SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–58
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

A Systematic Literature Review of How Mutation Testing Supports Quality Assurance Processes

Qianqian Zhu*, Annibale Panichella and Andy Zaidman

Delft University of Technology, 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 most influential realisations have been summarised by existing literature reviews, 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 191 papers published between 1981 and 2015. In particular, we analysed in which quality assurance processes mutation testing is used, which mutation tools and which mutation operators are employed. Additionally, we also investigated how the inherent core 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 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@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 the test oracle problem and equivalent mutant detection.

While existing surveys (e.g., [1, 2, 6]) provide us with a great overview of the most influential realisations in research, we lack insight into how mutation testing is actually *applied*. Specifically, we are interested in analysing in which quality assurance processes 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 quality assurance processes;
- 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 [7], we 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, i.e., we require at least a 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 quality assurance processes rather than focusing on the development of mutation tools, operators or challenges and open issues for mutation testing. This resulted in a collection containing 191 papers published from 1981 to 2015. We analysed this collection in order to answer the following two research questions:

RQ1: *How is mutation testing used in quality assurance processes?*

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 quality assurance processes 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 the usage of mutation testing and the quality assurance processes 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 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. For replicability, we mean 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 distribution of related techniques adopted under the above five facets and thereby provide guidelines for applying mutation testing and reporting the used setting/tools.

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 our findings, particularly Section 4.3 summarises the answers to the research questions, while Section 4.4 provides recommendations for future research. Section 5 discusses the threats to validity, and 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 [8, 9] 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 mutation operators to the original program more than once, stand for complex faults. The results showed that the test data generated for first-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 [10–12] and Kapoor [13].

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
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 to the program P: the rule that specifies syntactic variations are defined as a $mutation\ operator$, and the result of one application of a mutation operator is a set of $mutants$ $\mathbb{M}$. After that, each mutant $P_m \in \mathbb{M}$ is executed against T to verify whether test cases in T 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 the 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 summarised using the *mutation score* (also referred to as mutation coverage or mutation adequacy), which is defined as:

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

From the equation above, we can see that the detection of equivalent mutants is done before calculating the mutation score, as the denominator explicitly mentions non-equivalent mutants. Budd and Angluin [14] have theoretically proven that the equivalence of two programs is not decidable. Meanwhile, in their systematic literature survey, Madeyski et al. [6] have also indicated that the equivalent mutant problem takes an enormous amount of time in practice.

A mutation testing system 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 compared 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 mutant detection is also an inevitable stage of mutation testing which is a prominent undecidable problem, thereby requiring human effort to investigate. Thus, we consider the major limitations of mutation testing to be:

- the high computational cost caused by the large number of mutants;
- the undecidable Equivalent Mutant Problem resulting in the difficulty of fully automating the equivalent mutant analysis.

To deal with the two limitations above, a lot of research effort has been devoted to reduce the computational cost and to propose heuristics 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. [6]: (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 subsection, 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 as significantly better at 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 in the area of mutation

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

testing include (1) defining mutation operators [49], (2) developing mutation testing systems [17, 19, 33], (3) reducing the cost of mutation testing [30, 36], (4) overcoming the equivalent mutant detection problem [6], 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 testing 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 was then automatically generated by solving the constraint satisfaction problem using heuristics. Their proposed constraint-based 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]. Zhang and Mesbah [59] proposed assertion coverage, while Whalen et al. [60] presented observable modified condition/decision coverage (OMC/DC); these novel test criteria were also evaluated via mutation testing.

Test case prioritisation is one of the practical approaches to reducing the cost of regression testing by rescheduling test cases to expose the faults as earlier as possible. Mutation testing has also been applied to support test case prioritisation. Among these studies, two influential papers are 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.'s empirical study suggested that mutation faults can be representative of real faults [24].

The test-suite reduction is another testing 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 testing is adopted to

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Survey	Covered years	SLR?	Main idea
DeMillo [50]	1978-1989	No	Summarise the conceptual basis, development of the mutation testing at the early stage
Woodward [70]	1978-1989	No	Review the mutation testing techniques of strong, weak and firm mutation
Offutt and Untch [25]	1977-2000	No	Review the history of mutation testing and the existing optimisation techniques for mutation testing
Offutt [2]	1977-2010	No	Review past mutation analysis research starting with the Mothra project, and summarise new trends of applications of mutation testing
Jia and Harman [1]	1977-2009	No	Provide a comprehensive analysis and survey of Mutation Testing, including theories, problems, cost reduction techniques, applications, empirical evaluation, and tools
Madeyski et al. [6]	1979-2010	Yes	Present a systematic literature review in the field of the equivalent mutant problem
Hanh et al. [71]	1991-2014	No	Analyse and conduct a survey on generating test data based on mutation testing

Table I. Summary of existing surveys on mutation testing

investigate the effect of coincidental correctness in the context of a coverage-based fault localisation technique, and a novel fault localisation method by Papadakis et al. [68], [69] who used mutants to identify the faulty program statements.

2.3. Comparisons with existing literature surveys

In this section, we summarise the existing literature surveys on mutation testing and compare these surveys to our literature review. Table I lists seven literature surveys which we have found so far, including the years which the survey covered, whether the survey is a systematic literature review and the survey's main idea.

First of all, the scope of our literature review is different from the existing literature surveys. The surveys of DeMillo [50], Woodward [70], Offutt and Untch [25], Offutt [2] and Jia and Harman [1] focused on the development of mutation testing, where they summarised and highlighted the most influential realisations and findings on mutation testing. In the insightful works of Offutt and Untch [25], Offutt [2] and Jia and Harman [1], they only mentioned some of the most crucial studies which applied mutation testing to support quality assurance processes, thus, the relevant research questions posed by us could not be answered by their reviews. Madeyski et al. [6] reviewed the equivalent mutant problem which is a subarea of mutation testing. Compared to their survey work, we are more interested in how approaches for detecting equivalent mutant are actually *used* in a research context. Hanh et al. [71] analysed the literature on mutation-based test data generation, which is a subset of our literature review. Our literature review not only covers the test data generation but also other quality assurance processes, e.g., test case prioritisation and debugging.

Moreover, our literature review follows the systematic literature review (SLR) methodology [72] which is not the case for six other literature reviews (Madeyski et al. [6] being the exception): we aim to review the existing articles in a more systematic way and provide a more complete list of the existing works on how mutation testing is actually applied in quality assurance processes. It is important to mention, though, that taking a subset of Offutt and Untch [25], Offutt [2] and Jia and Harman [1]'s results regarding quality assurance applications will not give as complete a view on quality assurance applications as our SLR actually does.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

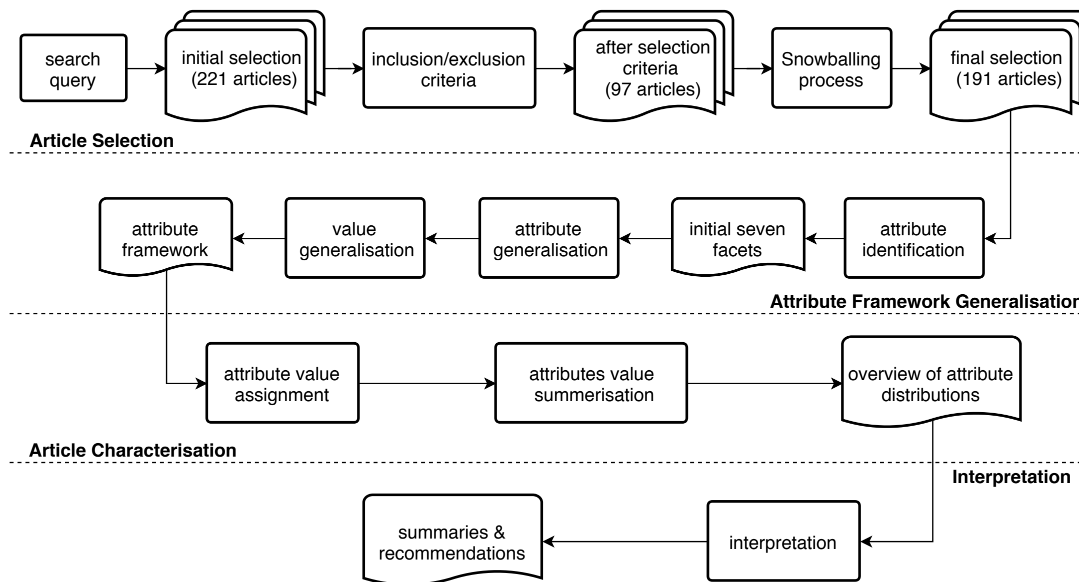

Figure 1. Overview of the systematic review process [74]

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 [7] 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, the systematic literature review aims to eliminate bias and incompleteness through a systematic mechanism [73]. Kitchenham [7] 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 [72] 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 limitations 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 interests, the application perspective of mutation testing, we naturally further divide it into two aspects: (1) how mutation testing is used and (2) how the related 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 quality assurance processes** [†]**?**

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 quality assurance processes?*

There is a broad range of quality assurance processes that can benefit from the application of mutation testing, e.g., fault localisation and test data generation. RQ1.2 seeks to uncover these activities.

RQ1.2: *Which quality assurance process does mutation testing 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 investigate whether the application of mutation testing is also mostly done at the unit testing level, or whether other levels of testing have been also investigated in the literature.

RQ1.3: *Which test level does mutation testing 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 testing strategies can also benefit from the application of mutation testing.

