	6	1	8
	International Conference on Software Testing, Verification, Validation (ICST)	46	24	21
	International Symposium on Software Testing and Analysis (ISSTA)	14	3	9
	International Conference on Automated Software Engineering (ASE)	7	3	6
	International Conference on Software Maintenance and Evolution (ICSME/ICSM)	6	3	9
	International Symposium on Empirical Software Engineering and Measurement (ESEM/ISESE)	2	1	3
	Proceedings International Symposium on Search Based Software Engineering (SSBSE)	0	0	0
	Proceedings International Conference on Quality Software (QSIC)	8	5	6
	International Symposium on Software Reliability Engineering (ISSRE)	26	10	20
	Proceedings Asia Pacific Software Engineering Conference (APSEC)	0	0	1
	Proceedings of the International Conference on Testing Computer Software (TCS)	0	0	1
	Total	221	98	159

Note: the venues marked in **bold** font are not initially selected, but where added after the Snowballing procedure.

Table I. Venues Involved in study selection

Inclusion/Exclusion Criteria:

Since we are interested in how mutation testing is *applied* in practice, we need selection criteria to include the papers that use mutation testing as a tool for evaluating or improving other testing activities and exclude the papers focusing on the development of mutation tools and operators, or challenges and open issues for mutation analysis. Moreover, the selected articles should also provide sufficient evidence for answering the research questions. Therefore, we define two inclusion/exclusion criteria for study selection. The inclusion/exclusion criteria are as follows:

1. The article must focus on the supporting role of mutation testing in testing activities. This criterion excludes the research *solely* on mutation testing itself, such as defining mutation

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

operators, developing mutation systems, investigating ways to solve open issues related to mutation analysis and comparisons between mutation testing and other testing techniques.

2. The article exhibits sufficient evidence that mutation testing is used to support testing related activities. The sufficient evidence means that the article must clearly describe how the mutation testing is involved in the testing activities. The author(s) must state at least one of the following details about the mutation testing in the article: mutation tool, mutation operators, mutation score. This criterion also excludes theoretical studies on mutation testing.

The first author then carefully read the titles and abstracts to check whether the papers in the initial collection belong to our set of selected papers based on the inclusion/exclusion criteria. If it is unclear from the titles and abstracts whether mutation testing was applied, the entire article especially the experiment part was read as well. After we have applied the inclusion/exclusion criteria, 98 papers remained.

Snowballing Procedure:

After selecting 98 papers from digital databases and applying our selection criteria, there is still a high potential to miss articles of interest. According to Brereton et al. [71]'s work, they pointed out that most online platforms do not provide adequate support for systematic identification of relevant papers. To overcome this shortfall of online databases, we then adopted a both backward and forward snowballing strategies [73] to find missing papers. Snowballing refers to using the list of references in a paper or the citations to the paper to identify additional papers [73]. Using the references and the citations respectively is referred to as backward and forward snowballing [73].

We used the 98 papers as the start set and performed a backward and forward snowballing procedure recursively until no further papers could be added to our set. During the snowballing procedure, we extended the initially selected venues to minimise the chance of missing related papers. The snowballing process resulted in another 61 articles (and four additional venues).

3.3. Data Extraction Strategy

Data extracted from the papers are used to answer the research questions we proposed. Based on our research questions, we draw seven facets of interest that are highly relevant to the information we need to answer the questions. The seven facets are: (1) the roles of mutation testing in testing activities; (2) the testing activities; (3) the mutation tools used in experiments; (4) the mutation operators used in experiments; (5) the description of equivalent mutant problem; (6) the description of cost reduction techniques for mutation testing; (7) the subjects involved in experiments. An overview of these facets is given in Table II.

For each facet, we first just read the corresponding details in each paper and extracted the exact text from the papers. During the reading procedure, we started to identify and classify more specific attributes of interest under each facet and assigned values to each attribute. The values of each attribute were generalised and modified during the reading process: we combined some values together or divided one into several smaller groups. In this way, we generated an attribute framework, and then we used the framework to characterise each paper. Therefore, we can show quantitative results for each attribute to support our answers. And the attribute framework can also be further used for validation and replication of the review work.

(1) roles of mutation testing in testing activities:

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

The first facet concerns the role of mutation testing in testing activities drawn from RQ1.1. We identified two classes for the function of mutation testing: assessment and guide. When mutation testing is used as a measure of test effectiveness concerning fault-finding capability, we classify this role as “assessment”. While for the “guide” role, mutation testing is adopted to improve the testing effectiveness as guidance, i.e., it is an inherent part of an approach.

To identify and classify the role of mutation testing, we mainly read the description of mutation testing in *experiment part* of each paper. If we find the phrases which have the same meanings as “evaluate fault-finding ability” or “assess the testing effectiveness” in a paper, we then classify the paper into the class of “assessment”. In particular, when used as a measure of testing effectiveness, mutation testing is usually conducted at the end of the experiment; this means mutation testing is not involved in the generation or execution of test suites. Unlike the “assessment” role, if mutation testing is adopted to help to generate test suites or run test cases, we then classify these paper into the “guide” set. In this case, mutation testing is not used in the final step of the experiment.

(2) testing activities:

The second facet focuses on testing activities. Three attributes are relevant to testing activities: the categories of testing activities (RQ1.2), test levels (RQ1.3) and testing strategies (RQ1.4). To identify the categories of testing activities, we group similar testing activities based on information in title and abstract. The testing activities we identified so far consist of 11 classes: test data generation, test-suite reduction/selection, test strategy evaluation, test case minimisation, test case prioritisation, test oracle, fault localisation, programming repairing, development scheme evaluation, model clone detection and model review. We classify the papers by reading the description appeared in *title and abstract*.

For test level, the values are based on the concept of test level and the authors’ specification. More precisely, we are considering five test levels: unit, integration, system, others and n/a. To characterise the test level, we search the exact words “unit”, “integration”, “system” in the article, as these four test levels are regular terms and cannot be replaced by other synonyms. If there is no relevant result after searching in a paper, we then classify the paper’s test level into “n/a”, i.e. no specification regarding the test level. In addition, for the paper which is difficult for us to categorise into any of the four phases, such as testing of the grammar of a programming language and spreadsheet testing, we mark this situation as “others”.

For testing strategies, coarse-grained classification is adequate to gain an overview of the distribution of testing strategies. We identified five classes according to the test design techniques: structural testing, specification-based testing, similarity-based testing, hybrid testing and others [74, 75]. Among the structural testing and specification-based testing classes, we further divided into traditional version and enhanced one based on whether the regular testing is improved by other methods.

To be classified into the “structural testing” class, the paper should either contain the keywords of “structure-based”, “code coverage-based” or “white box”, or use structural test design techniques, such as statement testing, branch testing and condition testing. For the “specification-based testing” class, the articles should either contain the keywords of “black box”, “requirement-based” or “specification-based”, or use specification-based test design techniques, such as equivalence partitioning, boundary value analysis, decision tables and state transition testing. The similarity-based method aims to maximise the diversity of test cases to improve the test effectiveness;

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

this technique is mainly based on test case relationship rather than software artefacts. Therefore, similarity-based testing does not belong to either structural testing or specification-based testing. The hybrid testing combines structural testing and specification testing together. Besides, several cases are using static analysis, code review or other techniques which cannot fit the above classes; in such situations, we mark the value as “others”.

Furthermore, to classify “enhanced” version of structural and specification-based testing we rely on whether other testing methods were adopted to improve the traditional testing. For instance, Whalen et al. [60] combined the MC/DC coverage metric with a notion of *observability* to ensure the fault propagation conditions. Papadakis and Malevris [76] proposed an automatic mutation test case generation via dynamic symbolic execution. To distinguish such instances from the traditional structural and specification-based testing, we marked as “enhanced”.

(3) mutation tools used in experiment:

For the mutation tools (derived from RQ2.1), we are interested in their types, but also in their availability. Our emphasis on tool availability is instigated to address possible replication of the studies. The values of “Yes” or “No” for the tool availability depends on whether the mutation tool is open to the public. The tool type intends to provide further analysis of mutation tool, which is based on the whether the tool is self-developed and whether the tool itself is a complete mutation testing system. We identified five types of mutation tools: existing, partially-based, self-written, hand-seeded and n/a. The “existing” tool must be a complete mutation testing system, while “partially-based” means these tools are used as a base or a framework for mutation testing. The example for “partially-based” tools are EvoSuite, jFuzz, TrpAutoRepair and GenProg. The self-written tool category represents those tools that have been developed by the authors of the study. The “hand-seeded” value means the mutants were generated manually in the studies. Besides, we defined “n/a” value in addition to the “tool types” attribute; the value of “n/a” marks the situation where lacks of a description of mutation tools including tool names/citations and whether hand-seeded or not.

(4) mutation operators used in experiment:

As for the mutation operators (related to RQ2.2), we focus on two attributes: description level and generic classification. The former is again designed to assess the repeatability issue related to mutation testing. The description degree depends on the way that the authors presented the mutation operators used in their studies, consisting of three values: “well-defined”, “not sufficient” and “n/a”. If the paper showed that the complete list of mutation operators is available, then we classify such paper into “well-defined”. The available full list includes two main situations: (1) the authors listed each name(s) of mutation operators and/or specified how the mutation operators make changes to programs in the articles; (2) the studies adopted existing tools and mentioned the used mutation operator (including the option were all or the default set of mutation operators provided by that tool were used). The well-defined category thus enables the traceability of the complete list of mutation operators. For the remaining set, the incomplete list, if there is some information about the mutation operators in the article but not enough for replication of the whole list of mutation operators, then we classify the paper into “not sufficient”. The typical example is that the author used such words as “etc.”, “such as” or “e.g.” in the specification of the mutation operators; this indicates that only some mutation operators are explicitly listed in the paper, but not all. The last value, “n/a”, means no description of the mutation operators was given in the paper at all.

Copyright © 2010 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

In order to compare the mutation operators from different tools to analyse the popularity of involved mutation operators amongst the papers, we collected the information about mutation operators mentioned in the articles. Notably, we only consider the articles which are classified as “well-defined”. We excluded the papers with “not sufficient” label as their lists of mutation operators are not complete as this might result in biased conclusions based on incomplete information. Moreover, during the reading process, we found that different mutation testing tools use slightly different names for their mutation operators. For example, in MuJava [19], the mutation operator which replaces relational operators with other relational operators is called “Relational Operator Replacement”, while that is named “Conditionals Boundary Mutator” in PIT [77]. Therefore, we saw a need to compose a generic classification of mutation operators, which enables us to more easily compare mutation operators from different tools or definitions.

The method we adopted here to generate the generic classification is to group the similar mutation operators together among all the existing mutation operators in the literature based on how they mutate the programs. Firstly, mutation testing can be applied to both program source code and program specification. Thus, we classified the mutation operators into two top-level groups: program mutation and specification mutation operators. In particular, we are more interested in the program mutation, so we further divided program mutation testing into three sub-categories: expression-level, statement-level and others. The expression-level mutation operators focus on the inner components of the statements, such as operators and operands, while the statement-level ones mutate the at least one single statement. For the “others” class, it includes mutation operators related to the programming language’s unique features, e.g. Objected-Oriented specific mutation operators. Our generic classification of mutation operators is as follows:

1. Specification mutation
2. Program mutation
   (a) Expression-level
      i. **arithmetic operator:** it mutates the arithmetic operators (including addition“+”, subtraction“–”, multiplication“*”, division “/”, modulus “%” , unary operators “+”, “–”, and short-cut operators “++”, “– –”)[†] by replacement, insertion or deletion.
      ii. **relational operator:** it mutates the relational operators (including “>”, “>=”, “<”, “<=”, “==”, “!=”) by replacement.
      iii. **conditional operator:** it mutates the conditional operators (including and “&”, or “|”, exclusive or “^”, short-circuit operator “&&”, “||”, and negation“!”) by replacement, insertion or deletion.
      iv. **shift operator:** it mutates the shift operators (including “>>”, “<<” and “>>>”) by replacement.
      v. **bitwise operator:** it mutates the bitwise operators (including bitwise and “&”, bitwise or “|”, bitwise exclusive or “^” and bitwise negation “~”) by replacement, insertion or deletion.
      vi. **assignment operator:** it mutates the assignment operators (including the plain operator “=” and short-cut operators “+=”, “– =”, “*=”, “/=”, “%=”, “&=”, “|=”,

[†]The syntax of these operators might vary slightly in different languages. Here we just used the operators in Java as an example. So as the same in (ii) - (vi) operators.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

"^=", "<<=", ">>=", ">>>=") by replacement. Besides, the plain operator "=" is also changed to "==" in some cases.

vii. **absolute value:** it mutates the arithmetic expression by preceding unary operators including ABS (computing the absolute value), NEGABS (compute the negative of the absolute value) and ZPUSH (testing whether the expression is zero. If the expression is zero, then the mutant is killed; otherwise execution continues and the value of the expression is unchanged)[‡].

viii. **constant:** it changes the literal value including increasing/decreasing the numeric values, replacing the numeric values by zero or swapping the boolean literal (`true`/`false`).

ix. **variable:** it substitutes a variable with another already declared variable of the same type and/or of the compatible type.

x. **type:** it replaces a type with the other compatible types including type casting.[§]

xi. **conditional expression:** it replaces the conditional expression by `true`/`false` so that the statements following the conditional always execute or skip.

xii. **parenthesis:** it changes the precedence of the operation by deleting, adding or removing the parentheses.