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

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

RQ2: **How are empirical studies related to 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.

[†]The quality assurance processes include testing activities and debugging in general. In more specific, the quality assurance processes include all the daily work responsibilities of test engineers (e.g. designing test inputs, producing test case values, running test scripts, analyzing results, and reporting results to developers and managers) [15].

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

RQ2.1: *Which mutation testing tools are being used?*

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 are being used?*

The equivalent mutant problem, i.e., the situation where a mutant leads to a 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. [6] 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 in practice when applying mutation testing during quality assurance activities. RQ2.3 aims to seek answers for exactly this question.

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

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. However, 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 subjects are being used in the experiments (regarding programming language, size, and data availability)?*

3.2. Study Selection Strategy

Initial Study Selection. We started with search 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 considered the articles published in seven top journals and ten top conferences (as listed in Table II) dating from 1971 as data sources. The above 17 venues are chosen because they 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

[‡] In the snowballing procedure, we took the "Mutation Testing" workshop series into consideration, since this is the closest venue to mutation testing.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

the paper's language is not English, we also excluded such a paper. After this step, 220 papers were initially selected.

Type	Venue Name	No. of papers After Applying Search queries	No. of papers After Applying In./Ex. Criteria	No. of papers After Snow-balling procedure
Journal	Journal of Empirical Software Engineering (EMSE)	4	3	6
	Information and Software Technology (IST)	0	0	3
	Journal Software Maintenance and Evolution (JSME)	0	0	0
	Software Quality Journal (JSQ)	0	0	2
	Journal of Systems and Software (JSS)	17	8	9
	Journal on Software Testing, Verification and Reliability (STVR)	33	16	23
	Transaction on Software Engineering and Methodology (TOSEM)	3	2	4
	Transaction on Reliability (TR)	1	1	1
	Transaction on Software Engineering (TSE)	19	9	21
Conference	**Proceedings Asia Pacific Software Engineering Conference (APSEC)**	0	0	1
	International Conference on Automated Software Engineering (ASE)	7	3	7
	European Software Engineering Conference / International Symposium on the Foundations of Software Engineering (ESEC/FSE)	6	1	9
	International Symposium on Empirical Software Engineering and Measurement (ESEM/ISESE)	2	1	3
	International Conference on Software Engineering (ICSE)	29	9	22
	International Conference on Software Maintenance and Evolution (ICSME/ICSM)	6	3	9
	International Conference on Software Testing, Verification, Validation (ICST)	45	23	22
	International Symposium on Software Reliability Engineering (ISSRE)	26	10	20
	International Symposium on Software Testing and Analysis (ISSTA)	14	3	12
	Proceedings International Conference on Quality Software (QSIC)	8	5	6
	Proceedings International Symposium on Search Based Software Engineering (SSBSE)	0	0	1
	Proceedings of the International Conference on Testing Computer Software (TCS)	0	0	1
Workshop	**International Workshop on Mutation Analysis**	0	0	9
	Total	220	97	191

Note: the venues marked in **bold** font are not initially selected, but where added after the snowballing procedure. We listed the venues alphabetically according to their abbreviations, e.g., EMSE is ahead of IST.

Table II. Venues Involved in study selection

Inclusion/Exclusion Criteria. Since we are interested in how mutation testing is *applied* in a research context, *thereby not excluding industrial practice*, we need selection criteria to include the papers that use mutation testing as a tool for evaluating or improving other quality assurance processes and exclude the papers focusing on the development of mutation tools and operators, or challenges and open issues for mutation testing. Moreover, the selected articles should

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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 the quality assurance processes. This criterion excludes the research *solely* on mutation testing itself, such as defining mutation operators, developing mutation systems, investigating ways to solve open issues related to mutation testing 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 quality assurance processes. 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 of this SLR 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, 97 papers remained.

Snowballing Procedure. After selecting 97 papers from digital databases and applying our selection criteria, there is still a high potential to miss articles of interest. As Brereton et al. [72] pointed out, most online platforms do not provide adequate support for systematic identification of relevant papers. To overcome this shortfall of online databases, we then adopted both backward and forward snowballing strategies [75] 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 [75]. Using the references and the citations respectively are referred to as backward and forward snowballing [75].

We used the 97 papers as the starter 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 82 articles (and five additional venues). The International Workshop on Mutation Analysis was added during the snowballing procedure.

To check the completeness of the initial study collection, we first ran a reference check based on Jia et al.'s survey (among 264 papers) as our literature review was initially motivated from their paper. The results showed that: (1) 5 papers have already been included in our collection; (2) 3 additional papers that should be included; and (3) 246 papers are excluded. Again, we applied snowballing techniques to the additional three papers, and the three papers resulted in a total of 12 papers for our final collection (191 papers in total[¶]).

Furthermore, we ran a sanity check on our final collection to examine how many papers do not have the keywords "mutation analysis" or "mutation testing" in their abstracts, titles or keywords. The sanity check resulted in 112 papers; 15 papers are

[§]The studies which *merely* adopted hand-seeded faults which are not based on a set of mutation operators are not part of this survey.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

missing in the initial data collection by applying search queries in online platforms. Most of the missing papers (10 out of 15) (e.g., Offutt et al. [65] and Knauth et al. [76]) are not from the pre-considered 17 venues. The results of the sanity check indicate that there are potentials of missing papers based on search queries in online platforms; however, the snowballing procedure can minimise the risks of missing papers.

The detailed records of each step can be found in our GitHub repository [77].

3.3. Data Extraction Strategy

Data extracted from the papers are used to answer the research questions we formulated in Section 3.1. 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 quality assurance processes; (2) the quality assurance processes; (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 III.

For each facet, we first read the corresponding details in each paper and extracted the exact text from the papers. During the reading procedure, we started by identifying and classifying 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 merged some values 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. Moreover, the attribute framework can also be further used for validation and replication of the review work. To categorise the attributes for each paper, all the abstracts, introductions, empirical studies and conclusions of the selected papers were carefully read. If these sections were not clear or were somehow confusing, we also took other sections from the paper into consideration. Furthermore, for categorising the test level attribute, to minimise misinterpretations of the original papers, we looked beyond the aforementioned sections to determine the test level (i.e., “unit”, “module”, “integration”, “system” and “acceptance” [15]). In particular, we used the former five words as keywords to search for the entire paper. If this search yielded no results, we did not necessarily categorise the paper as “n/a”. Instead, we read the entire paper, and if a study deals with a particular type of testing, e.g., testing of the grammar of a programming language or spreadsheet testing, we placed the paper in the category “others”. If the paper lacks any description of the test level, we classified the test level as “n/a”.

(1) roles of mutation testing in quality assurance processes:

The first facet concerns the role of mutation testing in quality assurance processes 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.

¶We did not control for double counting here as there are usually additional experiments and further discussion in the extended version.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

To identify and classify the role of mutation testing, we mainly read the description of mutation testing in the *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 the paper into the "guide" set. In this case, mutation testing is not used in the final step of the experiment.

(2) quality assurance processes:

The second facet focuses on quality assurance processes. Three attributes are relevant to quality assurance processes: the categories of quality assurance processes (RQ1.2), test levels (RQ1.3) and testing strategies (RQ1.4). To identify the categories of quality assurance processes, we group similar quality assurance processes based on information in title and abstract. The quality assurance processes we identified so far consist of 12 classes: test data generation, test-suite reduction/selection, test strategy evaluation, test case minimisation, test case prioritisation, test oracle, fault localisation, program repairing, development scheme evaluation, model clone detection, model review, and fault tolerance. 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 consider 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. Also, for the paper which is difficult for us to categorise into any of the four phases (e.g., testing of the grammar of a programming language and spreadsheet testing) we mark this situation as "others".

For testing strategies, a 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 [78, 79]. For the structural testing and specification-based testing classes, we further divided the classes into *traditional* and *enhanced*, based on whether other methods improve the regular testing.

To be classified into the "structural testing" class, the paper should either contain the keywords "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 "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; 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 grey-box testing combines structural testing and specification testing. 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".

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Furthermore, to classify the "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 [80] 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 them as "enhanced".

(3) mutation tools used in experiments:

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 the mutation tool, which is based on 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, manual 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 [81], jFuzz [82], TrpAutoRepair [83], and GenProg [84]. The self-written tool category represents those tools that have been developed by the authors of the study. The "manual" value means the mutants were generated manually according to mutation operators 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 manually generated or not.

(4) mutation operators used in experiments: As for the mutation operators (related to RQ2.2), we focus on two attributes: description level and generalised classification. The former is again designed to assess the repeatability issue related to mutation testing. The description degree depends on the way 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 the complete list of mutation operators, then we classify such a paper into "well-defined". The available full list includes two main situations: (1) the authors listed each name of mutation operator(s) 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 where all or the default set of mutation operators provided by that tool were used). Thus, the well-defined category enables the traceability of the complete list of mutation operators. Instead, if there is some information about the mutation operators in the article but not enough for the 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. Finally, we use the label "n/a" when no description of the mutation operators was given in the paper at all.