(b) Statement-level

i. **`return` statement:** it mutates `return` statement in the method calls including `return` value replacement or `return` statement swapping.

ii. **`switch` statement:** it mutates `switch` statements by making different combinations of the `switch` labels (`case`/`default`) or the corresponding block statement.

iii. **`if` statement:** it mutates `if` statements including removing additional semicolons after conditional expressions, adding an `else` branch or replacing last `else if` symbol to `else`.

iv. **statement deletion:** it deletes statements including removing the method calls or removing each statement[¶].

v. **statement swap:** it swaps the sequences of statements including rotating the order of the expressions under the use of the *comma* operator, swapping the contained statements in `if-then-else` statements and swapping two statements in the same scope.

vi. **brace:** it moves the closing brace up or down by one statement.

vii. **`goto` label:** it changes the destination of the `goto` label.

viii. **loop trap:** it introduces a guard (trap after n^{th} loop iteration) in front of the loop body. The mutant is killed if the guard is evaluated the nth time through the loop.

[‡]The definition of this operator is from the Mothra [17] system. In some cases, this operator only applies the absolute value replacement.

[‡]The types of the variables varies in different programming languages.

[§]The changes between the objects of the parent and the child are excluded which belongs to "OO-specific"

[¶]To maintain the syntactical validity of the mutants, semicolons or other symbols, such as `continue` in Fortran, are retained.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

ix. **bomb statement:** it replaces each statement by a special `Bomb()` function. The mutant is killed if the `Bomb()` function is executed which ensures each statement is reached.
x. **control-flow disruption (break/continue):** it disrupts the normal control flow by adding, removing, moving or replacing `continue`/`break` labels.
xi. **exception handler:** it mutates the exception handlers including changing the `throws`, `catch` or `finally` clauses.
xii. **method call:** it changes the number or position of the parameters/arguments in a method call, or replace a method name with other method names that have the same or compatible parameters and result type.
xiii. **`do` statement:** it replaces `do` statements with `while` statements.
xiv. **`while` statement:** it replaces `while` statements with `do` statements.

(c) Others

i. **OO-specific:** the mutation operators related to O(bject)-O(riented) Programming features [78], such as Encapsulation, Inheritance and Polymorphism, e.g. `super` keyword insertion.
ii. **SQL-specific:** the mutation operators related to SQL-specific features [20], e.g. replacing **SELECT** to **SELECT DISTINCT**.
iii. **Java-specific**[‖]**:** the mutation operators related to Java-specific features [78] (the operators in Java-Specific Features), e.g. `this` keyword insertion.
iv. **JavaScript-specific:** the mutation operators related to JavaScript-specific features [79] (including DOM, JQUERY, and XMLHTTPREQUEST operators), e.g. `var` keyword deletion.
v. **SpreadSheet-specific:** the mutation operators related to SpreadSheet-specific features [80], e.g. changing range of cell areas.
vi. **AOP-specific:** the mutation operators related to A(spect)-O(riented)-P(rogramming) features [81, 82], e.g. removing pointcut.
vii. **concurrent mutation:** the mutation operators related to concurrent programming features [83, 84], e.g. replacing `notifyAll()` with `notify()`.
viii. **Interface mutation:** the mutation operators related to Interface-specific features [85, 86], suitable for use during integration testing.

(5) description of the equivalent mutant problem & (6) description of cost reduction techniques for mutation testing:

The fifth and sixth facets aim to show how the most significant problems are coped with when applying mutation testing (related to RQ2.3 and RQ2.4 respectively). We composed the list of techniques based on both our prior knowledge and the descriptions given in the papers. We identified seven methods for dealing with the equivalent mutant problem and five for reducing computational cost except for "n/a" set (more details are given in Table II).

[‖] This set of mutation operators originated from Java features but not limited to Java language, since other languages can share certain features, e.g., `this` keyword is also available in C++ and C#, and `static` modifier is supported by C and C++ as well.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

For the equivalent mutant problem, we started by searching the keywords “equivalen*” and “equal” in each paper to target the context of equivalent mutants issue. Then we extracted the corresponding text from the articles. If there are no relevant findings in a paper, we mark this article as “n/a” which means the authors did not mention how they overcame the equivalent mutant problem. Here it should be noted that we only considered the description related to the equivalent mutant problem given by the authors; this means we excluded the internal heuristic mechanisms adopted by the existing tools if the author did not point out such internal approaches. For example, the tool of JAVALANCHE [87] ranks mutations by impact to help users detect the equivalent mutants. But if the authors who used JAVALANCHE did not specify that internal feature, we will not label the paper into the class that used the approach of “ranking the mutations”.

For the cost reduction techniques, we read the experiment part carefully to extract the reduction mechanism from the papers. In addition, we excluded the runtime optimisation and selective mutation. Because the former one, runtime optimisation, is an internal optimisation adopted during the tool implementation, thereby such information is more likely to be reported in the tool documentation. We did not consider the runtime optimisation to avoid incomplete statistics. As for the second one, selective mutation, we assume it is adopted by all papers since it is nearly impossible to implement and use all the operators in practice. If a paper does not contain any description of the reduction methods in the experiment part, we mark this article as “n/a”.

(7) subjects involved in the experiment:

For the subject programs in the evaluation part, we are interested in three aspects: programming language, size and data availability. From the programming language, we can obtain an overall idea of how established mutation testing is in each programming language domain and what the current gap is. From the subject size, we can see the scalability issue related to mutation testing. From the data availability situation, we can assess the replicability of the studies.

For the programming language, we extracted the programming language of the subjects involved in the experiment in these articles, such as Java, C, SQL, etc. If the programming language of the subject programs is not clearly pointed out, we mark it as “n/a”. Note, more than one languages might be involved in a single experiment.

For the subject size, we defined four categories according to the lines of code (LOC): preliminary, small, medium and large. If the subject size is less than 100 LOC, then we classify it into the “preliminary” category. If the size is between 100 to 10K LOC, we consider it “small”, while between 10K and 1M LOC we appraised it as “medium”. If the size is greater than 1M LOC, we consider it as “large”. Since our size scale is based on LOC, if the LOC of the subject is not given, or other metrics are used, we mark it as “n/a”. To assign the value to a paper, we always take the biggest subjects used in the papers.

For the data available, we defined two classes: Yes and No. “Yes” means *all* subjects in the experiments can be openly accessible; this can be identified either from the keywords “open source”, `SIR` [88], `GitHub`[**], `SF100` [89] or `SourceForge`[††], or from the open link provided by the authors. It is worth noting that if one of the subjects used in a study is not available, we classify the paper into “No”.

**https://github.com/
††https://sourceforge.net/

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

The above facets of interest and corresponding attributes and detailed specification of values are listed in Table II.

Facet	Attribute	Value	Description
Roles	classification	assessment	assessing the fault-finding effectiveness
		guide	improving other testing activities as guidance
Testing Activities	category	test data generation	creating test input data
		test-suite reduction/selection	reducing the test suite size while maintaining its fault detection ability
		test strategy evaluation	evaluating test strategies by carrying out the corresponding whole testing procedure, including test pool creation, test case selection and/or augmentation and testing results analysis.
		test-case minimisation	simplifying the test case by shortening the sequence and removing irrelevant statements
		test case prioritisation	reordering the execution sequence of test cases
		test oracle	generating or selecting test oracle data
		fault localisation	identifying the detective part of a program given the test execution information
		program repairing	generating patches to correct detective part of a program
		development scheme evaluation	evaluating the practice of software development process via observational studies or controlled experiments, such as Test driven development (TDD)
		model clone detection	identifying similar model fragments within a given context
		model review	determining the quality of the model at specification level using static analysis techniques
	test level	unit	testing activities focus on unit level. Typical example of unit testing includes: using unit testing tools, such as Junit and Nunit, intra-method testing, intra-class testing.
		integration	testing activities focus on integration level. Typical example of integration testing includes: caller/callee and inter-class testing
		system	testing activities focus on system level. Typical examples of system testing includes: high-level model-based testing techniques and high-level specification abstraction methods
		others	testing activities are not related to source code. Typical example includes: grammar.
		n/a	no specification about the testing level in the article.
	testing strategy	structural	white-box testing, uses the internal structure of the software to derive test cases, such as statement testing, branch testing and condition testing
		enhanced structural	adopting other methods to improve the traditional structural testing, mutation-based techniques, information retrieval knowledge, observation notations and assertion coverage
		specification-based	viewing software as a black box with input and output, such as equivalence partitioning, boundary value analysis, decision tables and state transition testing
		enhanced specification-based	adopting other methods to improve the traditional specification-based testing, such as mutation testing.
		similarity-based	maximising the diversity of test cases to improve the test effectiveness
		hybrid	combining structural testing and specification testing together
		others	using static analysis, or focusing on other testing techniques which cannot fit in above six classes
Mutation Tools	availability	Yes/No	Yes: open to the public; No: no valid open access
	type	existing tool	a complete mutation testing system
		partially-based	used as a base or framework for mutation testing
		self-written	developed by the authors and the open link of the tool is also accessible
		hand-seeded	generating mutants manually based on the mutation operators
		n/a	no description of the adopted mutation testing tool
Mutation Operators	description Level	well-defined	the complete list of mutation operators is available
		not sufficient	the article provides some information about mutation operators but the information is not enough for replication
		n/a	no description of the mutation operators
	generic classification	refer to Section 3.3 (4)	refer to Section 3.3 (4)

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Equivalence Solver	methods	not killed as equivalent	treating mutants not killed as equivalent
		not killed as nonequivalent	treating mutants not killed as nonequivalent
		no investigation	no distinguishing between equivalent mutants and nonequivalent ones
		manual	manual investigation
		model checker	using model checker to remove functionally equivalent mutants
		reduce likelihood	generating mutants that are less likely to be equivalent, such as using behaviour-affecting variables, carefully-designed mutation operators and constraints binding
		deterministic model	adopting the deterministic model to make the equivalence problem decidable
		n/a	no description of mutant equivalence detector
Reduction Technique	methods	mutant sample	randomly select a subset of mutants for testing execution based on fixed selection ratio
		fixed number	select a subset of mutants based on fixed number
		weak mutation	compare internal state of the mutant and the original program immediately after the mutated statement(s)
		higher-order	reduce the number of mutants by selecting higher-order mutants which contains more than one faults
		selection strategy	generate less mutants by selecting where to mutate based on random algorithm or other techniques
		n/a	no description of reduction techniques (except for runtime optimisation and selective mutation)
Subject	language	Java, C, C#, etc.	various programming languages
	size (maximum)	preliminary	$<$100 LOC
		small	100 $\sim$ 10K LOC
		medium	10K $\sim$ 1M LOC
		large	$>$ 1M LOC
		n/a	no description of program size regarding LOC
	availability	Yes/No	Yes: open to the public; No: no valid open access

Table II. Attribute Framework

3.4. Review Protocol Validation

The review protocol is a critical element of a systematic literature review and researchers need to specify and carry out procedures for its validation [71]. The validation procedure aims to eliminate the potential ambiguity and unclear points in the review protocol specification. In this review, we conduct the review protocol validation among the three authors. We also used the results to improve our review protocol. The validation focuses on two things: selection criteria and attribute framework, including the execution of two pilot runs of study selection procedure and data extraction process.

3.4.1. Selection Criteria Validation We performed a pilot run of the study selection process, for which we randomly generated ten candidate papers from selected venues (including articles out of our selection scope) and carried out the paper selection among three authors independently based on the inclusion/exclusion criteria. After that, the three authors compared and discussed the selection results. The results show that for 9 out of 10 papers, the authors had an immediate agreement. The three authors discussed the one paper that showed disagreement, leading to a revision of the first inclusion/exclusion criterion. In the first exclusion criterion, we added “solely” to the end of the sentence “...This criterion excludes the research on mutation testing itself...”. By adding “solely” to the first criterion, we include articles whose main focus is mutation testing, but also cover the application of mutation testing.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

3.4.2. Attribute Framework Validation To execute the pilot run of the data extraction process, we randomly select ten candidate papers from our selected collection. These 10 papers are classified by all three authors independently using the attribute framework that we defined earlier. The discussion that follows from this process leads to revisions of our attribute framework. Firstly, we clarified that the information extracted from the papers must have exactly the same meaning as described by the authors; this mainly means that we cannot further interpret the information. If the article does not provide any clear clue for a certain attribute, we use the phrase “not specified” (“n/a”) to mark this situation. By doing so, we can minimise the potential misinterpretation of the articles.

Secondly, we ensured that the values of the attribute framework are as complete as possible, so that for each attribute we can always select a value. For instance, when extracting testing activities information from the papers, we can simply choose one or several options of the 11 categories provided by the predefined attribute framework. The purpose of providing all possible values to each attribute is to assist data extraction in an unambiguous and trustworthy manner. Through looking at the definitions of all potential values for each attribute, we can easily target unclear or ambiguous points in data extraction strategy. If there are missing values for certain attributes, we can only add the additional data definition to extend the framework. The attribute framework can also be of clear guideline for future replication. Furthermore, we can then present quantitative distributions for each attribute in later discussion to support our answers to research questions.

To achieve the above two goals, we made revisions to several attributes as well as values. The specified modifications are listed as follows:

Mutation Tools: Previously, we combined tool availability and tool types together by defining three values: self-written, existing and not available; this is not clear to distinguish available tools from unavailable ones. Therefore, we further defined two attributes, i.e., tool availability and tool types.

Mutation Operators: We added “description level” to address the interest of how mutation operators are specified in the articles; this also helps in the generalisation of mutation operator classification.