To compare the mutation operators from *different tools* for *different programming languages*, and 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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

mutation testing tools use different naming conventions 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 [85]. Therefore, we saw a need to compose a generalised 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 generalised classification is to group the similar mutation operators 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 a similar vein to Jia and Harman’s survey. As we are more interested in program mutation, we further analysed this area and summarised different mutation operators based on literature. More specifically, we first followed the categories and naming conventions of MuJava [86, 87] and Proteum [49] as their mutation operator groups are more complete than the others. Based on the mutation operator groups from MuJava and Proteum, we further divided the program mutation operators into three sub-categories: expression-level, statement-level, and others. The expression-level mutation operators focus on the inner components of the statements, i.e., operators (method-level mutation operators in MuJava [86]) and operands (Constant and Variable Mutations in Proteum [49]), while the statement-level ones mutate at least a single statement (Statement Mutations in Proteum [49]). As we are interested in a more generalised classification independent of the programming language, we came up with the class of “others” to include mutation operators related to the programming language’s unique features, e.g., Objected-Oriented specific and Java-specific mutation operators (Class Mutation in MuJava [87]). It is important to note that our generalised classification of mutation operators aims to provide an overall distribution of different mutation operator groups. Thus, we did not look into lower-level categories. If the paper used one of the mutation operators in one group, we assign the group name to the paper. For example, while the PIT tool only adopts a small number of arithmetic operators [85], we still assign PIT with the “arithmetic operator”.

Our generalised classification of mutation operators is as follows:

Listing 1. Generalised classification of mutation operators

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.

[*]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.

DOI: 10.1002/stvr

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 “+=”, “− =”, “*=”, “/=”, “%=”, “&=”, “|=”, “^=”, “<<=”, “>>=”, “>>>=”) 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 a compatible type.[‡]
x. **type:** it replaces a type with another compatible type 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` statements 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`.

[†]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”

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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.
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 [87], 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 [87] (the operators in Java-Specific Features), e.g. `this` keyword insertion.
iv. **JavaScript-specific:** the mutation operators related to JavaScript-specific features [88] (including DOM, JQUERY, and XMLHTTPREQUEST operators), e.g. `var` keyword deletion.
v. **SpreadSheet-specific:** the mutation operators related to SpreadSheet-specific features [89], e.g. changing the range of cell areas.
vi. **AOP-specific:** the mutation operators related to A(spect)-O(riented)-P(rogramming) features [90, 91], e.g. removing pointcut.

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

vii. **concurrent mutation:** the mutation operators related to concurrent programming features [92, 93], e.g. replacing `notifyAll()` with `notify()`.

viii. **Interface mutation:** the mutation operators related to Interface-specific features [94, 95], 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 III).

For the equivalent mutant problem, we started by searching the keywords "equivalen*" and "equal" in each paper to target the context of the equivalent mutant 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 [96] ranks mutations by impact to help users detect the equivalent mutants. However, if the authors who used JAVALANCHE did not specify that internal feature, we do 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. Also, we excluded runtime optimisation and selective mutation. The former one, runtime optimisation, is an internal optimisation adopted during the tool implementation, therefore 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,

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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 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` [97], `GitHub`**, `SF100` [98] 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 as “No”.

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

Facet	Attribute	Value	Description
Roles	classification	assessment	assessing the fault-finding effectiveness
		guide	improving other quality assurance processes as guidance
Quality assurance processes	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
		fault tolerance	assessing the ability of the system to continue operating properly in the event of failure
	test level	unit	quality assurance processes focus on unit level. A typical example of unit testing includes: using unit testing tools, such as Junit and Nunit, intra-method testing, intra-class testing.
		integration	quality assurance processes focus on integration level. A typical example of integration testing includes: caller/callee and inter-class testing
		system	quality assurance processes focus on system level. A typical examples of system testing include: high-level model-based testing techniques and high-level specification abstraction methods
		others	quality assurance processes are not related to source code. A 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

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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

		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
		grey-box	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
		manual	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
	generalised classification	refer to Listing 1	refer to Listing 1
Equivalence Solver	methods	not killed as equivalent	treating mutants not killed as equivalent
		not killed as non-equivalent	treating mutants not killed as non-equivalent
		no investigation	no distinguishing between equivalent mutants and non-equivalent 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 a 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 contain more than one faults
		selection strategy	generate fewer mutants by selecting where to mutate based on a 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 III. 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 [72]. 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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

our review protocol. The validation focuses on 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 the 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.

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 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 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 quality assurance processes information from the papers, we can simply choose one or several options of the 12 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 our 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 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 which means the fixed number of selected mutants.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Subjects: We changed the values of "data availability" from "open source", "industrial" or "self-defined" to "Yes" or "No". Since the previous definitions cannot distinguish between available and unavailable datasets.

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 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 our GitHub repository [77].

4.1. RQ1: How is mutation testing used in quality assurance processes?

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

Observation. We opt to discuss the two research questions RQ1.1 and RQ1.2 together, because it gives us the opportunity to analyse per quality assurance (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 IV, in which we report the role mutation testing plays in the two columns "Assessment" and "Guide" (see Table III for the explanation about our attribute framework), while the quality assurance processes 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 IV shows, test data generation and test strategy evaluation occupy the majority of quality assurance processes (accounting for 75.9%) and test suite reduction/selection (7.9%). 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 quality assurance processes, we will discuss them separately. Also, for the "guide" 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 or debug techniques (70.2%) as it is widely considered as a "high end" test criterion [15]. To this aim, mutation testing typically generates a significant number of mutants of a program, which are sometimes also combined with natural defects or hand-seeded

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Testing Activity	**Assessment**	**Guide**	**Total**
test data generation	38	36	75
test strategy evaluation	63	6	70
test oracle	13	5	18
test case prioritisation	11	6	17
test-suite selection/reduction	10	5	15
fault localisation	8	4	12
program repairing	2	1	3
test case minimisation	1	1	2
fault tolerance	1	0	1
development scheme evaluation	0	1	1
model clone detection	1	0	1
model review	1	0	1
Total	134	57	191

Table IV. quality assurance processes summary

ones. The results of the assessment are usually quantified as metrics of fault-finding capability: mutation score (or mutation coverage, mutation adequacy) [99, 100] and killed mutants [101, 102] are the most common metrics in mutation testing. Besides, in test-case prioritisation, the Average Percentage of Faults Detected (APFD) [103, 104], 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 19 studies that performed *mutant analysis*, which means that the researchers tried 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. [105] 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 quality assurance process, we will highlight the most significant research efforts to demonstrate why mutation testing can be of benefit as a building block to guide other quality assurance processes. 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 (36 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 (73.3%) are satisfied with a weak mutation condition (necessity), while a strong mutation condition (sufficiency) appears less (33.3%). The same is true when comparing first-order mutants (93.3%) to higher-order

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

mutants (6.7%). Except for the entirely automatic test data generation, Baudry et al. [106–108] 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. [109] and Smith and Williams [110] augmented test input data using the requirement of killing as many mutants as possible. Compared to the existing literature survey of Hanh et al. [71], which shed light on mutation-based test data generation, we cover more studies and extend their work to 2015.

The second-most-frequent use cases when applying mutation testing to guide the testing efforts come from test case prioritisation (6 instances) and 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. [111] ordered test cases according to prior fault detection rate using both hand-seeded and mutation faults. Kwon et al. [112] proposed a linear regression model to reschedule test cases based on Information Retrieval and coverage information, where mutation testing determines the coefficients in the model. Moreover, Rothermel et al. [61, 103] and Elbaum et al. [62] compared different approaches of test-case prioritisation, which included the prioritisation in order of the probability of exposing faults estimated by the killed mutant information. In Qi et al. [83]'s study, they adopted a similar test-case prioritisation method to improve patch validation during program repairing.

Zooming in on the test strategy evaluation (6 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. [113]: in their influential work they examined the fault-exposing potential (FEP) coverage adequacy which is estimated by mutation testing 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.

When it comes to the test oracle problem (5 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 [114]'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. [115] and Gay et al. [116] selected the most "effective" oracle data by ranking variables (trace data) based on their killed mutants information.

Mutation-based test-suite reduction (5 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. [117] who

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

compared the trade-offs among various test-suite reduction techniques based on statement coverage and killed mutants.

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. [118] 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. [119] ranked the suspiciousness of "faulty" statements based on their passing and failing test executions of the generated mutants.

From the aforementioned guide roles of the quality assurance processes, 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 an issue unique to mutation testing, but rather an inherent challenge of test data generation. As mentioned above, mutation testing cannot only help in precisely targeting at *where* to check but also suggesting *what* to check for [114] (see the first recommendation labelled 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 quality assurance processes, 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 V. We can see 43 cases (75.4%), which is the addition of the first and the third rows in Table V (39 + 4), still adopted mutation faults to assess the effectiveness. Among these studies, four instances [115, 116, 120, 121] 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 as evaluation set. Besides, one study [55] used a different mutation tool, while the other [94] 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 labelled 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 quality assurance processes when applying mutation testing (75.9%). While as guidance, mutation testing is primarily used in test data generation (36 instances), test strategy evaluation (6 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

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Evaluation Fault Type	Total
mutation faults	39
hand-seeded faults	7
hand-seeded + mutation faults	4
no evaluation	4
real faults	2
other coverage criteria	1

Table V. Guide Role Summary

Test Level	Total
n/a	84
unit	77
integration	15
other	10
system	10

Table VI. Test Level Summary

adequate fault set instead of mutation faults to effectively evaluate mutation-based testing methods. The recommendation is that mutation testing can suggest not only *where* to check but also *what* to check. *Where* to check is widely used to generate killable mutant constraints in different quality assurance processes, while *what* to check is seldom adopted to improve the test data quality.