Reduction Techniques: We added the “fixed number” value to this attribute.

Subjects: We changed the values of “data availability” from “open source”, “industrial” or “self-defined” to “Yes” or “No”. Since the previous definitions can not distinguish between available dataset and unavailable ones.

4. REVIEW RESULTS

After developing the review protocol, we conducted the task of article characterisation accordingly. Given the attribute assignment under each facet, we are now at the stage of interpreting the observations and reporting our results. In the following section, we discuss our review results following the sequence of our research questions. While Section 4.1 deals with the observations related to how mutation testing is applied (RQ1), Section 4.2 will present the RQ2-related discussion. For each sub-research question, we will first show the distribution of the relevant

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Testing Activity	Assessment	Guide	Total
test data generation	31	28	59
test strategy evaluation	55	3	58
test case prioritisation	10	6	16
test oracle	9	4	13
test-suite selection/reduction	9	4	13
fault localisation	7	4	11
program repairing	1	1	2
test case minimisation	1	1	2
development scheme evaluation	0	1	1
model clone detection	1	0	1
model review	1	0	1
Total	112	49	159

Table III. Testing Activities Summary

attributes and our interpretation of the results (marked as **Observation**). Each answer to a sub-research question is also summarised at the end. More detailed characterisations results of all the surveyed papers are presented in PeerJ version [90].

4.1. RQ1: How is the mutation testing used in testing activities?

4.1.1. RQ1.1 & RQ1.2: Which role does mutation testing play in each testing activity?

Observation. We opted to discuss the two research questions RQ1.1 and RQ1.2 together, because it gives us the opportunity to analyse per testing activity (e.g., test data generation) whether mutation testing is used as a way to guide the technique, or whether mutation testing is used as a technique to assess some (new) approach. Consider Table III, in which we report the role mutation testing plays onto the two columns "Assessment" and "Guide" (see Table II for the explanation about our attribute framework), while the testing activities are projected onto the rows. The table is then populated with our survey results, with the additional note that some papers belong to multiple categories.

As Table III shows, test data generation and test strategy evaluation occupies the majority of testing activities (accounting for 73.6%, 117 instances). The remaining are test case prioritisation (10.1%), test oracle generation/selection (8.2%) and test suite reduction/selection (8.2%). Only two instances studied test-case minimisation; this shows mutation testing has not been widely used to simplify test cases by shortening the test sequence and removing irrelevant statements.

As the two roles (assessment and guide) are used quite differently depending on the testing activities, we will discuss them separately. Also, for the "guiding" role, for which we see an increasing number of applications in recent decades, we find a number of hints and insights for future researchers to consider, which explains why we will analyse this part in a more detailed way when compared to the description of mutation testing as a tool for assessment.

(1) Assessment.

We observed that mutation testing mainly serves as an assessment tool to evaluate the fault-finding ability of the corresponding test techniques (70.4%) as it is widely considered as a "high end" test criterion [15]. In order to do so, mutation testing typically generates a large number of mutants of a

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

program, which are sometimes also combined with natural defects or hand-seeded ones. The results of the assessment are usually quantified as metrics of fault finding capability: mutation score (or mutation coverage, mutation adequacy) and killed mutants are the most common metrics in mutation testing. Besides, in test-case prioritisation, the Average Percentage of Faults Detected (APFD) [91], which measures the rate of fault detection per percentage of test suite execution, is also popular.

Amongst the papers in our set, we also found 18 studies that performed *mutant analysis*, which means that the researchers are trying to get a better understanding about mutation faults, e.g. which faults are more valuable in a particular context. A good example of this mutant analysis is the hard mutant problem investigated by Czemerinski et al. [92] where they analysed the failure rate for the hard-to-kill mutants (killed by less than 20% of test cases) using the domain partition approach.

(2) Guide.

To provide insight into how mutation testing acts as guidance to improve testing methods per test activity, we will highlight the most significant research efforts to demonstrate why mutation testing can be of benefit as a building block to guide other testing activities. In doing so, we hope the researchers in this field can learn from the current achievements so as to explore other interesting applications of mutation testing in the future.

Firstly, let us start with test data generation, which attracts most interest when mutation testing is adopted as a building block (28 instances). The main idea of mutation-based test data generation is to generate test data that can effectively kill mutants. For automatic test data generation, killing mutants serves as a condition to be satisfied by test data generation mechanisms, such as constraint-based techniques and search-based algorithms; in this way, mutation-based test data generation can be transformed into the structural testing problem. The mutation killable condition can be classified into three steps as suggested by Offutt and Untch [25]: reachability, necessity and sufficiency. When observing the experiments contained in the papers that we surveyed (except the model-based testing), we see that with regard to the killable mutant condition most papers (78.5%) are satisfied with a weak mutation condition (necessity), while a strong mutation condition (sufficiency) appears less (28.6%). The same is true when comparing first-order mutants (92.9%) to higher-order mutants (7.1%). Except for the entirely automatic test data generation, Baudry et al. [93–95] focused on the automation of the test case enhancement phase: they optimised the test cases regarding mutation score via genetic and bacteriological algorithms, starting from an initial test suite. von Mayrhauser et al. [96] and Simith and Williams [97] augmented test input data using the requirement of killing as many mutants as possible.

The second and third most-frequent use cases when applying mutation testing to guide the testing efforts come from test case prioritisation (6 instances) and the test strategy evaluation (6 instances). For test case prioritisation, the goal is to detect faults as early as possible in the regression testing process. The incorporation of measures of fault proneness into prioritisation techniques is one of the directions to overcome the limitation of the conventional coverage-based prioritisation methods. As relevant substitutes of real faults, mutants are used to approximate the fault-proneness capability to reschedule the testing sequences. Qu et al. [98] ordered test cases according to prior fault detection rate using both hand-seeded and mutation faults. Kwon et al. [99] proposed a linear regression model to reschedule test cases based on Information Retrieval and coverage information, where the coefficients in the model are determined by mutation testing. Moreover, Rothermel et al. [61,91] and Elbaum et al. [62] compared different approaches of test-case prioritisation, among which included

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

the prioritisation in order of the probability of exposing faults estimated by the killed mutants information. In Qi et al. [100]'s study, they adopted a similar test-case prioritisation method to improve patch validation during program repairing.

Thirdly, as for the fault localisation (4 instances), the locations of mutants are used to assist the localisation of "unknown" faults (the faults which have been detected by at least one test case, but that have still to be located [68]). The motivation of this approach is based on the following observation: "Mutants located on the same program statements frequently exhibit a similar behaviour" [68]. Thus, the identification of an "unknown" fault could be obtained thanks to a mutant at the same (or close) location. Taking advantage of the implicit link between the behaviour of "unknown" faults with some mutants, Murtaza et al. [101] used the traces of mutants and prior faults to train a decision tree to identify the faulty functions. Also, Papadakis et al. [68,69] and Moon et al. [102] ranked the suspiciousness of "faulty" statements based on their passing and failing test executions of the generated mutants.

When it comes to the test oracle problem (4 instances), mutation testing can also be of benefit for driving the generation of assertions, as the prerequisite for killing the mutant is to distinguish the mutant from the original program. In Fraser and Zeller [103]'s study, they illustrated how they used mutation testing to generate test oracles: assertions, as commonly used oracles, are generated based on the trace information of both the unchanged program and the mutants recorded during the executions. First, for each difference between the runs on the original program and its mutants, the corresponding assertion is added. After that, these assertions are minimised to find a sufficient subset to detect all the mutants per test case; this becomes a minimum set covering problem. Besides, Staats et al. [104] and Gay et al. [105] selected the most "effective" oracle data by ranking variables (trace data) based on their killed mutants information.

Mutation-based test-suite reduction (4 instances) relies on the number of killed mutants as a heuristic to perform test-suite reduction, instead of the more frequently used traditional coverage criteria, e.g., statement coverage. The intuition behind this idea is that the reduction based on the mutation faults can produce a better-reduced test suite with less or no loss in fault-detection capability. The notable examples include an empirical study carried out by Shi et al. [106] who compared the trade-offs among various test-suite reduction techniques based on statement coverage and killed mutants.

Zooming in on the test strategy evaluation (3 instances), we observe, on the one hand, the idea of incorporating an estimation of fault-exposure probability into test data adequacy criteria intrigued some researchers. Among them are Chen et al. [107]: in their influential work they examined the fault-exposing potential (FEP) coverage adequacy which is estimated by mutation analysis. Their findings show quite small, but statistically significant increases in the fault-detection ability of FEP-based test suites compared to statement-based ones. On the other hand, a mutation-based test strategy was also investigated and evaluated under the whole testing procedure including test case generation, augmentation and evaluation. A remarkable example includes an observational user study conducted by Smith and Williams [108] with four software testers to explore the cost-effectiveness of mutation testing for manually augmenting test cases. Their results indicate that mutation testing was regarded as an effective but relatively expensive technique for writing new test cases.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Evaluation Fault Type	Total
mutation faults	33
hand-seeded faults	7
hand-seeded + mutation faults	4
no evaluation	3
real faults	2

Table IV. Guide Role Summary

From the above guide roles of the testing activities, we can see that mutation testing is mainly used as an indication of the potential defects: either (1) to be killed in test data generation, test case prioritisation, test-suite reduction and test strategy evaluation, or (2) to be suspected in the fault localisation. In most cases, mutation testing serves as *where*-to-check constraints, i.e., introducing a modification in a certain statement or block. In contrast, only four studies applied mutation testing to solving the test oracle problem, which targets the *what*-to-check issue. The *what*-to-check problem is not a problem unique to mutation testing, but rather an inherent challenge of test data generation. As mentioned above, mutation testing can not only help in precisely targeting at *where* to check, but also suggesting *what* to check for [103] (see the first recommendation labeled as **R1** in Section 4.4). In this way, mutation testing could be of benefit to improve the test code quality.

After we had analysed how mutation testing is applied to guide various testing activities, we are now curious to better understand how these mutation-based testing methods were evaluated, especially because mutation testing is commonly used as an assessment tool. Therefore, we summed up the evaluation fault types among the articles labelled as "guide" in Table IV. We can see 37 cases (75.5%), which is the addition of the first and the third rows in Table IV (33 + 4), still adopted mutation faults to assess the effectiveness. Among these studies, four instances [104, 105, 109, 110] realised the potentially biased results caused by the same set of mutants being used in both guidance and assessment. They partitioned the mutants into different groups and used one for evaluation set. Besides, one study [55] used a different mutation tool while the other [85] adopted different mutation operators to generate mutants intending to eliminate bias. These findings signal an open issue: how to find an adequate fault set instead of mutation faults to effectively evaluate mutation-based testing methods? (see the second recommendation labeled as **R2** in Section 4.4) Although hand-seeded faults and real bugs could be an option, searching for such an adequate fault set increases the difficulty when applying mutation testing as guidance.

Summary. Test data generation and test strategy evaluation occupy the majority of testing activities when applying mutation testing (73.6%). Mutation testing mainly serves as an assessment tool to evaluate the fault-finding ability of various testing techniques (70.4%). While as guidance, mutation testing is primarily used in test data generation (28 instances) and test-case prioritisation (6 instances). From the above observations, we draw one open issue and one recommendation for the "guide" role of mutation testing. The open issue is how to find an adequate fault set instead of mutation faults to effectively evaluate mutation-based testing methods. The recommendation is mutation testing can suggest not only *where* to check but also *what* to check. *Where* to check widely used to generate killable mutant constraints in different testing activities; while *what* to check is seldom adopted to improve the test data quality.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Test Level	Total
n/a	73
unit	64
integration	9
other	9
system	8

Table V. Test Level Summary

4.1.2. RQ1.3: Which test level does mutation testing usually target at?

Observations. Table V presents the summary of the test level distribution across the articles. We observe that the authors of 73 papers do not provide a clear clue about the test level they target (the class marked as "n/a"). This is a clear and open invitation for future investigations in the area to be more clear about the essential elements of testing activities such as the test level. For the remainder of our analysis of RQ1.3, we excluded the papers labelled as "n/a" when calculating percentages, i.e., our working set is 86 $(159 - 73)$ papers.

Looking at Table V, mutation testing mainly targets the unit testing level (74.4%), an observation which is in accordance with the results in Jia and Harman's survey [1]. One of the underlying causes for the popularity of the unit level could be the origin of mutation testing. The principle of mutation testing is to introduce small syntactic changes to the original program; this means the mutation operators only focus on small parts of the program, such as arithmetical operators and *return* statements. Thus, such small changes mostly reflect the abnormal behaviour of unit-level functionality.

While unit testing is by far the most observed test level category in our set of papers, higher-level testing, such as integration testing, system testing, can also benefit from the application of mutation testing. Here we highlight several research works as examples: Hao et al. [111] and Do and Rothermel [64] used the programs with system test cases as their subjects in case studies. Hou et al. [85] studied interface-contract mutation in support of integration testing under the context of component-based software. Li et al. [112] proposed a two-tier testing method (one for integration level, the other for system level) for graphical user interface (GUI) software testing. Rutherford et al. [113] defined and evaluated adequacy criteria under system-level testing for distributed systems. In Denaro et al. [114]'s study, they proposed a test data generation approach using data flow information for inter-procedural testing of object-oriented programs.

The important point we discovered here is that all the aforementioned studies did not restrict mutation operators to model integration errors or system ones. In other words, the traditional program mutations can be applied to higher-level testing. Amongst these articles, the mutation operators adopted are mostly at the unit level, e.g. Arithmetic Mutation Replacement, Relational Mutation Replacement. The mutation operators designed for higher-level testing, e.g. [115, 116], are seldom used in these studies. This reveals a potential direction for future research: the cross-comparison of different levels of mutation operators and testing activities at different test levels (see the third recommendation labeled as **R3** in Section 4.4). The investigation of different level of mutations can explore the effectiveness of mutation faults at different test levels, such as the doubts whether integration-level mutation is better than unit-level mutation when assessing testing

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

techniques at the integration-level. In the same vein, an analysis of whether mutants are a good alternative to real/hand-seeded ones (proposed by Andrews et al. [24]) at higher levels of testing also seems like an important avenue to check out.

In addition, we created a class "others" in which we list 9 papers that we found hard to classify in any of the other four test phases. These works can be divided into three groups: grammar-based testing [117–119], spreadsheet-related testing [80,120,121] and SQL-related testing [122–124]. The application of mutation testing on the "other" set indicates that the definition of mutation testing is actually quite broad, thus potentially leading to even more intriguing possibilities [2]: What else can we mutate?

Summary. The application of mutation testing is mostly done at the unit-level testing (74.4%), while we did still observe several investigations targeting higher level testing efforts. What is worth noticing, is that those studies focusing on higher-level testing still apply traditional (unit-level) mutation operators instead of specific ones designed to model higher-level defects. This reveals a clear gap in current research: the application of mutation testing at higher test levels. Hence, we recommend future research to explore the relationship between different level mutations and different test levels. For example, empirical evaluations on whether mutations at the integration level can represent real faults at the integration testing level. In a different context, but still very important to highlight: 45.9% of papers did not clearly specify their target test level(s). For reasons of clarity, understandability and certainly replicability, it is very important to understand exactly at what level the testing activities take place. It is thus a clear call to arms to researchers to better describe these essential testing activity features.

4.1.3. RQ1.4: Which testing strategies does mutation testing support more frequently?

Observations. In Table VI we summarised the distribution of testing strategies based on our coarse-grained classification (e.g. structural testing, specification-based testing) as mentioned in Table II. Looking at Table VI, structure-based testing comes first among all the testing strategies (59.1%, 94 instances). The underlying cause could be that structural testing is still the main focus of testing strategies in the software testing context. The other testing strategies have also been supported by mutation testing: (1) specification-based testing accounts for 54 cases; (2) hybrid testing for five instances (combination of structural and structure-based testing); (3) three cases applying mutation testing in similarity-based testing; (4) 15 instances in others, e.g. static analysis.

One interesting finding is that the enhanced structural testing ranks second among the seven groups of testing strategies. The enhanced structural testing uses other approaches to overcome its inherent fault-revealing disadvantages (i.e. "coverage is not strongly correlated with test effectiveness" [58]), including mutation-based techniques, information retrieval knowledge, observation notations and assertion coverage. The popularity of enhanced structural testing reveals the awareness of the shortage of conventional coverage-based testing strategies has increased.

Compared to enhanced structural testing, enhanced specification-based testing did not attract much interest. The 11 instances mainly adopted mutation testing (e.g. Qu et al. [98] and Papadakis et al. [125]) to improve the testing strategies.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Testing Strategies	Total
structural testing	48
structural testing (enhanced)	46
specification-based testing	43
others	15
specification-based testing (enhanced)	11
hybrid testing	5
similarity-based testing	3

Table VI. Testing Strategies summary

Summary. Mutation testing has been widely applied in support of different testing strategies. From the observation, the testing strategies other than white box testing can also benefit from the application of mutation testing, such as specification-based testing, hybrid testing and similarity-based testing. But structural testing is more popular than the others (59.1%). Moreover, techniques like mutation-based techniques and information retrieval knowledge are also being adopted to improve the traditional structural-based testing, which typically only relies on the coverage information of software artefacts. This serves an indication of the increasing realisation of the limitations of coverage-based testing strategies.

4.2. R2: How are empirical studies related to the mutation testing designed and reported?

4.2.1. RQ2.1: Which mutation tools have been used more frequently?

Observations. We are interested in getting insight into the types (as defined in Table II) of mutation testing tools that are being used and into their availability. Therefore, we tabulated the different types of mutation testing tools and their availability in Table VII. As shown in Table VII, 49.1% of the studies adopted existing tools which are open to the public; this matches our expectation: as mutation testing is not the main focus of the studies, if there exists a well-performing and open-source mutation testing tool, the researchers are likely willing to use these tools. However, we also encountered 12 cases of using self-implemented tools and nine studies that manually applied mutation operators. A likely reason for implementing a new mutation testing tool or for applying mutation operators manually is that existing tools do not satisfy a particular need of the researchers. In addition, most existing mutation testing tool are typically targeting one specific language and a specific set of mutation operators [2] and they are not always easy to extend, e.g., when wanting to add a newly-defined mutation operator. As such, providing more flexible mechanisms for creating new mutation operators in mutation testing tools is an important potential direction for the future research [2] (see the fourth recommendation labeled as **R4** in Section 4.4).

Unfortunately, there are also still 77 studies (47%) that do not provide access to the tools, in particular, 40 papers did not provide any information about the tools at all, a situation that we marked as “n/a” in Table VII. This unclarity should serve as a notice to researchers: the mutation testing tool is one of the basic elements in mutation testing and lack of information on it seriously hinders replicability of the experiments.

Having discussed the tool availability and types, we are wondering which existing open-source mutation testing tools are most popular. The popularity of the tools can not only reveal their level of

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Availability	Types		Total
Yes	existing	78	84
	partially-based	7	
	self-written	1	
No	n/a (no information)	40	77
	existing (given the name/citation)	17	
	self-written	11	
	hand-seeded	9	

Table VII. Mutation Tool Summary

Language	**Tool**	**Total**
Java	MuJava/μ-java/Muclipse	31
	PIT/PiTest	7
	JAVALANCHE	7
	MAJOR	5
	Jumble	2
	Sofya	1
	Jester	1
C	Proteum	11
	MiLu	2
	SMT-C	1
Fortran	Mothra	4
SQL	SQLMutation/JDAMA	2
	SchemaAnalyst	1
C#	GenMutants	1
	PexMutator	1
JavaScript	MUTANDIS	2
AspectJ	AjMutator	1
UML specification	MoMuT::UML	1

Table VIII. Existing Mutation Tool Summary

maturity, but also provide a reference for researchers entering the field to help them choose a tool. To this end, we summarised the names of mutation tools for different programming languages in Table VIII. Table VIII shows that we encountered 18 mutation tools in total. Most tools target at one programming language (except for Mothra [17] which can support C as well). We encountered eight mutation tools for Java, with the top 3 most-used being MuJava [126], PIT [127]x and JAVALANCHE [87]. We found that three mutation tools for C are used, where Proteum [18] is most-frequently applied.

In Jia and Harman [1]'s literature review, they summarised 36 mutation tools developed between 1977 and 2009. When comparing their findings (36 tools) to ours (18 tools), we find that there are 12 tools in common. The potential reason for us not covering the other 24 is that we only consider peer-reviewed conference papers and journals; this will likely filter some papers which applied the other 24 mutation tools. Also important to stress, is that the goal of Jia and Harman's survey is different to

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Description Level	Total
well-defined	98
n/a	39
not sufficient	22

Table IX. Description Level of Mutation Operators Summary

ours: while we focus on the application of mutation tools, their study surveys articles that introduce mutation testing tools. In doing so, we still managed to discover 8 mutation tools which are not covered by Jia and Harman: (1) two tools are for Java: PIT and Sofya; (2) one for C: SMT-C; (3) one for SQL: SchemaAnalyst; (4) one for UML: MoMuT::UML; (5) two for C#: GenMutants and PexMutator; (6) one for JavaScript: MUTANDIS. Most of these tools were released after 2009, which makes them too new to be included in the review of Jia and Harman. Moreover, we can also witness the trend of the development of the mutation testing for programming languages other than Java and C means compared to Jia and Harman [1]'s data.

Summary. Around 50% of the articles that we have surveyed adopt existing (open source) tools, while in a few cases (21 in total) the authors implemented their own tools or seeded the mutants by hand. This calls for a more flexible mutation generation engine that allows to easily add mutation operators or certain types of constraints. Furthermore, we found 77 papers that did not provide any information about the mutation tools they used in their experiments; this should be a clear call to arms to the research community to be more precise when reporting on mutation testing experiments. We have also gained insight into the most popular tools for various programming languages, e.g., MuJava for Java and Proteum for C. We hope this list of tools can be a useful reference for new researchers who want to apply mutation analysis.

4.2.2. RQ2.2: Which mutation operators have been used more frequently?

Observations For the mutation operators, we first present the distribution of the three description levels (as mentioned in Table II) in Table IX. As Table IX shows, 61.6% (98 instances) of the studies that we surveyed specify the mutation operators that they use, while more than one-third of the articles do not provide enough information about the mutation operators to replicate the studies. These 61 instances that are labelled as "n/a" and "not sufficient" indicate the necessity of raising the awareness of the use of mutation in the testing experiment as claimed by Namin and Kakarla [128]. They highlighted that mutation testing is highly sensitive to external threats caused by the influential factors including mutation operators. Therefore, one suggestion here is to report the complete list of the mutation operators when applying mutation testing in testing experiments: the authors can provide either a self-contained list in the paper or the relevant citations which can be used to trace the complete list.

After that, based on our generic classification of the mutation operators (as defined in Section 3.3 (4)), we characterised the 98 papers labelled as "well-defined". In addition to the overall distribution of the mutation operators regardless of the programming languages, we are also interested in the differences of the mutation operators for different languages as the differences could indicate

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Level	Operator	Java	C	C++	C#	Fortran	SQL	JavaScript	Total
Specification Mutation		2	2	1	-	-	-	-	22
Program Mutation		43	9	3	2	2	4	1	77
Expression-level	arithmetic operator	42	8	3	2	2	2	-	66
	relational operator	38	7	3	2	2	2	-	62
	conditional operator	39	6	3	2	2	2	-	61
	assignment operator	24	3	2	-	-	-	-	29
	bitwise operator	26	3	2	-	-	-	-	31
	shift operator	26	1	2	-	-	-	-	29
	constant	12	3	2	1	2	2	-	27
	variable	8	3	2	1	2	2	1	23
	absolute value	5	3	1	2	1	2	-	15
	conditional expression	2	2	-	-	-	-	-	5
	parenthesis	1	1	-	-	-	-	-	2
	type	-	2	-	-	-	-	-	2
Statement-level	statement deletion	6	3	-	2	2	-	-	17
	method call	7	-	2	1	-	-	1	13
	`return` statement	6	2	-	-	2	-	-	11
	exception handler	1	-	1	1	-	-	-	5
	`goto` label	-	2	-	-	2	-	-	5
	control-flow disruption	2	1	2	-	-	-	-	4
	statement swap	2	1	2	-	-	-	-	4
	bomb statement	-	1	-	-	2	-	-	4
	`switch` statement	2	2	-	-	-	-	-	4
	`do` statement	-	1	-	-	2	-	-	3
	brace	-	2	-	-	-	-	-	2
	loop trap	-	2	-	-	-	-	-	2
	`while` statement	-	2	-	-	-	-	-	2
	`if` statement	-	-	-	-	-	-	-	-
Others	OO-specific	17	-	-	1	-	-	-	21
	Java-specific	12	-	1[‡‡]	-	-	-	-	12
	SQL-specific	-	-	-	-	-	4	-	4
	Concurrent mutation	3	-	-	-	-	-	-	3
	AOP-specific	-	-	-	-	-	-	-	2
	Interface mutation	1	-	-	-	-	-	-	2
	Spreadsheet-specific	-	-	-	-	-	-	-	2
	JavaScript-specific	-	-	-	-	-	-	1	1

Table X. Mutation Operators Used In the Literature

potential gaps in the existing mutation operator sets for certain programming languages. In Table X we project the different languages onto seven columns and our predefined mutation operator categories onto the rows, thus presenting the distribution of the mutation operators used in the literature under our research scope.

Overall, we can see that program mutation is more popular than specification mutation from the Table X. Among the program mutation operators, the arithmetic, relational and conditional operators are the top 3 mutation operators. These three operators often used together in most cases as their total numbers of applications are similar. The underlying cause of the popularity of these three operators could be that the three operators are among Offutt et al. [30]'s 5 sufficient mutation operators. Moreover, the expression-level operators are more popular than the statement-level ones. As for the

[‡‡]The Java-specific operator here refers to the `static` modifier change (including insertion and deletion).

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

statement-level mutation operators, statement deletion, method call and `return` statement are the top 3 mutation operators.

When we compare the mutation operators used in different languages to our mutation operator categories, we see that there exist differences between different programming languages, just like we assumed. Table XI leads to several interesting findings that reveal potential gaps in various languages:

1. for Java, seven mutation operators at the expression and statement level (except `go to` label which is not supported in Java) are not covered: type, bomb statement, `do` statement, brace, loop trap, `while` statement and `if` statement
2. for C, only one potential mutation operator, method call, was not covered. The C programming language does not provide direct support for exception handling
3. for C++, 3 expression-level, 10 statement-level and the OO-specific operators are not used in literature
4. for C#, only a basic set of mutation operators are covered
5. for Fortran, the earliest programming language mutation testing was applied to, the relevant studies covered a basic set
6. for SQL, since the syntax of SQL is quite different from the imperative programming languages, only six operators at the expression level and SQL-specific ones are used
7. for JavaScript, only three mutation operators other than JavaScript-specific ones are adopted in existing studies.