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

Observations. Table VI presents the summary of the test level distribution across the articles. We observe that the authors of 84 papers do not provide a clear clue about the test level they target (the class marked as "n/a"). For example, Aichernig et al. [122]'s study: they proposed a fully automated fault-based test case generation technique and conducted two empirical case studies derived from industrial use cases; however, they did not specify which test level they targeted. One good instance that clearly provides the information of the test level is Jee et al. [123] where they specified that their test case generation technique for FBD programs is at unit testing level. This is an open invitation for future investigations in the area to be clearer about the essential elements of quality assurance processes 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 107 (191 – 84) papers.

Looking at Table VI, mutation testing mainly targets the unit-testing level (72.0%), an observation which is in accordance with the results of 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.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

While unit testing is by far the most observed test level category in our set of papers, higher-level testing, such as integration testing (15 instances), can also benefit from the application of mutation testing. Here we highlight some research works as examples: Hao et al. [124] and Do and Rothermel [64] used the programs with system test cases as their subjects in case studies. Hou et al. [94] studied interface-contract mutation in support of integration testing under the context of component-based software. Li et al. [125] 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. [126] defined and evaluated adequacy criteria under system-level testing for distributed systems. In Denaro et al. [127]'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., [128,129], are seldom used in these studies. The only three exceptions in our collections are Flores and Polo [130], Vincenzi et al. [131] and Flores and Polo [132], who adopted interface mutation to evaluate the integration testing techniques. This reveals a potential direction for future research: the cross-comparison of different levels of mutation operators and quality assurance processes at different test levels (see the third recommendation labelled as **R3** in Section 4.4). The investigation of different levels of mutants can explore the effectiveness of mutation faults at different test levels, such as the doubts whether the integration-level mutation is better than unit-level mutation when assessing testing 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 [133–135], spreadsheet-related testing [89,136,137] and SQL-related testing [138–140]. 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 (44.0% 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 quality assurance processes 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?

Observations. In Table VII we summarised the distribution of testing strategies based on our coarse-grained classification (e.g., structural testing, specification-based testing) as mentioned in Table III. Looking at Table VII, structure-based testing (including the first and the third rows in

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Testing Strategies	Total
structural testing	57
specification-based testing	52
structural testing (enhanced)	52
others	21
specification-based testing (enhanced)	13
hybrid testing	7
similarity-based testing	3

Table VII. Testing Strategies summary

Table VII), 109 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 (including the second and the fifth rows in Table VII) accounts for 65 cases; (2) hybrid testing (combination of structural and structure-based testing) for seven instances, e.g., Briand et al. [141] investigated how data flow information can be used to improve the cost-effectiveness of state-based coverage criteria; (3) three cases applying mutation testing in similarity-based testing; (4) 21 instances in others, e.g., static analysis.

One interesting finding is that enhanced structural testing ranks third, 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 13 instances mainly adopted mutation testing (e.g., Qu et al. [111] and Papadakis et al. [142]) to improve the testing strategies.

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. However, structural testing is more popular than the others (57.1%). 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 mutation testing designed and reported?

4.2.1. RQ2.1: Which mutation testing tools are being used?

Observations. We are interested in getting insight into the types (as defined in Table III) 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 VIII. As shown in Table VIII, 50.3% of the studies adopted existing tools which are open to the public; this matches our expectation: as mutation

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Availability	Types	Total	
Yes	existing	96	103
	partially-based	7	
	self-written	1	
No	n/a (no information)	44	92
	existing (given the name/citation)	22	
	self-written	14	
	manual	12	

Table VIII. Mutation Tool Summary

testing is not the main focus of the studies, if there exists a well-performing and open-source mutation testing tool, researchers are likely willing to use these tools. However, we also encountered 15 cases of using self-implemented tools and 12 studies that manually applied mutation operators. A likely reason for implementing a new mutation testing tool or for manually applying mutation operators is that existing tools do not satisfy a particular need of the researchers. Besides, most existing mutation testing tools 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 future research [2, 143] (see the fourth recommendation labelled as **R4** in Section 4.4).

Unfortunately, there are also still 92 studies (48.2%) that do not provide access to the tools, in particular, 44 papers did not provide any accessible information about the tools (e.g., Hong et al. [92], Belli et al. [101] and Papadakis and Malveris [144]), a situation that we marked as "n/a" in Table VIII. This unclarity should serve as a notice to researchers: the mutation testing tool is one of the core 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 cannot only reveal their level of 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 IX. Table IX shows that we encountered 19 mutation tools in total. Most tools target one programming language (except for Mothra [17], which supports both C and Fortran). We encountered seven mutation tools for Java, with the top 3 most-used being MuJava [145], JAVALANCHE [96] and Major [146]. We found that four mutation tools for C are used, where Proteum [18] is the most-frequently applied. Proteum/IM [147] is a special mutation testing tool that targets interface mutation, which concerns integration errors. The integration errors are related to a connection between two units and the interactions along the connection, such as a wrong subprogram call.

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 (19 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

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Language	Tool	Total
Java	MuJava/μ-java/Muclipse	41
	JAVALANCHE	9
	MAJOR	9
	PIT/PiTest	7
	Jumble	2
	Sofya	1
	Jester	1
C	Proteum	12
	Proteum/IM	3
	MiLu	2
	SMT-C	1
Fortran	Mothra	4
SQL	SQLMutation/JDAMA	3
	SchemaAnalyst	1
C#	GenMutants	1
	PexMutator	1
JavaScript	MUTANDIS	3
AspectJ	AjMutator	2
UML specification	MoMuT::UML	1

Table IX. Existing Mutation Tool Summary

24 mutation tools. Also important to stress, is that the goal of Jia and Harman's survey is different to 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 when compared to Jia and Harman [1]'s data.

Summary. 50.3% of the articles that we have surveyed adopt existing (open access) tools, while in a few cases (27 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 44 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 testing.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Description Level	Total
well-defined	119
n/a	44
not sufficient	28

Table X. Description Level of Mutation Operators Summary

4.2.2. RQ2.2: Which mutation operators are being used?

Observations For the mutation operators, we first present the distribution of the three description levels (as mentioned in Table III) in Table X. As Table X shows, 62.3% (119 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 are labelled as "n/a" (e.g., Briand et al. [148], DeMillo and Offutt [149] and Shi et al. [150]).

After that, based on our generalised classification of the mutation operators (as defined in Listing 1), we characterised the 119 papers labelled as "well-defined". In addition to the overall distribution of the mutation operators regardless of the programming language, we are also interested in the differences of the mutation operators for different languages as the differences could indicate potential gaps in the existing mutation operator sets for certain programming languages. In Table XI we project the various 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 Table XI. Among the program mutation operators, the arithmetic, relational and conditional operators are the top 3 mutation operators. These three operators are often used together in most cases as their total number 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 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 (note that Table XI only listed seven programming languages that have been widely used). Moreover, as Jia and Harman [1] also discussed mutation operators in their review, it is interesting to see whether their summary agrees with our work. Therefore, we also compared our findings to Jia and Harman's as follows:

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 by our survey: type, bomb statement,

*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

Level	Operator	Java	C	C++	C#	Fortran	SQL	JavaScript	Total
Specification Mutation		2	2	1	-	-	-	-	23
Program Mutation		55	12	4	3	2	5	1	95
Expression-level	arithmetic operator	51	10	4	1	2	3	-	79
	relational operator	47	8	4	1	2	3	-	74
	conditional operator	47	7	4	2	2	3	-	72
	bitwise operator	36	4	2	-	-	-	-	43
	assignment operator	33	4	2	-	-	-	-	39
	constant	18	5	2	-	2	3	-	37
	shift operator	33	2	2	-	-	-	-	36
	variable	15	4	2	-	2	3	1	31
	absolute value	19	3	2	1	1	3	-	31
	conditional expression	9	3	-	1	-	-	-	14
	parenthesis	1	2	-	-	-	-	-	3
	type	-	3	-	-	-	-	-	3
Statement-level	statement deletion	11	4	-	2	2	-	-	23
	method call	11	-	2	-	-	-	1	16
	`return` statement	10	3	-	-	2	-	-	16
	control-flow disruption	6	2	2	-	-	-	-	9
	exception handler	1	-	1	-	-	-	-	5
	`goto` label	-	3	-	-	2	-	-	6
	statement swap	2	2	2	-	-	-	-	5
	bomb statement	-	2	-	-	2	-	-	5
	`switch` statement	2	3	-	-	-	-	-	5
	`do` statement	-	2	-	-	2	-	-	5
	brace	-	3	-	-	-	-	-	3
	loop trap	-	3	-	-	-	-	-	3
	`while` statement	-	3	-	-	-	-	-	3
	`if` statement	-	-	-	-	-	-	-	-
Others	OO-specific	23	-	-	-	-	-	-	26
	Java-specific	17	-	1*	-	-	-	-	17
	Interface mutation	4	2	-	-	-	-	-	7
	SQL-specific	-	-	-	-	-	5	-	5
	Concurrent mutation	4	-	-	-	-	-	-	4
	AOP-specific	-	-	-	-	-	-	-	3
	Spreadsheet-specific	-	-	-	-	-	-	-	2
	JavaScript-specific	-	-	-	-	-	-	1	1

Table XI. Mutation Operators Used In our collection

`do` statement, brace, loop trap, `while` statement and `if` statement. Compared to Jia and Harman's survey, Alexander et al. [151,152]'s design of Java Object mutation cannot be found in our collection.