The above findings are just the initial results in which we neither did further analysis to chart the syntax differences of these languages nor investigate the possibility of the equivalent mutants caused by our classification. Moreover, for some languages, e.g. JavaScript, the relevant studies are too few to draw any definitive conclusions. Here, we can only say that for different languages the existing studies did not cover all the mutation operators that we listed in Table X: some are caused by the differences in the syntax, while the others could point to potential gaps. The distribution of the mutation operators of different languages summarised in Table X can be a reference for further investigations into mutation operator differences in various programming languages.

Furthermore, our generic classification of the existing mutation operators can be of benefit to compare mutation tools in the future. Thereby, we compared the existing mutation testing tools (as listed in Table VIII) to our mutation operator categories in Table XI. The result shows that none of the existing mutation testing tools we analysed can cover all types of operators we classified. For seven mutation testing tools for Java, they mainly focus on the expression-level mutations and only four kinds of statement-level mutators are covered. Furthermore, MuJava, PIT and Sofya provide some OO-specific operators, whereas PIT only supports one type, the Constructor Calls Mutator. For the three mutation testing tools for C that we have considered, Proteum covers the most mutation operators. SMT-C is an exceptional case of the traditional mutation testing which targets at semantic mutation testing. For the tools designed for C#, OO-specific operators are not present. Another interesting finding when we compared Table X and Table XI, is that the `if` statement mutator is not used in literature but it is supported by SMT-C. This observation indicates that not all the operators provided by the tools are used in the studies when applying mutation testing, which reinforces our message of the need for “well-defined” mutation operators when reporting mutation testing studies.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

	MuJava/μ-java/Muclipse	PIT/PiTest	JAVALANCHE	MAJOR	Jumble	Sofya	Jester	Proteum	MiLu	SMT-C	Mothra	SQLMutation/JDAMA	SchemaAnalyst	GenMutants	PexMutator	MUTANDIS	AjMutator	MoMuT::UML
Specification Mutation	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	√
arithmetic operator	√	√	√	√	√	√	-	√	√	-	√	√	-	√	√	√	-	-
relational operator	√	√	-	√	-	√	√	√	√	√	√	√	-	√	√	√	-	-
conditional operator	√	√	√	√	√	√	-	√	√	-	√	√	-	√	√	√	-	-
assignment operator	√	√	-	-	-	-	-	-	√	√	-	-	-	-	-	-	-	-
bitwise operator	√	√	-	-	√	-	-	√	√	-	-	-	-	-	-	-	-	-
shift operator	√	√	-	√	√	-	-	√	-	-	-	-	-	-	-	-	-	-
constant	-	√	√	-	√	-	√	√	√	√	√	√	-	-	-	√	-	-
variable	-	√	-	-	-	-	-	√	-	-	√	√	-	-	-	√	-	-
absolute value	-	-	-	√	-	-	-	-	-	-	√	√	-	√	√	-	-	-
conditional expression	-	√	-	-	-	-	-	√	-	-	-	-	-	-	-	-	-	-
parenthesis	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-	-	-
type	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	√	-	-
statement deletion	-	√	√	-	-	-	-	√	√	-	√	-	-	-	-	√	-	-
method call	-	-	-	-	-	√	-	-	-	√	-	-	-	-	-	√	-	-
return statement	-	√	-	-	√	-	-	√	-	-	√	-	-	-	-	√	-	-
if statement	-	-	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-
exception handler	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
goto label	-	-	-	-	-	-	-	√	-	-	√	-	-	-	-	-	-	-
control-flow disruption	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	√	-	-
statement swap	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	√	-	-
bomb statement	-	-	-	-	-	-	-	√	-	-	√	-	-	-	-	-	-	-
switch statement	-	√	-	-	√	-	-	√	-	√	-	-	-	-	-	√	-	-
do statement	-	-	-	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-
brace	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-	-	-
loop trap	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-	-	-
while statement	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-	-	-
OO-specific	√	√	-	-	-	√	-	-	-	-	-	-	-	-	-	-	-	-
Java-specific	√	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SQL-specific	-	-	-	-	-	-	-	-	-	-	-	√	√	-	-	-	-	-
JavaScript-specific	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	√	-	-
AOP-specific	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	√	-

Table XI. Comparison of Mutation Operators in Existing Mutation Tools

Summary. For the mutation operators, we focused on two attributes: their description level and a generic classification across tools and programming languages. When investigating the description level of the mutation operators that are used in studies, we found that 61.6% (98 instances) explicitly defined the mutation operators used. This leads us to strongly recommend improving the description level for the sake of replicability. Furthermore, the distribution of mutation operators based on our predefined categories shows the lacking of certain mutation operators in some programming languages among the existing (and surveyed) mutation testing tools. A possible avenue for future work is to see which of the *missing* mutation operators can be implemented for the programming languages lacking these operators.

4.2.3. RQ2.3: Which approaches are used to overcome the equivalent mutant problem more often when applying mutation testing?

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Equivalence Detector	Total
n/a	91
manual investigation	29
not killed as equivalent	16
no investigation	11
reduce likelihood	7
model checker	6
deterministic model	1
not killed as nonequivalent	1

Table XII. Methods for Overcoming E(quivalent) M(utant) P(roblem) Summary

Observations. In Table XII we summarised our findings of how the studies that we surveyed deal with the equivalent mutant problem. More specifically, Table XII presents how many times we encountered each of the approaches for solving the equivalent mutant problem. When looking at the results, we firstly observe that in 57.2% of the cases we assigned "n/a"; one possible reason for this high number of papers not providing information is that we excluded the internal methods of overcoming the equivalent mutant problem, meaning that mutation tools have it built in already. Moreover, as a non-decidable problem [13], the equivalent mutant problem is still under-developed [14].

As shown in Table XII, there are only 14 instances (6 "model checker" + 7 "reduce likelihood"+1 "deterministic model") actually adopting equivalent mutant detectors by using automatic mechanisms. In the remainder of the papers, the problem of equivalent mutants is solved by (1) manual analysis, (2) making assumptions (treating mutants not killed as either equivalent or nonequivalent), and (3) no investigation. In particular, the manual investigation (29 instances) and the method of treating mutants not killed as equivalent (16 instances) are more commonly used than other methods.

We can only speculate as to the reasons behind the above situation: Firstly, most studies use mutation testing as an evaluation mechanism or guiding heuristic, rather than their main research topic. So the authors are maybe not willing to spare too much effort in dealing with problems associated with mutation testing. Moreover, looking at the internal features of existing tools used in literature (in Table XIII), we found that only four tools adopt certain techniques to address the equivalent mutant problem. Most of the tools did not assist in dealing with the equivalent mutant problem. Thereby, the aforementioned three solutions, (1) manual analysis, (2) making assumptions or (3) no investigation, are considered by the authors. If there exists a well-developed auxiliary tool that can be seamlessly connected to the existing mutation systems for helping the authors detect equivalent mutants, this tool might be more than welcomed. We recommend that future research on the equivalent mutant problem can further implement their algorithms in such an auxiliary tool and make it open to the public (see the fifth recommendation labeled as **R5** in Section 4.4).

Secondly, mutation score is mainly used as a relative comparison for estimating the effectiveness of different techniques. Sometimes, mutation testing is only used to generate likely faults; equivalent mutants have no impact on the other measures such as the Average Percentage of Fault Detection rate (APFD) [91]. Furthermore, the definition of the mutation score is also modified by some authors: they used the total number of mutants as the denominator instead of number of

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Language	Tool	Equivalent Mutants	Cost Reduction
Java	MuJava/μ-java/Muclipse	n/a	MSG, bytecode translation (BCEL) [126]
	PIT/PiTest	n/a	Bytecode translation (ASM), coverage-based test selection [127]
	JAVALANCHE	Ranking mutations by impact [87]	MSG, bytecode translation (ASM), coverage-based test selection, parallel execution [87]
	MAJOR	n/a	Compiler-integrated, coverage-based test selection [129]
	Jumble	n/a	Bytecode translation (BCEL), conventional test selection [130]
	Sofya	n/a	Bytecode translation (BCEL) [131]
	Jester	n/a	n/a
C	Proteum	n/a	Mutant sample [18]
	MiLu	n/a	Higher-order mutants, test harness [132]
	SMT-C	n/a	Interpreter-based, weak mutation [133]
Fortran	Mothra	n/a	Interpreter-based [1]
SQL	SQLMutation/JDAMA	Constraint binding [20]	n/a
	SchemaAnalyst	n/a	n/a
C#	GenMutants	n/a	n/a
	PexMutator	n/a	Compiler-based [55]
JavaScript	MUTANDIS	Reduce likelihood [79]	Selection strategy [79]
AspectJ	AjMutator	Static analysis [134]	Compiler-based [134]
UML specification	MoMuT::UML	n/a	Compiler-based [135]

Note: “n/a” in the table means we did not find any relevant information recorded in literature or websites, and some tools might adopt certain techniques but did not report such information in the sources we can trace.

Table XIII. Inner features of Existing Mutation Tool

nonequivalent mutants. The equivalent mutant problem seems to not pose a significant threat to the validation of the testing techniques involved in these studies.

However, we should not underestimate the impact of the equivalent mutant problem on the accuracy of the mutation score. Previous empirical results indicated that 10 to 40 percent of mutants are equivalent [136, 137]. What's more, in Schuler and Zeller's study [47], they further claimed that around 45% of all undetected mutants turned out to be equivalent; this observation leads to the assumption that by simply treating mutants not killed as equivalent mutations, we could be overestimating the mutation score. Therefore, we recommend performing more large-scale investigations on whether the equivalent mutant problem has a strong impact on the accuracy of the mutation score.

Summary. The techniques for equivalent mutant detection are not commonly used when applying mutation testing. The main approaches that are used are the manual investigation and treating mutants not killed as equivalent. Based on the results, we recommend further research on the equivalent mutant problem can develop a mature and useful auxiliary tool which can easily link

to the existing mutation system. Such an extra tool assists people to solve the equivalent mutant problem when applying mutation testing more efficiently. Moreover, research on whether the equivalent mutant problem has a high impact on the accuracy of the mutation score is still needed, as the majority did not consider the equivalent mutant problem as a significant threat to validation of the testing activities. Also, 57.2% of the studies are lacking an explanation as to how they are dealing with overcoming the equivalent mutants problem; this again calls for more attention on reporting mutation testing *appropriately*.

4.2.4. RQ2.4: Which techniques are used to reduce the computational cost more frequently when applying mutation testing?

Observations. Since mutation testing requires high computational demands, cost reduction is necessary for applying mutation testing, especially in the industrial environment. We summed up the uses of such computational cost reduction techniques when using mutation testing in Table XIV. Please note that we excluded the runtime optimisation and selective mutation techniques. We opted to exclude this because the runtime optimisation is related to tool implementation which is not very likely to appear in the papers under our research scope, while the second one, selective mutation, is adopted by all the papers.

First of all, we noticed that 108 articles (67.9%) did not mention any reduction techniques. If we take into account those papers that used the runtime optimisation and selective mutation, one plausible explanation for the numerous “n/a” instances is a lack of awareness of *properly* reporting mutation testing, as we mentioned earlier. Secondly, random selection of the mutants based on a fixed number comes next (25 instances), followed by weak mutation (15 instances) and mutant sampling (9 cases).

But why is the technique of using a “fixed number” of mutants more popular than the others? We speculate that this could be because of the fact that choosing a certain number of mutants is more realistic in real software development: the total number of mutants generated by mutation tools is huge; while, realistically, only a few faults are made by the programmer during implementation. By fixing the number of mutants, it becomes easier to control the mutation testing process. Instead, relying on the weak mutation condition would require additional implementation efforts to modify the tools. Also of importance to note is the difference between the “fixed number” and “mutant sample” choice: while the first one implies a fixed number of mutants, the second one relies on fixed sampling rate. Compared to using a fixed number, mutant sampling sometimes cannot achieve the goal of reducing the mutants number efficiently. In particular, it is difficult to set one sample ratio if the size of the subjects varies greatly. For example, consider the following situation: one subject has 100,000 mutants while the other has 100 mutants. When the sample ratio is set to 1%, the first subject still has 1000 mutants left, while the number of mutants for the second one is reduced to one.

Moreover, we performed a further analysis of the mutation tools in Table XIII. We find that most tools adopted certain types of cost reduction techniques to overcome the high computational expense problem. For mutation testing tools for Java, bytecode translation is frequently adopted while Mutant Schemata Generation (MSG) is used in two tools, MuJava and JAVALANCHE. Another thing to highlight is that MiLu used a special test harness to reduce runtime [132]. This

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Cost Reduction Technique	Total
n/a	108
fixed number	25
weak mutation	15
mutant sample	9
selection strategy	4
higher-order	1

Table XIV. Cost Reduction Summary

test harness is created containing all the test cases and settings for running each mutant. Therefore, only the test harness need to be executed while each mutant runs as an internal function call during the testing process.

Selective mutation is also widely applied in nearly all the existing mutation testing tools (as shown in Table XI). This brings us to another issue, namely whether the selected subset of mutation operators is sufficient to represent the whole mutation operator set? When adopting selective mutation, some configurations are based on prior empirical evidence, e.g., Offutt et al. [30]'s five sufficient Fortran mutation operators and Siami et al. [32]'s 28 sufficient C mutation operators. However, most cases are not supported by the empirical or theoretical studies that show a certain subset of mutation operators can represent the whole mutation operator set. Thereby, we recommend more empirical studies on selective mutation in programming languages other than Mothra and C (see the sixth recommendation labeled as **R6** in Section 4.4).

Based on the above discussion, we infer that mutation testing's high computational problem can be adequately controlled using the state-of-art reduction techniques, especially selective mutation and runtime optimisation.

Summary. Selective mutation is the most widely used method for reducing the high computational cost of mutation testing. However, in most cases, there are no existing studies to support the prerequisite that selecting a particular subset of mutation operators is sufficient to represent the whole mutation operator set. Therefore, one recommendation is to conduct more empirical studies on selective mutation in various programming languages. Without regard to runtime optimisation and selective mutation, random selection of the mutants based on a fixed number (25 papers) is the most popular technique used to reduce the computational cost. The following ones are weak mutation and mutant sampling. Besides, a high percentage of the papers (67.9%) did not report any reduction techniques used to cope with computational cost when applying mutation testing; this again should serve as a reminder for our research community to pay more attention to *properly* reporting mutation testing in testing experiments.

4.2.5. RQ2.5: What are the most common subjects used in the experiments?

Observations. To analyse the most common subjects used in the experiments, we focus on three attributes of the subject programs, i.e., programming language, size and data availability. We will discuss these three attributes one by one in the following paragraphs.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Language	Total
Java	74
C	31
C#	6
Fortran	6
n/a	5
C++	4
Lustre/Simulink	8
SQL	4
Eiffel	3
Spreadsheet	3
AspectJ	3
C/C++	2
JavaScript	2
Ada	1
Kermeta	1
Delphi	1
Enterprise JavaBeans application	1
ISO C++ grammar	1
PLC	1
Sulu	1
XACML	1
XML	1
other specification languages	10

Table XV. Programming Language Summary

Table XV shows the distribution of the programming languages. We can see that Java and C dominate the application domain (66.0%, 105 instances). While JavaScript is an extensively used language in the web application domain, we only found two cases of it being described in the studies we surveyed. The potential reasons for this uneven distribution are unbalanced data availability and the complex nature of building a mutation testing system. The first cause, uneven data availability, is likely instigated through the fact that existing, well-defined software repositories such as `SIR` [88], `SF100` [89] are based on C and Java. We have not encountered such repositories for JavaScript, C# or SQL. Furthermore, it is easier to design a mutation system targeting one programming language. This stands in contrast to many web applications, that are often composed out of a combination of JavaScript, HTML, CSS, etc. This thus increases the difficulty of developing a mutation system for these combinations of programming languages. It is also worth noticing that we have not found any research on a purely functional programming language in our research scope.

When considering the maximum size of the subject programs, studies involving preliminary (<100 LOC), small (100∼10K LOC) subjects or studies with no information about programs size (“n/a” instances in Table XVI) occupy the 79% (126 instances) of papers in our collection. This high percentage of preliminary, small and “n/a” subjects indicates that mutation testing is rarely applied to programs whose size is higher than 10K LOC. We did find that only 31studies use medium size subjects, which corresponds to 19% of cases. Finally, we observed only two cases where mutation testing is applied in more large-scale projects: (i) Qi et al. [100] adopted mutation testing to prioritise test cases to speed up patch validation during program repairing and (ii) von Mayrhauser et al. [96]

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Subject Size	Total
n/a	59
small (100∼10K LOC)	58
medium (10K∼1M LOC)	31
preliminary (<100 LOC)	9
large (>1M LOC)	2

Table XVI. Subject Size Summary

Data Availability	Total
no	81
yes	78

Table XVII. Data Availability Summary

proposed a new test data generation technique based on an Artificial Intelligence (AI) planner and evaluated their method on an Automated Cartridge System (ACS). These two instances show the full potential of mutation testing to be employed as a practical testing tool for large industrial systems.

With regard to the concern of data availability, we observe the following: 49.1% of the studies provide open access to their experiments. Together with 5 instances of "n/a" in Table XV and 59 in Table XVI (including subjects which cannot be measured as LOC, e.g. SpreadSheet applications), it is worth noticing that subject programs used in the experiment should be clearly specified. In addition, basic information on programming language, size and subject should also be clearly specified in the articles to ensure replicability.

Summary. For the subject systems used in the experiments in our survey, we discussed three aspects: programming language, size of subject programs and data availability. For programming languages, Java and C are the most common programming languages used in the experiments when applying mutation testing. There is a clear challenge in creating more mutation testing tools for other programming languages, especially in the area of web applications and functional programming (see the seventh recommendation labeled as **R7** in Section 4.4).

As for the maximum size of subject programs, small to medium scale projects (100∼1M) are widely used when applying mutation testing. Together with two large-scale cases, we can see the full potential of mutation testing as a practical testing tool for large industrial systems. We recommend more research on large-scale systems to further explore scalability issues (see the eighth recommendation labeled as **R8** in Section 4.4).

The third aspect we consider is data availability. Only 49.1% of the studies that we surveyed provide access to the subjects used. This again calls for more attention on reporting test experiments *appropriately*: the authors should explicitly specify the subject programs used in the experiment, covering at least the details of programming language, size and source.

[‡‡]Here, we did not consider the number of "n/a" value when calculating the percentage ($159 - 59 = 100$).

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

4.3. Summary of Research Questions

We will now revisit the research questions and answer them in the light of our observations.

RQ1: How mutation testing is used in testing activities? Mutation testing is mainly used as a fault-based evaluation method (70.4%) in different testing activities. It assesses the fault-finding ability of various testing techniques through the mutation score or the number of killed mutants. Adopting mutation testing to improve other testing activities as a guide was first proposed by DeMilli and Offutt [35] in 1991 when they used it to generate test data. As a "high end" test criterion, mutation testing started to gain popularity as a building block in different testing activities, like test data generation (28 instances), test case prioritisation (6 cases) and test strategy evaluation (3 instances). However, using mutation testing as part of new test approaches raises a challenge in itself, namely how to efficiently evaluate mutation-based testing? Besides, we found one limitation related to the "guide" role of mutation testing: mutation testing usually serves as a *where*-to-check constraint rather than a *what*-to-check improvement. Another finding of the application of mutation testing is that it often targets unit-level testing (74.4%), while only a small amount of studies featuring higher-level testing show the benefit of mutation testing. As a result, we conclude that the current state of application of mutation testing is still rather limited.

RQ2: How are empirical studies related to the mutation testing designed and reported? First of all, for the mutation testing tools and mutation operators used in literature, we found that 49.1% of the articles adopted existing (open-source) mutation testing tools, such as MuJava for Java and Proteum for C. In contrast, we did encounter a few cases (21 in total) where the authors implemented their own tools or seeded mutants by hand. Furthermore, to investigate the distribution of mutation operators in the studies, we created a generic classification of mutation operators as shown in Section 3.3 (4). The results indicate that certain programming languages lack specific mutation operators, at least as far as the mutation tools that we have surveyed concern.

Moreover, when looking at the two most significant problems related to mutation testing, the main approaches to dealing with the equivalent mutant problem are (1) treating mutants not killed as equivalent and (2) not investigating the equivalent mutants at all. In terms of cost reduction techniques, we observed that the "fixed number of mutants" is the most popular technique used to reduce computational cost, although we should mention that we did not focus on built-in reduction techniques.

The above findings suggest that the existing techniques designed to support the application of mutation testing are largely still under development: a mutation testing tool with a more complete set of mutation operators or a flexible mutation generation engine to which mutation operators can be added, is still needed. In the same vein, a more mature and efficient auxiliary tool for assisting in overcoming the equivalent mutant problem is needed. Furthermore, we have observed that we lack insight into the impact of selective mutation on mutation testing; this suggests a deeper understanding of mutation testing is required. For example, if we know what particular kinds of faults mutation is good at finding or how useful a particular type of mutant is when testing real software, we can then design the mutation operators accordingly.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The poorly-specified aspects in reporting mutation testing	Number of papers
test level	73
mutation tool source	77
mutation operators	61
equivalent mutant problem	91
reduction problem	108
subject program source	81

Table XVIII. Poorly-specified aspects in empirical studies

Based on the distribution of subject programs used in testing experiments or case studies, Java and C are the most common programming languages used in the experiments. Also, small to medium scale projects (100∼1M LOC) are the most common subjects employed in the literature.

Besides, from the statistics of the collection, we found a considerable amount of papers did not provide a sufficiently clear or thorough specification when reporting mutation testing in their empirical studies. We summarised the poorly-specified aspects of mutation testing in Table XVIII. As a result, we call for more attention on reporting mutation testing appropriately. The authors should provide at least the following details in the articles: the mutation tool (preferably with a link to its source code), mutation operators used in experiments, how to deal with the equivalent mutant problem, how to cope with high computational cost and details on the subject program (see the ninth recommendation labeled as **R9** in Section 4.4).

4.4. Recommendation of Future Research

In this section, we will summarise the recommendations for the future research based on the discussion in two research questions above (in Section 4.1.1-4.3). We propose nine recommendations for the future research as follows:

- **R1: Mutation testing can not only be used as *where*-to-check constraints but also to suggest *what* to check to improve test code quality.**
  As shown in Table III in Section 4.1.1, when mutation testing serves as a "guide", mutants generated by the mutation testing system are mainly used to suggest the location to be checked, i.e., *where*-to-check constraints. For example, the location of mutants is used to assist the localisation of "unknown" faults in fault localisation. The mutation-based test data generation also used the position information to generate killable mutant conditions. However, mutation testing is not widely considered to be a benefit to improve test code quality by suggesting *what* to check, especially in test oracle problem. The *what*-to-check direction can be one opportunity for future research in mutation testing as a "guide" role.
- **R2: For testing approaches that are guided by a mutation approach, more focus can be given to finding an appropriate way to evaluate mutation-based testing in an efficient manner.**
  When looking at the evaluation types in Table IV in Section 4.1.1, we observe that 75.5% of the mutation-based testing techniques still adopt mutation faults to assess their effectiveness.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

This raises the question of whether the conclusions might be biased. As such, we open the issue of finding an appropriate way to evaluate mutation-based testing in an efficient manner.

- **R3: Study the higher-level application of mutation testing.**
  In Section 4.1.2 we made the observation that mutation testing seems to mainly target the unit-level testing, accounting for 74.4% of the studies we surveyed. This reveals a potential gap in how mutation testing is currently applied. It is thus our recommendation that researchers pay more attention to higher-level testing, such as integration testing and system testing. The research community should not only investigate potential differences in applying mutation testing at the unit-level or at a higher level of testing, but also explore whether the conclusions based on unit-level mutation testing could still apply to higher-level mutation testing. A pertinent question in this area could for example be whether an integration mutation fault can be considered as an alternative to a real bug at the integration level.
- **R4: The design of a more flexible mutation generation engine that allows for the easy addition of new mutation operators.**
  As shown in Table VII in Section 4.2.1, 49.1% of the articles adopted the existing tools which are open-source, while we also found 21 instances of researchers implementing their own tool or seeding the mutants by hand. Furthermore, in Table X and Table XI, we can see certain existing mutation testing tools lack certain mutation operators. These findings imply that existing mutation testing tools can not always satisfy all kinds of needs, and new types of mutation operators are also potentially needed. Since most existing mutation testing tools have been initialized for one particular language and a specific set of mutation operators, we see a clear need for a more flexible mutation generation engine to which new mutation operators can be added easily [2].
- **R5: A mature and efficient auxiliary tool to detect equivalent mutants that can be easily integrated with existing mutation tools.**
  In Section 4.2.3, the problem of equivalent mutants is mainly solved by manual analysis, assumptions (treating mutants not killed as either equivalent or nonequivalent) or no investigation at all during application. This observation leads to doubt about the efficacy of the state-of-art equivalent mutant detection. In the meanwhile, if there is a mature and efficient auxiliary tool which can easily link to the existing mutation system, the auxiliary tool can be a practical solution for the equivalent mutant problem when applying mutation testing. As a result, we call for a well-developed and easy to integrate auxiliary tool for the equivalent mutant problem.
- **R6: More empirical studies on the selective mutation method can pay attention to programming languages other than Mothra and C.**
  As mentioned in Section 4.2.4, selective mutation is used by all the studies in our research scope. However, the selection of a subset of mutation operators in most papers is not well supported by existing empirical studies, except for Mothra [30] and C [32]. Selective mutation requires more empirical studies to explore whether a certain subset of mutation operator can be applied in different programming languages.
- **R7: More attention should be given to other programming languages, especially web applications and functional programming projects.**

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

As discussed in RQ2.5 in Section 4.2.5, Java and C are the most common programming languages that we surveyed. While JavaScript and functional programming languages are seldom applied. JavaScript, as one of the most popular languages in developing web applications, calls for more attention from the researchers. In the meanwhile, functional programming languages, such as Lisp and Haskell, are still playing an inevitable role in the implementation of programs; thus also require more focus in future studies.

- **R8: Application of mutation testing in large-scale systems to explore scalability issues.** From Table XVI in Section 4.2.5 we learn that the application of mutation testing to large-scale programs whose size is greater than 1M LOC rarely happens (only two cases). In order to effectively apply mutation testing in industry, the scalability issue of mutation testing requires more attention. We recommend future research to use mutation testing in more large-scale systems to explore scalability issues.
- **R9: Authors should provide at least the following details in the articles: mutation tool source, mutation operators used in experiments, how to deal with the equivalent mutant problem, how to cope with high computational cost and subject program source.** From Table XVIII in Section 4.4 we remember that a considerable amount of papers reported mutation testing in a poor manner. To help progress research in the area more quickly and to allow for replication studies, we all need to take care to be careful in how we report mutation testing in empirical studies. We consider the above five elements to be essential when reporting mutation testing.