2. For C, only two operators are not covered by our dataset. The C programming language does not provide direct support for exception handling. There is no article applying mutation operators that target specific C program defects or vulnerabilities surveyed by Jia and Harman, such as buffer overflows [153] and format string bugs [154].
3. For C++, 3 expression-level, 10 statement-level and the OO-specific operators are not used in our dataset. Jia and Harman have not covered mutation operators for C++.

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

DOI: 10.1002/stvr

4. For C#, only a limited set of mutation operators are applied based on our dataset. Our collection has no application of OO-specific operators [155] summarised in Jia and Harman's survey.
5. For Fortran, the earliest programming language mutation testing was applied to, the studies in our collection cover a basic set. These mutation operators agree with Jia and Harman's work.
6. For SQL, since the syntax of SQL is quite different from imperative programming languages, only six operators at the expression level and SQL-specific ones are used in our dataset. Compared to Jia and Harman's literature review, a set of mutation operators addressing SQL injection vulnerabilities [156] are not found in our collection.
7. For JavaScript, only three JavaScript-specific mutation operators are adopted in studies we selected. Mutation operators for JavaScript [157] have not been covered by Jia and Harman as the paper introducing them (i.e., Milani et al. [157]) is more recent.
8. For interface mutation, we have found studies solely targeting Java and C in our selection.

The comparison with Jia and Harman's literature review has shown that for most programming languages (except for C++ and JavaScript), a few mutation operators summarised by Jia and Harman have no *actual* applications in our collection. As we only considered 22 venues, we might miss the studies that adopted these mutation operators in other venues. Without regard to the potential threat of missing papers in our dataset, these mutation operators that have no applications in our collection pose interesting questions for further exploration, e.g., why are these operators seldom used by other researchers? What is the difference between these operators and other widely used operators?

Also, from the above findings, we can see that for different languages the existing studies did not cover all the mutation operators that we listed in Table XI: some are caused by the differences in the syntax, while the others could point to potential gaps. However, these potential gaps 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. We can only say that Table XI can be a valuable reference for further investigations into mutation operator differences and gaps in different programming languages.

Furthermore, our generalised classification of the existing mutation operators can also be of benefit to compare mutation tools in the future. Thereby, we compared the existing mutation testing tools (as listed in Table IX) to our mutation operator categories in Table XII. Table XII is based on the documentation or manuals of these tools. Here we used the definitions of mutation operator groups from Listing 1 (mainly based on MuJava [86, 87] and Proteum [49]) as the baseline: if there is a possible mutation missing in a group for a mutation testing tool, we marked "*" in Table XII. As there exist different syntaxes in different programming languages, we also consider syntactic differences when categorising the mutation operators of different tools. For example, there is no modulus operator "%" in Fortran (but a `MOD` function instead), therefore, when considering the arithmetic mutation operators for Fortran (Mothra), we do not require the modulus operator "%" to be included in Mothra.

It is important to mention that Table IX is not the complete list of all the existing mutation tools that have been published so far; these tools are chosen to investigate how mutation testing supports quality assurance processes. We analyse them here as they are open to public and have

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

	MuJava/μ-java/Muclipse [86, 87, 158]	PIT/PiTest [85, 159]	JAVALANCHE [96, 160]	MAJOR [161, 162]	Jumble [163]	Sofya [164]	Jester [165]	Proteum [49, 166]	MiLu [167]	SMT-C [168]	Proteum/IM [147, 169]	Mothra [17]	SQLMutation/JDAMA [20]	SchemaAnalyst [170]	GenMutants [171]	PexMutator [172]	MUTANDIS [88]	AjMutator [173]	MoMuT::UML [174]
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	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	√	-
interface mutation	-	-	-	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-

Note: the entry marked with * means the tool does not provide the full possible mutations.

Our summary of the mutation tools is based on the available manuals and open repositories (if they exist) for the tools. If there are different versions of the tools, we only consider the newest one.

Table XII. Comparison of Mutation Operators in Existing Mutation Tools

been applied by researchers at least once. The analysis of mutation operators in these tools could also be a valuable resource for researchers in mutation testing to consider.

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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

on the expression-level mutations and only five 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 four mutation testing tools for C (including Mothra) that we have considered, Proteum covers the most mutation operators. SMT-C is an exceptional case of the traditional mutation testing which targets semantic mutation testing. Proteum/IM is the only mutation tool listed in Table XII that supports interface mutation. For the tools designed for C#, OO-specific operators are not present.

Moreover, when we further analyse the missing mutations in each mutation operator group (marked as "*" in Table XII), we found that most tools miss one or several mutations compared to our generalised classification. Particularly for the arithmetic operator, only MuJava and Proteum apply all possible mutations. The other tools that adopt the arithmetic operator all miss Arithmetic Operator Deletion (as defined in MuJava [86]).

Another interesting finding when we compared Table XI and Table XII, 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. Therefore, we zoom in on this finding and investigate whether it is a common case that only a subset of the mutation operators from the existing mutation tools is adopted in studies based on our collection. The result shows that 21 studies out of 54[†] applied a subset of the mutation operators from the existing tools; this reinforces our message of the need for "well-defined" mutation operators when reporting mutation testing studies.

Summary. For the mutation operators, we focused on two attributes: their description level and a generalised classification across tools and programming languages. When investigating the description level of the mutation operators that are used in studies, we found that 62.3% (119 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 when applying mutation testing?

Observations. In Table XIII, we summarised our findings of how the studies that we surveyed deal with the equivalent mutant problem. More specifically, Table XIII presents how many times we encountered each of the approaches for solving the equivalent mutant problem. When looking at the results, we first observe that in 56.5% of the cases we assigned "n/a", such as Androutsopoulos et al. [104] and Flores and Polo [132].

[†]The studies that are categorised as "yes" in the tool availability, "existing tool" in the tool type and "well-defined" in the mutation operator description level.

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Equivalence Detector	Total
n/a	108
manual investigation	38
not killed as equivalent	17
no investigation	11
reduce likelihood	8
model checker	6
deterministic model	3
not killed as non-equivalent	3

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

As shown in Table XIII, there are only 17 instances actually adopting equivalent mutant detectors by using automatic mechanisms. Specifically, 6 instances use "model checker" (e.g., Gay et al. [175]); 8 instances use "reduce likelihood" (e.g., Milani et al. [157]); and 3 instances apply "deterministic model" (Belli et al. [101]). In the remaining papers, the problem of equivalent mutants is solved by: (1) manual analysis (e.g., Liu et al. [176] and Xie et al. [177]); (2) making assumptions (treating mutants not killed as either equivalent or non-equivalent, e.g., Fang et al. [178] and Rothermel et al. [61]); (3) no investigation (e.g., Offutt and Liu [179], Chen et al [113] and Fraser and Zeller [114]). The manual investigation (38 instances) and the method of treating mutants not killed as equivalent (17 instances) are more commonly used than other methods. We also compared our results with Madeyski et al. [6]'s survey on the equivalent mutant problem. In their review, they reviewed 17 methods for overcoming the equivalent mutant problem. Among these 17 techniques, we found that only the model-checker approach [42] was adopted.

We can only speculate as to the reasons behind the situation above: Firstly, most studies use mutation testing as an evaluation mechanism or guiding heuristic, rather than their main research topic. So, the authors might be 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 XIV), we found that only five tools adopt techniques to address the equivalent mutant problem. Most of the tools did not assist in dealing with the equivalent mutant problem. Therefore, in this paper, we consider the aforementioned three solutions: (1) manual analysis, (2) making assumptions, or (3) no investigation. 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 labelled as **R5** in Section 4.4).

Secondly, the 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) [103]. Furthermore, the definition of the mutation score is also modified by some authors (e.g., Rothermel et al. [61]) : they used the total number of mutants as the denominator instead of the number of non-equivalent mutants. The equivalent mutant problem seems to not pose a significant threat to the validation of the testing techniques involved in these studies.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Language	Tool	Equivalent Mutants	Cost Reduction
Java	MuJava/μ-java/Muclipse	n/a	MSG, bytecode translation (BCEL) [145]
	PIT/PiTest	n/a	Bytecode translation (ASM), coverage-based test selection [180]
	JAVALANCHE	Ranking mutations by impact [96]	MSG, bytecode translation (ASM), coverage-based test selection, parallel execution [96]
	MAJOR	n/a	Compiler-integrated, coverage-based test selection [146]
	Jumble	n/a	Bytecode translation (BCEL), conventional test selection [181]
	Sofya	n/a	Bytecode translation (BCEL) [182]
	Jester	n/a	n/a
C	Proteum	n/a	Mutant sample [18]
	MiLu	Trivial Compiler Equivalence [183]	Higher-order mutants, test harness [184]
	SMT-C	n/a	Interpreter-based, weak mutation [185]
	Proteum/IM	n/a	Compiler-based, control flow optimisation [147]
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 [88]	Selection strategy [88]
AspectJ	AjMutator	Static analysis [173]	Compiler-based [173]
UML specification	MoMuT::UML	n/a	Compiler-based [174]

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 XIV. Inner features of Existing Mutation Tool

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 [186, 187]. What’s more, Schuler and Zeller [47] 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 that further research on the equivalent mutant problem can develop a mature and useful auxiliary tool which can easily

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

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 the validation of the quality assurance processes. Also, 56.5% of the studies are lacking an explanation as to how they are dealing with overcoming the equivalent mutant 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 when applying mutation testing?

Observations. Since mutation testing requires high computational demands, cost reduction is necessary for applying mutation testing, especially in an industrial environment. We summarized the use of such computational cost reduction techniques when using mutation testing in Table XV. 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 131 articles (68.6%) did not mention any reduction techniques, e.g., Außerlechner et al. [89] and Baker and Habli [188]. 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 (28 instances, e.g., Namin and Andrews [57] and Staats et al. [189]), followed by weak mutation (15 instances, e.g. Hennessy and Power [133] and Vivanti et al. [190]) and mutant sampling (11 cases, e.g. Arcuri and Briand [191] and Stephan and Cordy [192]). However, why is the technique of using a "fixed number" of mutants more popular than the others? We speculate that this could be because choosing a certain number of mutants is more realistic in real software development: the total number of mutants generated by mutation tools is enormous; 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. It is also important to note that 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 a fixed sampling rate. Compared to using a fixed number, mutant sampling sometimes cannot achieve the goal of reducing the number of mutants efficiently. In particular, it is hard 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.

We performed a further analysis of the mutation tools in Table XIV. We find that most tools adopted some 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

Copyright © 2010 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Cost Reduction Technique	Total
n/a	131
fixed number	28
weak mutation	15
mutant sample	11
selection strategy	8
higher-order	1

Table XV. Cost Reduction Summary

thing to highlight is that MiLu used a special test harness to reduce runtime [184]. This test harness is created containing all the test cases and settings for running each mutant. Therefore, only the test harness needs to be executed while each mutant runs as an internal function call during the testing process.

Selective mutation is also widely applied in almost all the existing mutation testing tools (as shown in Table XII). This brings us to another issue: *is the selected subset of mutation operators 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.'s five sufficient Fortran mutation operators [30], and Siami et al.'s 28 sufficient C mutation operators [32]. However, most of the articles are not supported by empirical or theoretical studies that show a certain subset of mutation operators can represent the whole mutation operator set. As far as we know, most studies on selective mutation are merely based on Fortran [27,30,31] and C [32,193–195] programs. Thereby, we recommend more empirical studies on selective mutation in programming languages other than Fortran).

Compared to Jia and Harman's literature review [1], most of the cost reduction techniques they surveyed have been adopted in our collection. Runtime optimisation techniques which they summarised, e.g., interpreter-based technique [17], compiler-based approach [18] and bytecode translation [145] have been widely adopted in existing mutation testing tools. However, the articles we have reviewed did not apply Firm Mutation [196] and advanced platforms support for mutation testing, such as SIMD [197] and MIMD machines [198]. For Firm Mutation, there is no publicly available tool to support this method; thus, other researchers cannot adopt this approach conveniently. As for advanced platforms, these machines are not easy for other researchers in the mutation testing field to obtain. One exception is the cost reduction technique with a *fixed number*, which was not covered by Jia and Harman [1]. As mentioned earlier, the fixed-number-of-mutants technique is different from mutant sampling as the former selects a subset of mutants based on a fixed number rather than a ratio. We speculated the reason why Jia and Harm [1] did not include this method is that reduction based on a fixed number is too simple to be considered as a real technique for cost reduction in mutation testing. However, such a technique is surprisingly popular among the applications of mutation testing.

Summary. Based on the above discussion, we infer that the problem of the high computational cost of mutation testing can be adequately controlled using the state-of-art reduction techniques, especially selective mutation and runtime optimisation. Selective mutation is the most widely used

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

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 for other programming languages instead of C and Fortran. Therefore, one recommendation is to conduct more empirical studies on selective mutation in various programming languages. Random selection of the mutants based on a fixed number (28 papers) is the most popular technique used to reduce the computational cost. The other popular techniques are weak mutation and mutant sampling. Besides, a high percentage of the papers (68.6%) did not report any reduction techniques used to cope with computational cost when applying mutation testing; this again should serve as a reminder for the research community to pay more attention to *properly* reporting mutation testing in testing experiments.

4.2.5. RQ2.5: What subjects are being used in the experiments (regarding programming language, size, and data availability)?

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.

Table XVI shows the distribution of the programming languages. We can see that Java and C dominate the application domain (66.0%, 126 instances). While JavaScript is an extensively used language in the web application domain, we only found three research studies in our datasets that applied mutation testing for this programming language. 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 by the fact that existing, well-defined software repositories such as `SIR` [97], `SF100` [98] 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, which are often composed out of a combination of JavaScript, HTML, CSS, etc; thus, this 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 size of the subject programs, in addition to our collection, we also summarise the data presented in Jia and Harman's survey [1] in Table XVII. In Jia and Harman [1]'s survey, they summarised all the programs used in empirical studies related to mutation testing including the program size and the total number of uses. In Table XVII, to summarise the data from Jia and Harman, we first categorise the programs into five classes (as shown in the first column) according to their size in LOC, and then add up the total number of usages (as shown in the third column). It is important to note that the program data in Jia and Harman's survey [1] is different from ours: they collected the programs from empirical studies which aimed to evaluate the effectiveness of mutation testing, e.g., Mathur and Wong compared data flow criteria with mutation testing [21]. These studies are not part of our research scope: we focus on the application perspective while these works target the development of mutation testing approaches. The purpose of the comparison of Jia and Harman's data is to investigate the difference between two perspectives of mutation testing, i.e.,

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

the *development* and the *application* perspective. Moreover, they listed all the possible programs used in empirical studies while we only collected the *maximum* size of the programs.

As for our collection, studies involving preliminary ($<$100 LOC), small (100$\sim$10K LOC) subjects or studies with no information about programs size ("n/a" instances in Table XVII) represent 80.6% (154 instances) of papers in our collection. Among these "n/a" cases, some papers (e.g., Xie et al. [177]) did not provide any information about the subject size (LOC), and a few cases (e.g., Baudry et al [107]) report outdated links. This high percentage of preliminary, small and "n/a" subjects indicates that mutation testing is rarely applied to programs whose size is larger than 10K LOC. We did find that only 35 studies use medium size subjects, which corresponds to 29.9%[‡] of papers. Compared to Jia and Harman's data [1], preliminary programs account for 63.4%[§] of the study set, which is much higher than in our study. There are two possible causes for this finding. The one is that we only consider the maximum size of the programs in each study, we could subtract many potential cases that used preliminary programs from our results. The other is that a considerable number of the empirical studies on mutation testing that Jia and Harman reviewed date back to the 1990s. These early-stage research works on mutation testing mainly involving preliminary subjects. Also, we witness an increasing trend of medium and large size subject systems being used in studies on mutation testing; this shows the full potential of mutation testing in large-scale applications.

With regard to data availability, we observe the following: 49.7% of the studies provide open access to their experiments. Some studies, such as Staats et al. [199], used close-sourced programs from industry. There are also a few cases for which the source links provided by the authors are not accessible anymore, e.g., Jolly et al. [200]. We also found that several cases used open-source software corpora, such as SF110, but they only used a sample of the corpus (e.g., Rojas et al. [201]) without providing sample information. The others did not provide information about sources in their paper, e.g., Kanewala and Bieman [202].

Together with 6 instances of "n/a" in Table XVI and 74 in Table XVII (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. Also, basic information on the programming language, size and subject should also be clearly specified in the articles to ensure replicability.

Summary. For the subjects used in the experiments in our survey, we discussed three aspects: programming language, size of subject projects 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 recommendation labelled as **R7** in Section 4.4).

As for the maximum size of subject programs, small to medium scale projects (100$\sim$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

[‡]Notice that we did not consider the number of "n/a" value when calculating the percentage.(191 $-$ 74 $=$ 117).
[§]Similarly, we removed the cases of "n/a" here when calculating the percentage.($402 - 28 = 374$).

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Language	Total
Java	92
C	34
Lustre/Simulink	8
C#	6
Fortran	6
n/a	6
C++	5
SQL	5
Eiffel	3
Spreadsheet	3
AspectJ	3
JavaScript	3
Enterprise JavaBeans application	2
C/C++	2
Ada	1
Kermeta	1
Delphi	1
ISO C++ grammar	1
PLC	1
Sulu	1
XACML	1
XML	1
HLPSL	1
PHP	1
other specification languages	10

Table XVI. Programming Language Summary

Subject Size	Our collection	Jia and Harman
n/a	74	28
small (100∼10K LOC)	70	128
medium (10K∼1M LOC)	35	9
preliminary (<100 LOC)	10	237
large (>1M LOC)	2	0

Note: The results of our collection are based on the *maximum* size of the programs used in each study while Jia and Harman's is based on *all* the programs.

Table XVII. Subject Size Summary

Data Availability	Total
no	96
yes	95

Table XVIII. Data Availability Summary

more research on large-scale systems to further explore scalability issues (see the recommendation labelled as **R8** in Section 4.4).

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

The third aspect we consider is data availability. Only 49.7% 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.

4.3. Summary of Research Questions

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

RQ1: How is mutation testing used in quality assurance processes? Mutation testing is mainly used as a fault-based evaluation method (70.2%) in different quality assurance processes. It assesses the fault detection capability of various testing techniques through the mutation score or the number of killed mutants. Adopting mutation testing to improve other quality assurance processes as a guide was first proposed by DeMillo 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 quality assurance processes, like test data generation (36 instances), test case prioritisation (6 cases) and test strategy evaluation (6 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 (72.0%), with only a small number of studies featuring higher-level testing showing the overall benefit of mutation testing. As a result, we conclude that the current state of the application of mutation testing is still rather limited.