5. THREATS TO THE VALIDITY OF THIS REVIEW

We have presented our methodology for performing this systematic literature review and its findings in the previous sections. As conducting a literature review largely relies on manual work, there is the concern that different researchers might end up with slightly different results and conclusions. To eliminate this potential risk caused by researcher bias as much as possible, we follow the guidelines for performing systematic literature reviews by Kitchenman [6], Wohlin [73], Brereton et al. [71]) whenever possible. In particular, we keep a detailed record of procedures made throughout the review process by documenting all the metadata from article selection to characterisation (see PeerJ version) .

In this section, we are going to describe the main threats to validity of this review and discuss how we attempted to mitigate the risks regarding four aspects: the article selection, the attribute framework, the article characterisation and the result interpretation.

5.1. Article Selection

Mutation testing is an active research field, and a plethora of realisations have been achieved as shown in Jia and Harman's thorough survey [1]. To address the main interest of our review, i.e. actual application of mutation testing, we need to define inclusion/exclusion criteria to include papers of interest and exclude irrelevant ones. But this also introduces a potential threat to the validity of our study: unclear article selection criteria. To minimise the ambiguity caused by the selection strategies, we carried out a pilot run of the study selection process to validate our selection criteria among the

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

three authors. This selection criteria validation led to a tiny revision. Besides, if there is any doubt about whether a paper belongs in our selected set, we had an internal discussion to see whether the paper should be included or not.

The venues in Table I were selected because we considered them to be the key venues in software engineering and most relevant to software testing, debugging, software quality and validation. This presumption might result in an incomplete paper collection. In order to mitigate this threat, we also adopted snowballing to extend our set of papers from pre-selected venues to reduce the possibility of missing influential papers.

Although we made efforts to minimise the risks with regard to article selection, we cannot make definitive claims about the completeness of this review. We have one major limitation related to the article selection: we only considered top conference or journal papers to ensure the high quality while we excluded article summaries, interviews, reviews, workshops, panels and poster sessions. Vice versa, sometimes we were also confronted with a vague use of the "mutation testing" terminology, in particular, some papers used the term "mutation testing", while they are doing fault seeding, e.g. Lyu et al. [138]. The main difference between mutation testing and error seeding is the way how to introduce defects in the program [139]: mutation testing follows certain rules while error seeding adds the faults directly without any particular techniques.

5.2. *Attribute Framework*

We consider the attribute framework to be the most subjective step in our approach: the generalisation of the attribute framework could be influenced by the researcher's experience as well as the reading sequence of the papers. To generate a useful and reasonable attribute framework, we followed a two-step approach: (1) we first wrote down the facets of interest according to our research questions and then (2) derived corresponding attributes of interest. Moreover, for each attribute, we need to ensure all possible values of each attribute are available, as well as a precise definition of each value. In this manner, we can target and modify the unclear points in our framework quickly. In particular, we conducted a pilot run for specifically for validating our attribute framework. The results led to several improvements to the attribute framework and demonstrated the applicability of the framework.

5.3. *Article Characterisation*

Thanks to the complete definitions of values for each attribute, we can assign the value(s) to articles in a systematic manner. However, applying the attribute framework to the research body is still a subjective process. To eliminate subtle differences caused by our interpretation, we do no further interpretation of the information extracted from the papers in the second pilot run of validation. In particular, if there is no specified detail in a paper, we then mark as "n/a". Furthermore, we listed our data extraction strategies about how to identify and classify the values of each attribute in Section 3.3.

5.4. *Result Interpretation*

Researcher bias could cause a potential threat to validity when it comes to the result interpretation, i.e., the author might seek what he expected for in the review. We reduce the bias by (1) selecting

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

all possible papers in a manner that is fair and seen to be fair and (2) discuss our findings based on statistical data we collected from the article characterisation. Also, our results are discussed among all the authors to reach an agreement.

6. CONCLUSION

In this paper we have reported on a systematic literature review on the *application perspective* of mutation testing, clearly contrasting previous systematic reviews that surveyed the whole field of mutation testing and that did not specifically go into how mutation testing is applied (e.g. [1]). We have characterised the studies that we have found on the basis of seven facets: (1) the role that mutation testing has in testing activities; (2) the testing activities (including categories, test level and testing strategies); (3) the mutation tools used in the experiments; (4) the mutation operators used in the experiments; (5) the description of the equivalent mutant problem; (6) the description of cost reduction techniques for mutation testing; and (7) the subjects software systems involved in the experiments (in terms of programming language, size and data availability). These seven facets pertain to our two main research questions: **RQ1** *How is mutation testing used in testing activities?* and **RQ2** *How are empirical studies related to mutation testing designed and reported?*

Our main procedure to conduct this systematic literature review is shown in Figure 1. To collect all the relevant papers under our research scope, we started with search queries in online libraries considering 17 venues. We selected the literature that focuses on the supporting role of mutation testing in testing activities with sufficient evidence to suggest that mutation testing is used. After that, we performed a snowballing procedure to collect missing articles, thus resulting in a final selection of 159 papers in 21 venues. Through a detailed reading of this research body, we derived an attribute framework that was consequently used to characterise the studies in a structured manner. The resulting systematic literature review can be of benefit for researchers in the area of mutation testing. Specifically, we provide (1) guidelines on how to apply and subsequently report on mutation testing in testing experiments and (2) recommendations for future work.

The derived attribute framework is shown in Table II. This attribute framework generalises and details the essential elements related to the *actual* application of mutation testing, such as in which circumstances mutation testing is used and which mutation testing tool is selected. In particular, a generic classification of mutation operators is constructed to study and compare the mutation operators used in the experiments described. This attribute framework can be used as a reference for researchers when describing mutation operators. Based on our analysis of the results (in Section 4), four points are key to remember:

1. Most studies use mutation testing as an assessment tool; they target the unit level. Not only should we pay more attention to higher-level and specification mutation, but we should also study how mutation testing can be employed to improve the test code quality. Furthermore, we also encourage researchers to investigate and explore more interesting applications for mutation analysis in the future by asking such questions as: what else can we mutate? (Section 4.1.1-4.1.2)
2. Many of the supporting techniques for making mutation testing truly applicable are still under-developed. Also, existing mutation tools are not complete with regard to the mutation

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

operators they offer. The two key problems, namely the equivalent mutant detection problem and the high computation cost of mutation testing issues, are not well-solved in the context of our research body (Section 4.2.1-4.2.4).

3. A deeper understanding of mutation testing is required, such as what particular kinds of faults mutation testing is good at finding. This would help the community to develop new mutation operators as well as overcome some of the inherent challenges (Section 4.3).
4. The awareness of *appropriately* reporting mutation testing in testing experiments should be raised among the researchers (Section 4.3).

In summary, the work described in this paper makes following contributions:

1. A systematic literature review of 159 studies that apply mutation testing in scientific experiments, which includes an in-depth analysis of how mutation testing is applied and reported on.
2. A detailed attribute framework that generalises and details the essential elements related to *actual* use of mutation testing
3. A generic classification of mutation operators that can be used to compare different mutation testing tools.
4. An actual characterisation of all the selected papers based on the attribute framework.
5. A series of recommendations for future work including important suggestions on how to report mutation testing in testing experiments in an appropriate manner.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

REFERENCES

1. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on* 2011; **37**(5):649–678.
2. Offutt J. A mutation carol: Past, present and future. *Information and Software Technology* 2011; **53**(10):1098–1107.
3. Lipton R. Fault diagnosis of computer programs. *Student Report, Carnegie Mellon University* 1971; .
4. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. *Computer* 1978; (4):34–41.
5. Hamlet RG. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on* 1977; (4):279–290.
6. Kitchenham B. Guidelines for performing systematic literature reviews in software engineering. *Technical Report EBSE-2007-01*, 2007.
7. Offutt A. The coupling effect: fact or fiction. *ACM SIGSOFT Software Engineering Notes*, vol. 14, ACM, 1989; 131–140.
8. Offutt AJ. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1992; **1**(1):5–20.
9. Wah K. Fault coupling in finite bijective functions. *Software Testing, Verification and Reliability* 1995; **5**(1):3–47.
10. Wah K. A theoretical study of fault coupling. *Software testing, verification and reliability* 2000; **10**(1):3–45.
11. Wah KHT. An analysis of the coupling effect i: single test data. *Science of Computer Programming* 2003; **48**(2):119–161.
12. Kapoor K. Formal analysis of coupling hypothesis for logical faults. *Innovations in Systems and Software Engineering* 2006; **2**(2):80–87.
13. Budd TA, Angluin D. Two notions of correctness and their relation to testing. *Acta Informatica* 1982; **18**(1):31–45.
14. Madeyski L, Orzeszyna W, Torkar R, Józala M. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *Software Engineering, IEEE Transactions on* 2014; **40**(1):23–42.
15. Ammann P, Offutt J. *Introduction to software testing*. Cambridge University Press, 2008.
16. Budd T, Sayward F. Users guide to the pilot mutation system. *Yale University, New Haven, Connecticut, Technique Report* 1977; **114**.
17. King KN, Offutt AJ. A fortran language system for mutation-based software testing. *Software: Practice and Experience* 1991; **21**(7):685–718.
18. Delamaro ME, Maldonado JC, Mathur A. Proteum-a tool for the assessment of test adequacy for c programs users guide. *PCS*, vol. 96, 1996; 79–95.
19. Ma YS, Offutt J, Kwon YR. Mujava: a mutation system for java. *Proceedings of the 28th international conference on Software engineering*, ACM, 2006; 827–830.
20. Tuya J, Suárez-Cabal MJ, De La Riva C. Mutating database queries. *Information and Software Technology* 2007; **49**(4):398–417.
21. Mathur AP, Wong WE. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability* 1994; **4**(1):9–31.
22. Frankl PG, Weiss SN, Hu C. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software* 1997; **38**(3):235–253.
23. Li N, Praphamontripong U, Offutt J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, IEEE, 2009; 220–229.
24. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments?[software testing]. *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, IEEE, 2005; 402–411.
25. Offutt AJ, Untch RH. Mutation 2000: Uniting the orthogonal. *Mutation testing for the new century*. Springer, 2001; 34–44.
26. Acree Jr AT. On mutation. *Technical Report*, DTIC Document 1980.
27. Wong WE, Mathur AP. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software* 1995; **31**(3):185–196.
28. Hussain S. Mutation clustering. *Ms. Th., King's College London, Strand, London* 2008; .
29. Ji C, Chen Z, Xu B, Zhao Z. A novel method of mutation clustering based on domain analysis. *SEKE*, vol. 9, 2009; 422–425.
30. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1996; **5**(2):99–118.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

31. Mresa ES, Bottaci L. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing Verification and Reliability* 1999; **9**(4):205–232.
32. Siami Namin A, Andrews JH, Murdoch DJ. Sufficient mutation operators for measuring test effectiveness. *Proceedings of the 30th international conference on Software engineering*, ACM, 2008; 351–360.
33. Howden WE. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering* 1982; (4):371–379.
34. Offutt AJ, Lee SD. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering* 1994; **20**(5):337–344.
35. DeMillo R, Offutt AJ. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on* 1991; **17**(9):900–910.
36. Untch RH. Mutation-based software testing using program schemata. *Proceedings of the 30th annual Southeast regional conference*, ACM, 1992; 285–291.
37. Untch R, Offutt AJ, Harrold MJ. Mutation testing using mutant schemata. *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 1993; 139–148.
38. Baldwin D, Sayward F. Heuristics for determining equivalence of program mutations. *Technical Report*, DTIC Document 1979.
39. Hierons R, Harman M, Danicic S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability* 1999; **9**(4):233–262.
40. Martin E, Xie T. A fault model and mutation testing of access control policies. *Proceedings of the 16th international conference on World Wide Web*, ACM, 2007; 667–676.
41. Ellims M, Ince D, Petre M. The csaw c mutation tool: Initial results. *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, IEEE, 2007; 185–192.
42. du Bousquet L, Delaunay M. Towards mutation analysis for lustre programs. *Electronic Notes in Theoretical Computer Science* 2008; **203**(4):35–48.
43. Harman M, Hierons R, Danicic S. The relationship between program dependence and mutation analysis. *Mutation testing for the new century*. Springer, 2001; 5–13.
44. Adamopoulos K, Harman M, Hierons RM. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. *Genetic and evolutionary computation conference*, Springer, 2004; 1338–1349.
45. Vincenzi AMR, Nakagawa EY, Maldonado JC, Delamaro ME, Romero RAF. Bayesian-learning based guidelines to determine equivalent mutants. *International Journal of Software Engineering and Knowledge Engineering* 2002; **12**(06):675–689.
46. Schuler D, Dallmeier V, Zeller A. Efficient mutation testing by checking invariant violations. *Proceedings of the eighteenth international symposium on Software testing and analysis*, ACM, 2009; 69–80.
47. Schuler D, Zeller A. (un-) covering equivalent mutants. *2010 Third International Conference on Software Testing, Verification and Validation*, IEEE, 2010; 45–54.
48. Budd TA, Lipton RJ, DeMillo RA, Sayward FG. *Mutation analysis*. Yale University, Department of Computer Science, 1979.
49. Agrawal H, DeMillo R, Hathaway R, Hsu W, Hsu W, Krauser E, Martin RJ, Mathur A, Spafford E. Design of mutant operators for the c programming language. *Technical Report*, Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana 1989.
50. DeMillo RA. Test adequacy and program mutation. *Software Engineering, 1989. 11th International Conference on*, 1989; 355–356, doi:10.1109/ICSE.1989.714449.
51. Ntafos SC. On testing with required elements. *Proceedings of COMPSAC*, vol. 81, 1981; 132–139.
52. Ntafos SC. An evaluation of required element testing strategies. *Proceedings of the 7th international conference on Software engineering*, IEEE Press, 1984; 250–256.
53. Ntafos SC. On required element testing. *Software Engineering, IEEE Transactions on* 1984; (6):795–803.
54. Duran JW, Ntafos SC. An evaluation of random testing. *Software Engineering, IEEE Transactions on* 1984; (4):438–444.
55. Zhang L, Xie T, Zhang L, Tillmann N, De Halleux J, Mei H. Test generation via dynamic symbolic execution for mutation testing. *Software Maintenance (ICSM), 2010 IEEE International Conference on*, IEEE, 2010; 1–10.
56. Papadakis M, Malevris N. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Journal* 2011; **19**(4):691–723.
57. Namin AS, Andrews JH. The influence of size and coverage on test suite effectiveness. *Proceedings of the eighteenth international symposium on Software testing and analysis*, ACM, 2009; 57–68.
58. Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014; 435–445.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