RQ2: How are empirical studies related to mutation testing designed and reported? First of all, for the mutation testing tools and mutation operators used in literature, we found that 47.6% of the articles adopted existing (open-access) mutation testing tools, such as MuJava for Java and Proteum for C. In contrast, we did encounter a few cases (27 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 generalised classification of mutation operators as shown in Listing 1. 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, although we should mention that we did not focus on built-in reduction techniques.

The findings above 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 [2, 143]. 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 the selective mutation on mutation testing; this suggests that

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 (see Section 4.1.2)	84
mutation tool source (see Section 4.2.1)	92
mutation operators (see Section 4.2.2)	72
equivalent mutant problem (see Section 4.2.3)	108
reduction problem (see Section 4.2.4)	131
subject program source (see Section 4.2.5)	96

Table XIX. Poorly-specified aspects in empirical studies

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 certain type of mutant is when testing real software, we can then design the mutation operators accordingly, such as Just et al. [203].

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.

From the statistics of the collection, we found that 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 XIX. 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 of the subject program (see the ninth recommendation labelled as **R9** in Section 4.4).

4.4. Recommendation for Future Research

In this section, we will summarise the recommendations for future research based on the insights obtained for the two main research questions (see Sections 4.1 through 4.3). We propose nine recommendations for future research:

- **R1: Mutation testing cannot only be used as *where*-to-check constraints but also to suggest *what* to check to improve test code quality.**
  As shown in Table IV 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 the 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.**

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

When looking at the evaluation types in Table V in Section 4.1.1, we observe that 75.4% of the mutation-based testing techniques still adopt mutation faults to assess their effectiveness. 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 efficiently.

- **R3: Study the higher-level application of mutation testing.**
  In Section 4.1.2 we observed that mutation testing seems to mainly target unit-level testing, accounting for 72.0% 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 should also explore whether the conclusions based on unit-level mutation testing still apply to higher-level mutation testing. A pertinent question in this area could be, for example, 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 VIII in Section 4.2.1, 50.3% of the articles adopted the existing tools which are open-source, while we also found 27 instances of researchers implementing their own tool or seeding the mutants by hand. Furthermore, in Table XI and Table XII, we can see certain existing mutation testing tools lack certain mutation operators. These findings imply that existing mutation testing tools cannot 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 easily added [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 non-equivalent) 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 an auxiliary tool for the equivalent mutant problem.
- **R6: More empirical studies on the selective mutation method can pay attention to programming languages other than Fortran 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 Fortran [27, 30, 31] and C [32, 193–195]. Selective mutation requires more empirical studies to explore whether a certain subset of mutation operators can be applied in different programming languages.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

- **R7: More attention should be given to other programming languages, especially web applications and functional programming projects.**
  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 scarce. JavaScript, as one of the most popular languages for developing web applications, calls for more attention from researchers. In the meanwhile, functional programming languages, such as Lisp and Haskell, are still playing an inevitable role in the implementation of programs; they thus deserve more focus in future studies.
- **R8: Application of mutation testing in large-scale systems to explore scalability issues.**
  From Table XVII 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). 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 XIX in Section 4.4, we remember that a considerable amount of papers inadequately reported on mutation testing. 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 research. 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 [7], Wohlin [75], and Brereton et al. [72]) 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 [77]).

In this section, we describe the main threats to the 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., the 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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

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 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 listed in Table II 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 papers. Moreover, we also ran two sanity checks (as mentioned in Section 3.2) to examine the completeness of our study collection, and recorded the dataset in each step for further validation.

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 (except the International Workshop on Mutation Analysis), 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. [204]. The main difference between mutation testing and error seeding is the way how to introduce defects in the program [205]: 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 make no further interpretation of the information extracted from the papers in the second pilot run of validation. In particular, if a detail is not specified in a paper, we mark it 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.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

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 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 literature reviews that surveyed the main development of mutation testing, and that did not specifically go into how mutation testing is applied (e.g., [1, 2, 6]). We have characterised the studies that we have found on the basis of seven facets: (1) the role that mutation testing has in quality assurance processes; (2) the quality assurance processes (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 subject 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 quality assurance processes?* and **RQ2** *How are empirical studies related to mutation testing designed and reported?*

Figure 1 shows our main procedures to conduct this systematic literature review. 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 quality assurance processes 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 191 papers from 22 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 III. 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. We then presented the characterisation data of all the surveyed papers in our GitHub repository [77]. 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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

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 testing in the future by asking such questions as: what else can we mutate? (Sections 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 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 (Sections 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 191 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 the *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 valuable 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. 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.
7. Kitchenham B. Guidelines for performing systematic literature reviews in software engineering. *Technical Report EBSE-2007-01*, 2007.
8. Offutt A. The coupling effect: fact or fiction. *ACM SIGSOFT Software Engineering Notes*, vol. 14, ACM, 1989; 131–140.
9. Offutt AJ. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1992; **1**(1):5–20.
10. Wah K. Fault coupling in finite bijective functions. *Software Testing, Verification and Reliability* 1995; **5**(1):3–47.
11. Wah K. A theoretical study of fault coupling. *Software testing, verification and reliability* 2000; **10**(1):3–45.
12. Wah KHT. An analysis of the coupling effect i: single test data. *Science of Computer Programming* 2003; **48**(2):119–161.
13. Kapoor K. Formal analysis of coupling hypothesis for logical faults. *Innovations in Systems and Software Engineering* 2006; **2**(2):80–87.
14. Budd TA, Angluin D. Two notions of correctness and their relation to testing. *Acta Informatica* 1982; **18**(1):31–45.
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.

Prepared using stvrauth.cls
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. *7th International Workshop on Mutation Analysis (MUTATION'12)*, 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. Woodward MR. Mutation testing-an evolving technique. *Software Testing for Critical Systems, IEE Colloquium on*, IET, 1990; 3–1.
71. Hanh LTM, Binh NT, Tung KT. Survey on mutation-based test data generation. *International Journal of Electrical and Computer Engineering* 2015; **5**(5).
72. 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.
73. 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].
74. 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.
75. 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.
76. Knauth T, Fetzer C, Felber P. Assertion-driven development: Assessing the quality of contracts using meta-mutations. *The 4th International Workshop on Mutation Analysis (Mutation 2009)*, IEEE, 2009; 182–191.
77. Mutation testing literature survey metadata. https://zenodo.org/badge/latestdoi/95541866. [Online; accessed 17-August-2017].
78. Graham D, Van Veenendaal E, Evans I. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.
79. van Deursen A. Software Testing in 2048. https://speakerdeck.com/avandeursen/software-testing-in-2048 1 2016. [Online; accessed 13-Sep-2016].
80. 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.
81. Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, 2011; 416–419.
82. Jayaraman K, Harvison D, Ganesh V, Kiezun A. jfuzz: A concolic whitebox fuzzer for java 2009; .
83. 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.
84. Le Goues C, Nguyen T, Forrest S, Weimer W. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 2012; **38**(1):54–72.
85. Available mutation operations (PIT). http://pitest.org/quickstart/mutators/. [Online; accessed 10-August-2016].

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

DOI: 10.1002/stvr

86. Ma YS, Offutt J. Description of method-level mutation operators for java. *Electronics and Telecommunications Research Institute, Korea, Tech. Rep* 2005; .
87. Ma YS, Offutt J. Description of class mutation mutation operators for java. *Electronics and Telecommunications Research Institute, Korea, Tech. Rep* 2005; .
88. 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.
89. 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.
90. 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.
91. Xu D, Ding J. Prioritizing state-based aspect tests. *2010 Third International Conference on Software Testing, Verification and Validation*, IEEE, 2010; 265–274.
92. 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.
93. 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.
94. 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.
95. 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.
96. 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.
97. 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.
98. Fraser G, Arcuri A. Sound empirical evidence in software testing. *Proceedings of the 34th International Conference on Software Engineering*, IEEE Press, 2012; 178–188.
99. Antoniol G, Briand LC, Penta MD, Labiche Y. A case study using the round-trip strategy for state-based class testing. *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, IEEE, 2002; 269–279.
100. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on* 2006; **32**(8):608–624.
101. Belli F, Beyazit M, Takagi T, Furukawa Z. Mutation testing of" go-back" functions based on pushdown automata. *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, IEEE, 2011; 249–258.
102. Czemerinski H, Braberman V, Uchitel S. Behaviour abstraction coverage as black-box adequacy criteria. *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, IEEE, 2013; 222–231.
103. 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.
104. Androutsopoulos K, Clark D, Dan H, Hierons RM, Harman M. An analysis of the relationship between conditional entropy and failed error propagation in software testing. *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014; 573–583.
105. Czemerinski H, Braberman V, Uchitel S. Behaviour abstraction adequacy criteria for api call protocol testing. *Software Testing, Verification and Reliability* 2015; .
106. 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.
107. 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.
108. 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.
109. 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.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

110. Smith BH, Williams L. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering* 2009; **14**(3):341–369.
111. 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.
112. 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.
113. 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.
114. Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on* 2012; **38**(2):278–292.
115. 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.
116. Gay G, Staats M, Whalen M, Heimdahl MP. Automated oracle data selection support. *IEEE Transactions on Software Engineering* 2015; **41**(11):1119–1137.
117. 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.
118. 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.
119. 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.
120. 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.
121. 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.
122. Aichernig BK, Brandl H, Jöbstl E, Krenn W, Schlick R, Tiran S. Killing strategies for model-based mutation testing. *Software Testing, Verification and Reliability* 2015; **25**(8):716–748.
123. Jee E, Shin D, Cha S, Lee JS, Bae DH. Automated test case generation for fbd programs implementing reactor protection system software. *Software Testing, Verification and Reliability* 2014; **24**(8):608–628.
124. 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.
125. Li P, Huynh T, Reformat M, Miller J. A practical approach to testing gui systems. *Empirical Software Engineering* 2007; **12**(4):331–357.
126. 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.
127. 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.
128. Delamaro ME, Maidonado J, Mathur AP. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering* 2001; **27**(3):228–247.
129. Mateo PR, Usaola MP, Offutt J. Mutation at the multi-class and system levels. *Science of Computer Programming* 2013; **78**(4):364–387.
130. Flores A, Polo M. Testing-based process for component substitutability. *Software Testing, Verification and Reliability* 2012; **22**(8):529–561.
131. Vincenzi AMR, Maldonado JC, Barbosa EF, Delamaro ME. Unit and integration testing strategies for c programs using mutation. *Software Testing, Verification and Reliability* 2001; **11**(4):249–268.
132. Flores A, Usaola MP. Testing-based assessment process for upgrading component systems. *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, IEEE, 2008; 327–336.
133. 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.
134. 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.
135. Belli F, Beyazıt M. Exploiting model morphology for event-based testing. *IEEE Transactions on Software Engineering* 2015; **41**(2):113–134.
136. 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.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

137. 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.
138. 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.
139. 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.
140. 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.
141. Briand LC, Labiche Y, Lin Q. Improving statechart testing criteria using data flow information. *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, IEEE, 2005; 10–pp.
142. 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.
143. Kintis M, Papadakis M, Papadopoulos A, Valvis E, Malevris N. Analysing and comparing the effectiveness of mutation testing tools: A manual study. *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, IEEE, 2016; 147–156.
144. Papadakis M, Malevris N. Mutation based test case generation via a path selection strategy. *Information and Software Technology* 2012; **54**(9):915–932.
145. Ma YS, Offutt J, Kwon YR. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability* 2005; **15**(2):97–133.
146. 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.
147. Delamaro ME, Maldonado JC, Vincenzi AMR. Proteum/im 2.0: An integrated mutation testing environment. *Mutation testing for the new century*. Springer, 2001; 91–101.
148. Briand LC, Labiche Y, Wang Y. Using simulation to empirically investigate test coverage criteria based on statechart. *Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, 2004; 86–95.
149. DeMillo RA, Offutt AJ. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1993; **2**(2):109–127.
150. Shi A, Yung T, Gyori A, Marinov D. Comparing and combining test-suite reduction and regression test selection. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 2015; 237–247.
151. Bieman JM, Ghosh S, Alexander RT. A technique for mutation of java objects. *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, IEEE, 2001; 337–340.
152. Alexander RT, Bieman JM, Ghosh S, Ji B. Mutation of java objects. *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, IEEE, 2002; 341–351.
153. Vilela P, Machado M, Wong W. Testing for security vulnerabilities in software. *Software Engineering and Applications* 2002; .
154. Shahriar H, Zulkernine M. Mutation-based testing of format string bugs. *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, IEEE, 2008; 229–238.
155. Derezinska A. Quality assessment of mutation operators dedicated for c# programs. *Quality Software, 2006. QSIC 2006. Sixth International Conference on*, IEEE, 2006; 227–234.
156. Shahriar H, Zulkernine M. Music: Mutation-based sql injection vulnerability checking. *Quality Software, 2008. QSIC'08. The Eighth International Conference on*, IEEE, 2008; 77–86.
157. Milani Fard A, Mirzaaghaei M, Mesbah A. Leveraging existing tests in automated test generation for web applications. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, ACM, 2014; 67–78.
158. GitHub Repository for MuJava. https://github.com/jeffoffutt/muJava. [Online; accessed 18-April-2018].
159. GitHub Repository for PIT. https://github.com/hcoles/pitest. [Online; accessed 18-April-2018].
160. GitHub Repository for Javalanche. https://github.com/david-schuler/javalanche. [Online; accessed 18-April-2018].
161. The Major mutation framework. http://mutation-testing.org/doc/major.pdf. [Online; accessed 21-August-2017].
162. Major v1.3.2. http://mutation-testing.org/downloads/files/major-1.3.2_jre7.zip. [Online; accessed 18-April-2018].

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

163. Jumble. http://jumble.sourceforge.net/mutations.html. [Online; accessed 21-August-2017].
164. Sofya. http://sofya.unl.edu/doc/manual/user/mutator.html. [Online; accessed 21-August-2017].
165. Moore I. Jester-a junit test tester. *Proc. of 2nd XP* 2001; :84–87.
166. GitHub Repository for Proteum. https://github.com/magsilva/proteum. [Online; accessed 18-April-2018].
167. GitHub Repository for Milu. https://github.com/yuejia/Milu/tree/develop/src/mutators. [Online; accessed 21-August-2017].
168. Clark JA, Dan H, Hierons RM. Semantic mutation testing. *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, IEEE, 2010; 100–109.
169. GitHub Repository for Proteum/IM. https://github.com/jacksonpradolima/proteumIM2.0. [Online; accessed 18-April-2018].
170. Wright C. Mutation analysis of relational database schemas. PhD Thesis, University of Sheffield 2015.
171. GenMutants. http://pexase.codeplex.com/SourceControl/latest#Projects/GenMutants/GenMutants/Program.cs. [Online; accessed 21-August-2017].
172. PexMutator. http://pexase.codeplex.com/SourceControl/latest#Projects/PexMutator/PexMutator/Program.cs. [Online; accessed 21-August-2017].
173. 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.
174. 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.
175. Gay G, Staats M, Whalen M, Heimdahl MP. The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on* 2015; **41**(8):803–819.
176. Liu MH, Gao YF, Shan JH, Liu JH, Zhang L, Sun JS. An approach to test data generation for killing multiple mutants. *Software Maintenance, 2006. ICSM 2006. IEEE International Conference on*, IEEE, 2006; 113–122.
177. Xie X, Ho JW, Murphy C, Kaiser G, Xu B, Chen TY. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 2011; **84**(4):544–558.
178. Fang C, Chen Z, Wu K, Zhao Z. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal* 2014; **22**(2):335–361.
179. Offutt AJ, Liu S. Generating test data from sofl specifications. *Journal of Systems and Software* 1999; **49**(1):49–62.
180. Mutation testing systems for Java compared. http://pitest.org/java_mutation_testing_systems/. [Online; accessed 25-August-2016].
181. 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.
182. Motycka W. Installation Instructions for Sofya. http://sofya.unl.edu/doc/manual/installation.html 7 2013. [Online; accessed 25-August-2016].
183. Papadakis M, Jia Y, Harman M, Le Traon Y. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, IEEE, 2015; 936–946.
184. 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.
185. 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.
186. Offutt AJ, Craft WM. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability* 1994; **4**(3):131–154.
187. Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability* 1997; **7**(3):165–192.
188. Baker R, Habli I. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering* 2013; **39**(6):787–805.
189. Staats M, Loyola P, Rothermel G. Oracle-centric test case prioritization. *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, IEEE, 2012; 311–320.
190. Vivanti M, Mis A, Gorla A, Fraser G. Search-based data-flow test generation. *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, IEEE, 2013; 370–379.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

191. Arcuri A, Briand L. Adaptive random testing: An illusion of effectiveness? *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ACM, 2011; 265–275.
192. Stephan M, Cordy JR. Model clone detector evaluation using mutation analysis. *ICSME*, 2014; 633–638.
193. Barbosa EF, Maldonado JC, Vincenzi AMR. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability* 2001; **11**(2):113–136.
194. Namin AS, Andrews JH. On sufficiency of mutants. *Software Engineering-Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, IEEE, 2007; 73–74.
195. Kurtz B, Ammann P, Offutt J, Delamaro ME, Kurtz M, Gökçe N. Analyzing the validity of selective mutation with dominator mutants. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2016; 571–582.
196. Jackson D, Woodward MR. Parallel firm mutation of java programs. *Mutation testing for the new century*. Springer, 2001; 55–61.
197. Krauser EW, Mathur AP, Rego VJ. High performance software testing on simd machines. *IEEE Transactions on Software Engineering* 1991; **17**(5):403–423.
198. Offutt AJ, Pargas RP, Fichter SV, Khambekar PK. Mutation testing of software using a mimd computer. *in 1992 International Conference on Parallel Processing*, Citeseer, 1992.
199. Staats M, Whalen MW, Heimdahl MP. Better testing through oracle selection (nier track). *Proceedings of the 33rd International Conference on Software Engineering*, ACM, 2011; 892–895.
200. Jolly SA, Garousi V, Eskandar MM. Automated unit testing of a scada control software: an industrial case study based on action research. *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, IEEE, 2012; 400–409.
201. Rojas JM, Campos J, Vivanti M, Fraser G, Arcuri A. Combining multiple coverage criteria in search-based unit test generation. *International Symposium on Search Based Software Engineering*, Springer, 2015; 93–108.
202. Kanewala U, Bieman JM. Using machine learning techniques to detect metamorphic relations for programs without test oracles. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2013; 1–10.
203. Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing? *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014; 654–665.
204. 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.
205. 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