59. Zhang Y, Mesbah A. Assertions are strongly correlated with test suite effectiveness. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, 2015; 214–224.
60. Whalen M, Gay G, You D, Heimdahl MP, Staats M. Observable modified condition/decision coverage. *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, 2013; 102–111.
61. Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on* 2001; **27**(10):929–948.
62. Elbaum S, Malishevsky AG, Rothermel G. Test case prioritization: A family of empirical studies. *Software Engineering, IEEE Transactions on* 2002; **28**(2):159–182.
63. Do H, Rothermel G. A controlled experiment assessing test case prioritization techniques via mutation faults. *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, IEEE, 2005; 411–420.
64. Do H, Rothermel G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *Software Engineering, IEEE Transactions on* 2006; **32**(9):733–752.
65. Offutt AJ, Pan J, Voas JM. Procedures for reducing the size of coverage-based test sets. *Proceedings of the Twelfth International Conference on Testing Computer Software*, Citeseer, 1995.
66. Zhang L, Marinov D, Zhang L, Khurshid S. An empirical study of junit test-suite reduction. *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, IEEE, 2011; 170–179.
67. Wang X, Cheung SC, Chan WK, Zhang Z. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. *Proceedings of the 31st International Conference on Software Engineering*, IEEE Computer Society, 2009; 45–55.
68. Papadakis M, Le Traon Y. Using mutants to locate "unknown" faults. *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, IEEE, 2012; 691–700.
69. Papadakis M, Le Traon Y. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability* 2015; **25**(5-7):605–628.
70. Kysh L. Difference between a systematic review and a literature review. https://dx.doi.org/10.6084/m9.figshare.766364.v1 2013. [Online; accessed 4-August-2016].
71. Brereton P, Kitchenham BA, Budgen D, Turner M, Khalil M. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software* 2007; **80**(4):571–583.
72. Cornelissen B, Zaidman A, Van Deursen A, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 2009; **35**(5):684–702.
73. Wohlin C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ACM, 2014; 38.
74. Graham D, Van Veenendaal E, Evans I. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.
75. van Deursen A. Software Testing in 2048. https://speakerdeck.com/avandeursen/software-testing-in-2048 1 2016. [Online; accessed 13-Sep-2016].
76. Papadakis M, Malevris N. Automatic mutation test case generation via dynamic symbolic execution. *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, IEEE, 2010; 121–130.
77. Available mutation operations (PIT). http://pitest.org/quickstart/mutators/. [Online; accessed 10-August-2016].
78. Ma YS, Offutt J. Description of class mutation mutation operators for java. *Electronics and Telecommunications Research Institute, Korea* 2005; .
79. Mirshokraie S, Mesbah A, Pattabiraman K. Efficient javascript mutation testing. *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, IEEE, 2013; 74–83.
80. Außerlechner S, Fruhmann S, Wieser W, Hofer B, Spörk R, Mühlbacher C, Wotawa F. The right choice matters! smt solving substantially improves model-based debugging of spreadsheets. *2013 13th International Conference on Quality Software*, IEEE, 2013; 139–148.
81. Delamare R, Baudry B, Ghosh S, Gupta S, Le Traon Y. An approach for testing pointcut descriptors in aspectj. *Software Testing, Verification and Reliability* 2011; **21**(3):215–239.
82. Xu D, Ding J. Prioritizing state-based aspect tests. *2010 Third International Conference on Software Testing, Verification and Validation*, IEEE, 2010; 265–274.
83. Hong S, Staats M, Ahn J, Kim M, Rothermel G. The impact of concurrent coverage metrics on testing effectiveness. *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, IEEE, 2013; 232–241.
84. Hong S, Staats M, Ahn J, Kim M, Rothermel G. Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation. *Software Testing, Verification and Reliability* 2015; **25**(4):334–370.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

85. Hou SS, Zhang L, Xie T, Mei H, Sun JS. Applying interface-contract mutation in regression testing of component-based software. *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, IEEE, 2007; 174–183.
86. Yoon H, Choi B. Effective test case selection for component customization and its application to enterprise javabeans. *Software Testing, Verification and Reliability* 2004; **14**(1):45–70.
87. Schuler D, Zeller A. Javalanche: efficient mutation testing for java. *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ACM, 2009; 297–298.
88. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 2005; **10**(4):405–435.
89. Fraser G, Arcuri A. Sound empirical evidence in software testing. *Proceedings of the 34th International Conference on Software Engineering*, IEEE Press, 2012; 178–188.
90. Zhu Q, Annibale P, Zaidman A. A systematic literature review of how mutation testing supports test activities. *PeerJ Preprints* 2016; doi:10.7287/peerj.preprints.2483v1. URL https://doi.org/10.7287/peerj.preprints.2483v1.
91. Rothermel G, Untch RH, Chu C, Harrold MJ. Test case prioritization: An empirical study. *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, IEEE, 1999; 179–188.
92. Czemerinski H, Braberman V, Uchitel S. Behaviour abstraction adequacy criteria for api call protocol testing. *Software Testing, Verification and Reliability* 2015; .
93. Baudry B, Le Hanh V, Jézéquel JM, Le Traon Y. Building trust into oo components using a genetic analogy. *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, IEEE, 2000; 4–14.
94. Baudry B, Fleurey F, Jézéquel JM, Le Traon Y. Genes and bacteria for automatic test cases optimization in the. net environment. *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, IEEE, 2002; 195–206.
95. Baudry B, Fleurey F, Jézéquel JM, Le Traon Y. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability* 2005; **15**(2):73–96.
96. von Mayrhauser A, Scheetz M, Dahlman E, Howe AE. Planner based error recovery testing. *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, IEEE, 2000; 186–195.
97. Smith BH, Williams L. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering* 2009; **14**(3):341–369.
98. Qu X, Cohen MB, Rothermel G. Configuration-aware regression testing: an empirical study of sampling and prioritization. *Proceedings of the 2008 international symposium on Software testing and analysis*, ACM, 2008; 75–86.
99. Kwon JH, Ko IY, Rothermel G, Staats M. Test case prioritization based on information retrieval concepts. *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, vol. 1, IEEE, 2014; 19–26.
100. Qi Y, Mao X, Lei Y. Efficient automated program repair through fault-recorded testing prioritization. *2013 IEEE International Conference on Software Maintenance*, IEEE, 2013; 180–189.
101. Murtaza SS, Madhavji N, Gittens M, Li Z. Diagnosing new faults using mutants and prior faults (nier track). *Software Engineering (ICSE), 2011 33rd International Conference on*, IEEE, 2011; 960–963.
102. Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: Mutating faulty programs for fault localization. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, IEEE, 2014; 153–162.
103. Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on* 2012; **38**(2):278–292.
104. Staats M, Gay G, Heimdahl MP. Automated oracle creation support, or: how i learned to stop worrying about fault propagation and love mutation testing. *Proceedings of the 34th International Conference on Software Engineering*, IEEE Press, 2012; 870–880.
105. Gay G, Staats M, Whalen M, Heimdahl MP. Automated oracle data selection support. *IEEE Transactions on Software Engineering* 2015; **41**(11):1119–1137.
106. Shi A, Gyori A, Gligoric M, Zaytsev A, Marinov D. Balancing trade-offs in test-suite reduction. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014; 246–256.
107. Chen W, Untch RH, Rothermel G, Elbaum S, Von Ronne J. Can fault-exposure-potential estimates improve the fault detection abilities of test suites? *Software Testing, Verification and Reliability* 2002; **12**(4):197–218.
108. Smith BH, Williams L. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software* 2009; **82**(11):1819–1832.
109. Lou Y, Hao D, Zhang L. Mutation-based test-case prioritization in software evolution. *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, IEEE, 2015; 46–57.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

110. Hao D, Zhang L, Wu X, Mei H, Rothermel G. On-demand test suite reduction. *Proceedings of the 34th International Conference on Software Engineering*, IEEE Press, 2012; 738–748.
111. Hao D, Zhang L, Zhang L, Rothermel G, Mei H. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2014; **24**(2):10.
112. Li P, Huynh T, Reformat M, Miller J. A practical approach to testing gui systems. *Empirical Software Engineering* 2007; **12**(4):331–357.
113. Rutherford MJ, Carzaniga A, Wolf AL. Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. *Software Engineering, IEEE Transactions on* 2008; **34**(4):452–470.
114. Denaro G, Margara A, Pezze M, Vivanti M. Dynamic data flow testing of object oriented systems. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, 2015; 947–958.
115. Delamaro ME, Maidonado J, Mathur AP. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering* 2001; **27**(3):228–247.
116. Mateo PR, Usaola MP, Offutt J. Mutation at the multi-class and system levels. *Science of Computer Programming* 2013; **78**(4):364–387.
117. Hennessy M, Power JF. Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software. *Empirical Software Engineering* 2008; **13**(4):343–368.
118. Chae HS, Woo G, Kim TY, Bae JH, Kim WY. An automated approach to reducing test suites for testing retargeted c compilers for embedded systems. *Journal of Systems and Software* 2011; **84**(12):2053–2064.
119. Belli F, Beyazıt M. Exploiting model morphology for event-based testing. *IEEE Transactions on Software Engineering* 2015; **41**(2):113–134.
120. Hofer B, Wotawa F. Why does my spreadsheet compute wrong values? *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, IEEE, 2014; 112–121.
121. Hofer B, Perez A, Abreu R, Wotawa F. On the empirical evaluation of similarity coefficients for spreadsheets fault localization. *Automated Software Engineering* 2015; **22**(1):47–74.
122. Tuya J, Suárez-Cabal MJ, De La Riva C. Full predicate coverage for testing sql database queries. *Software Testing, Verification and Reliability* 2010; **20**(3):237–288.
123. Kapfhammer GM, McMinn P, Wright CJ. Search-based testing of relational schema integrity constraints across multiple database management systems. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, IEEE, 2013; 31–40.
124. Mcminn P, Wright CJ, Kapfhammer GM. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2015; **25**(1):8.
125. Papadakis M, Henard C, Le Traon Y. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, IEEE, 2014; 1–10.
126. Ma YS, Offutt J, Kwon YR. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability* 2005; **15**(2):97–133.
127. Mutation testing systems for Java compared. http://pitest.org/java_mutation_testing_systems/. [Online; accessed 25-August-2016].
128. Namin AS, Kakarla S. The use of mutation in testing experiments and its sensitivity to external threats. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ACM, 2011; 342–352.
129. Just R, Schweiggert F, Kapfhammer GM. Major: An efficient and extensible tool for mutation analysis in a java compiler. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, 2011; 612–615.
130. Irvine SA, Pavlinic T, Trigg L, Cleary JG, Inglis S, Utting M. Jumble java byte code to measure the effectiveness of unit tests. *Testing: Academic and industrial conference practice and research techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, IEEE, 2007; 169–175.
131. Motycka W. Installation Instructions. http://sofya.unl.edu/doc/manual/installation.html 7 2013. [Online; accessed 25-August-2016].
132. Jia Y, Harman M. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. *Practice and Research Techniques, 2008. TAIC PART'08. Testing: Academic & Industrial Conference*, IEEE, 2008; 94–98.
133. Dan H, Hierons RM. Smt-c: A semantic mutation testing tools for c. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE, 2012; 654–663.
134. Delamare R, Baudry B, Le Traon Y. Ajmutator: a tool for the mutation analysis of aspectj pointcut descriptors. *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, IEEE, 2009; 200–204.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

135. Krenn W, Schlick R, Tiran S, Aichernig B, Jobstl E, Brandl H. Momut:: Uml model-based mutation testing for uml. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2015; 1–8.
136. Offutt AJ, Craft WM. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability* 1994; **4**(3):131–154.
137. Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability* 1997; **7**(3):165–192.
138. Lyu MR, Huang Z, Sze SK, Cai X. An empirical study on testing and fault tolerance for software reliability engineering. *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, IEEE, 2003; 119–130.
139. Mutation Testing and Error Seeding-White Box Testing Techniques. http://www.softwaretestinggenius.com/mutation-testing-and-error-seeding-white-box-testing-techniques. [Online; accessed 28-July-2016].

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr