Mutation Testing

Robert M. Hierons*[1], Mercedes G. Merayo[1,2], and Manuel Núñez[2]

1: Department of Information Systems and Computing, Brunel University

Uxbridge, Middlesex, UB8 3PH United Kingdom

2 : Departamento Sistemas Informáticos y Computación

Universidad Complutense de Madrid, Madrid, Spain

February 20, 2008

Abstract

Traditionally in mutation testing we produce a set of mutants, which are variants of the system under test. The adequacy of a test suite is then judged by determining how many of the mutants it distinguishes from the original system. The aim is to produce mutants that are similar to real faults, in which case the ability of a test suite to distinguish between a program and its mutants should relate to its ability to distinguish between a correct program and a faulty program. This article describes mutation testing, the directions in which it has developed, and some of the main challenges.

Keywords: Testing, Mutation testing, test adequacy criteria, test generation, automation.

1 Introduction

Testing is one of the most important approaches to ensuring the quality of software and there has thus been a significant amount of research regarding test criteria, which are used to state when a test suite is considered to be sufficient, and techniques for finding a test suite that satisfies a given criterion. Most test criteria require that certain 'aspects' of the system under test (SUT) are exercised in testing, or covered, and these criteria are thus called *coverage criteria*. Such a criterion is white-box if it refers to the structure of the code of the SUT and is black-box if it relates to the structure of a model of the SUT. Popular examples include statement coverage and branch coverage in white-box testing and state coverage and transition coverage when testing from state machines.

While many of the most popular test criteria are coverage criteria, currently there is a lack of information regarding how coverage relates to fault detection. Mutation testing takes a very different approach and instead aims to produce test suites that distinguish between the given SUT p and programs that are similar to p and these are called *mutants*. The belief is that if a test suite is good at distinguishing between p and mutants and p is faulty then it will be good at distinguishing between p and a correct program. As a result, the test suite should be good at determining whether p is correct.

Mutation testing originated in the 1970s and initially considered imperative programming languages and, in particular, FORTRAN [5, 23]. More recent work has investigated other programming languages including object-oriented languages [16, 20]. There has also been interest in mutating a model or specification of the SUT rather than the source code. However, in each case the underlying principle is the same: we produce mutants and judge the adequacy of a test suite by determining how many of these mutants it distinguishes from the original artifact.

Since mutation testing can estimate the effectiveness of a test suite it has two additional uses. The first is that we can aim to produce a test suite that is effective in distinguishing an artefact from its mutants and thus mutation testing can be used to drive test generation. In addition, mutation testing can be used to estimate the quality of the test suites produced by other test technique and currently this is probably the main use of mutation testing.

Mutation testing has been a very active research field during the last years. In fact, mutation testing papers regularly appear not only in venues exclusively devoted to testing but are also published in software engineering conferences. In addition, there exists a workshop where the most recent advances on mutation testing are usually presented and discussed (see `http://cs.gmu.edu/mutation2007/` for the 2007 edition of the event).

This article reviews previous work in mutation testing and remaining challenges and is structured as follows. Section 2 reviews the traditional approach to mutation testing devised for procedural programming languages. Section 3

then describes some variants on this approach. Section 4 describes two of the main challenges that have prevented mutation testing from becoming an important technique used in practice and Section 5 then discusses recent work that has looked at mutating models rather than programs. Finally, conclusions are drawn in Section 6.

2 Traditional Mutation Testing

The initial work on mutation testing considered only the source code, or program, p and mutants are produced by mutating p. In practice, each mutant is produced by applying a *mutation operator* to the source code, a mutation operator making a small syntactic change such as replacing $>$ by $>=$. A mutant is a *first order mutant* if it can be produced from the program p by one application of a mutation operator and otherwise it is a *higher order mutant*.

The first mutation testing tool was *Mothra* and operated on FORTRAN programs [3, 15]. As a result, Mothra and its mutation operators have acted as a starting point for most work on mutation testing. The 22 standard mutation operators used in Mothra are listed in Figure 1. It is clear that all of these mutation operators involve small syntactic changes and most are relevant to many programming languages.

In testing we apply *test cases*, each test case usually at least containing the test input and expected outcome. A set of test cases is called a *test suite*. If we have a set M of mutants then in order to estimate the quality of a test suite $\mathcal{T}$

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

Figure 1: The Mothra Mutation Operators

we apply the test cases from $\mathcal{T}$ to each mutant in M. We say that a test case $t \in \mathcal{T}$, and also $\mathcal{T}$, *kills* the mutant $m \in M$ if m and p produce different output when given the input from t. If this is the case then the test case t is *effective* in distinguishing p and m. If a mutant has not been killed then it is *live*.

Let us consider, for example the following fragment of code, based on an example from [4], that takes two numbers as input and returns one plus the maximum of these numbers.

```
input(x,y);
z:=x+1;
if (y>x) then z:=y+1;
output(z);
```

Let us suppose that we mutate the statement `z:=x+1;` to `z:=y+1;`. Then the resultant mutant is:

```
input(x,y);
z:=y+1;
if (y>x) then z:=y+1;
output(z);
```

It is clear that for any input in which y is larger than or equal to x the two programs will produce the same output. For example, if we execute these with the input $(1, 5)$ the two programs output 6 and so the mutant is still live. However, if we test them with the input $(4, 2)$ we find that the original program produces output 5 while the mutant produces output 3 and so is killed.

Given a set M of mutants and a test suite $\mathcal{T}$ that kills the set $M' \subseteq M$ of

mutants, we obtain the following measure that estimates the effectiveness of $\mathcal{T}$ for testing p.

$$E(\mathcal{T}, M, p) = \frac{|M'|}{|M|}$$

There are at least two reasons for a test suite failing to kill a mutant $m \in M$. First, it may be that the test suite simply does not contain any of the test cases that kill m. However, it is possible that there are no such test cases: m and p may be functionally equivalent. If m is functionally equivalent to p then we say that m is an *equivalent mutant*. Now consider the mutant produced from the original program given above by replacing $>$ by $>=$ [4]. This produces the following mutant.

```
input(x,y);

z:=x+1;

if (y>=x) then z:=y+1;

output(z);
```

It is straightforward to show that this is an equivalent mutant since the behaviour of the two programs can only differ if x and y have the same value and in this case the two assignments to z assign the same value.

Since it is not possible to kill an equivalent mutant, such mutants should not be included in any measure that estimates test effectiveness. Thus, given a set M of mutants, a test suite $\mathcal{T}$ that kills the set $M' \subseteq M$ of mutants, and the subset M_e of equivalent mutants from M we obtain the following measure that estimates the effectiveness of $\mathcal{T}$ for testing p.

$$E_f(\mathcal{T}, M, p) = \frac{|M'|}{|M \setminus M_e|}$$

Clearly, E_f is a more appropriate measure of test effectiveness than E. However, in order to calculate E_f we need to determine which mutants from $M \setminus M'$ are equivalent mutants and this problem is uncomputable. As a result, we cannot always compute E_f. In addition, any attempt to produce a good approximation can lead to significant manual effort and this can make mutation testing an expensive technique to use. In Section 4 we describe some approaches that aim to reduce the equivalent mutant problem.

The justification for using mutation testing relies on a test suite that detects mutants being likely to find real faults. As a result, the mutants should differ from the original in a manner that is consistent with real faults. The mutation operators introduce small syntactic changes into a program and the use of such operators is explained in terms of the *competent programmer hypothesis* that states that programmers usually make small mistakes [5].

It is normal to only use first-order mutants and this helps limit the number of mutants produced. The use of first-order mutants is justified by the *coupling hypothesis*, which says that complex faults are coupled in the sense that a test suite that detects simple faults is also likely to detect more complex faults [5]. The coupling hypothesis is supported by experiments that found that test suites that killed a high proportion of first-order mutants also killed a high proportion of second order and third order mutants [17]. A theoretical study, which investigated the potential for coupling between two functions that have

finite domains, provide additional evidence [24].

One of the potential strengths of mutation testing is that for many white-box coverage criteria it is possible to apply mutation testing in a manner than ensures that if a test suite kills all of the mutants then it satisfies the coverage criterion (see, for example, [23]). Let us suppose, for example, that we wish to execute all of the statements in a program p. Then for each statement s of p, we can produce a mutant m_s of p by mutating s. If a test case kills m_s then it must execute s when applied to p and thus any test suite that kills all of the mutants created in this way must also execute all statements of p. Similarly, given a predicate P in p and the branch under which P takes the value $X \in \{True, False\}$, we can produce a mutant $m_{(p,X)}$ in which P is replaced by $\neg X$. If a test input x kills $m_{(p,X)}$ then when p is executed with x it must take a different branch from $m_{(p,X)}$ and so must take the X branch at P and so cover this branch. Thus, if a test suite kills all such mutants for a program p then it covers all branches of p. So, a tool that generates test cases that kill mutants can also be used to produce test cases that cover parts of a program.

In practice, one of the main uses of mutation testing is in the evaluation of other test techniques. Specifically, in order to evaluate a test technique we can produce test suites using this technique and estimate their effectiveness by determining what proportion of mutants they kill. Naturally, this determines what proportion of 'artificial' faults are found, rather than real faults. However, the results of recent experiments using programs with sets of real faults suggest

that it is appropriate to estimate the effectiveness of a test suite using mutants [1].

Two studies compared mutation testing to the all-uses data flow criterion [6, 19]. While both studies used quite small programs, they used different mutation operators and programming languages. Both of them found that mutation testing is slightly more effective than all-uses data flow testing and that most test suites that are adequate under the mutation testing criterion are also adequate under the all-uses criterion. In addition, it was less likely that a test suite that was adequate under the all-uses criterion was adequate under the mutation testing criterion. However, the differences in performance were variable and relatively small. Interestingly, experiments found that when test generation was based on a random selection of only 10% of each type of mutant, produced using the Mothra operators, then the resulting test suite had an average all-uses score of over 96% [25]. The results were similar when using just the mutants produced with the ABS and ROR operators. Average all-uses scores of over 99% were obtained by using only 25% of the mutants. If all the mutants were used then the average all-uses score was 100% for three of the four programs and over 99% for the other. However, again the experiments were performed on small programs.

While the use of first-order mutants limits the number of mutants produced, it is still possible to produce a large number of mutants even for a small program. This can lead to significant practical issues since it is necessary to generate, store, and execute these mutants. Two approaches to reducing this problem, weak

mutation and selective mutation, are described in Sections 3 and 4 respectively.

3 Variants on Traditional Mutation Testing

In this section we review some of the variations on classical mutation testing that have appeared in the literature to deal with the challenges that the original framework presented.

3.1 What it means to kill a mutant

Traditionally, in order to kill a mutant m of a program p we are required to execute it on an input x such that p and m produce different output when run with x. Assuming that we have created m by changing one part s of p only, according to [4], this requires the following:

1. When executed with x, the program p follows a path that includes s;
2. The change at s leads to a different program state (or a different output at this point); and
3. The difference in program state leads to a difference in output.

Thus, while it is necessary to execute the point s that was changed this is not sufficient to kill the mutant. In addition, it has been observed that sometimes it is difficult to propagate changes in the program state to give a different output and this makes it more difficult to apply mutation testing in test suite generation since there can be mutants for which it is difficult to find corresponding test

data. As a result, the concept of *weak mutation* was proposed [12]. Under weak mutation, in order to kill a mutant it is sufficient that when running the original program p and the mutant m with the test case we get a difference in program state or a different output. Weak mutation thus requires that a test case leads to a 'different behaviour' for the original program and the mutant but does not require that this difference is observed through the output. In order to distinguish it from weak mutation, mutation testing in which we require an observable difference in behaviour is normal called *strong mutation testing*. In essence, weak mutation testing and strong mutation testing differ in the point where we make observations: in weak mutation testing this is after the point mutated and in strong mutation testing it is the end of execution.

The weakening of the requirement that we observe a difference between the observed behaviour of the mutant and the original program seems to move weak mutation testing away from the original motivation — measuring test effectiveness by estimating how good a test suite is at distinguishing programs. However, if we consider a test input x and a program p, when executed p with x we can store the execution trace and thus the values of the program variables when each statement is met. Having done this, for each statement s we can determine which mutations of p would have been killed by executing p with x and so there is no longer the need to generate, store and execute many mutants. Thus, weak mutation provides a major practical benefit beyond typically being easier to satisfy.

It might initially seem that weak mutation testing must always be 'weaker' than strong mutation testing in the sense that any test suite that kills all mutants under strong mutation testing also kills all mutants under weak mutation testing. However, this is not the case if we eliminate equivalent mutants from our measurements, since a mutant could be an equivalent mutant in strong mutation testing but not in weak mutation testing. Let us consider, for example, the following simple program p and let us suppose that we have only one mutant and this is formed by replacing '+' by '-' in the second line.

```
input(x,y,z);
x:=y+z;
x:=1;
output(x);
```

For all input the original program and the mutant produce the same output and so under strong mutation testing we have that m is an equivalent mutant. However, there are inputs under which m and p have different internal states after the point mutated and so m is not an equivalent mutant under weak mutation testing. Thus, a test suite that kills all non-equivalent mutants from a set M of mutants including this mutant, under strong mutation, does not necessarily kill all the non-equivalent elements of M under weak mutation.

Another approach is given by *firm mutation* [26] where we mutate the program p at a point s to form a mutant m and then compare the states of p and m at some point s'. As a result, firm mutation testing is a generalization of weak mutation testing: weak mutation testing is firm mutation testing with

$s' = s$. In addition, if we allow the final state to be observed in strong mutation testing then this is firm mutation testing in which s' is the end of the program. In essence, firm mutation testing allows the tester to focus on parts of the computation in a program: that associated with paths from s to s'.

3.2 Different programming paradigms

The original work on mutation testing concerned imperative programming languages and, in particular, FORTRAN. This work developed the 22 Mothra mutation operators that have formed the basis for mutation testing. However, when considering programming languages with additional features it is necessary to develop new mutation operators. In particular, several researchers have investigated mutation testing for object-oriented languages, devising new mutation operators, called *class mutation operators*, and new tools. Figure 2 lists Java class mutation operators implemented in the tool MuJava alongside traditional mutation operators [16, 20].

A recent study applied class mutation operators and a set of traditional operators (ABS, AOR, LCR, ROR, UOI) to 11 classes of a Java program BCEL [16]. Interestingly, it was found that several of the class mutation operators produced mutants that were often killed by the test data generated to kill the traditional mutants. This suggests that several of the class mutation operators may add very little value. However, test data produced to kill the traditional mutants was found to be poor at killing mutants produced by certain class mutation operators and in particular killed none of the six mutants produced

Name	Description
AMC	Access modifier change
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overriding method calling position change
IOR	Overriding method rename
ISI	super keyword insertion
ISD	super keyword deletion
IPC	Explicit call of a parent's constructor deletion
PNC	new method call with child class type
PMD	Member variable declaration with parent class type
PPD	Parameter variable declaration with child class type
PCI	Type cast operator insertion
PCD	Type cast operator deletion
PCC	Cast type change
PRV	Reference assignment with other comparable variable
OMR	Overloading method contents replacement
OMD	Overloading method deletion
OAC	Argument of overloading method call change
JTI	this keyword insertion
JTD	this keyword deletion
JSC	static modifier insertion
JSC	static modifier deletion 15
JID	Member variable initialization deletion
JDC	Java-supported default constructor creation
EOA	Reference assignment and content assignment replacement
EOC	Reference comparison and content assignment replacement
EAM	Accessor method change

using IOP, EOA, and EOC. While these results only concern 11 classes from one system, they do indicate that there may be significant differences in the value of the class mutation operators, an issue that requires further investigation.

4 Remaining challenges

While mutation testing is a powerful and conceptually appealing approach to testing, in practice it is mainly used in order to evaluate other testing techniques. This is mainly due to two practical problems that we now discuss.

4.1 The number of mutants

Normally mutation operators make small changes to the syntax of a program. Since there are many ways in which this can be done, even small programs have many mutants. For example, it was found that the triangle program, with 28 executable statements, had 951 mutants when using the Mothra mutation operators [21]. If strong mutation testing is being applied then all of the mutants have to be generated, stored, and executed. This makes it hard for mutation testing to scale beyond small programs. In weak mutation testing we do not have to separately generate and execute all of the mutants, but the number of mutants is still problematic.

One approach to overcoming this problem is to only use some of the mutants. A possibility that has been discussed is to randomly generate some of the mutants. In experiments with four small programs it was found that using only 10% of the mutants led to test suites that killed over 96% of the entire

set of mutants [25]. It is important to note that random generation was used for each type of mutant (each operator). An alternative to randomly generating mutants is *selective mutation* (see, for example, [19]) in which we generate fewer mutant by not using all of the mutation operators. The aim is to significantly reduce the number of mutants produced without reducing the effectiveness of mutation testing. As a result, ideally we want to eliminate mutation operators that produce many mutants and where the resultant mutants add little value.

Early work on selective mutation testing devised the E-selective operator set $\{ABS, AOR, LCR, ROR, UOI\}$ [19], which is a subset of the Mothra operators. In an empirical study it was found that test suites that killed all non-equivalent mutants produced using the E-selective operators also killed over 98% of non-equivalent mutants produced by Mothra operators [19]. In addition, while the number of mutants produced using the Mothra operators is quadratic in the number of lines of code and the number of variable references, the number of mutants generated using the E-selective operators is linear [19]. Interestingly, similar results were found at around the same time using only the ROR and ABS operators but not using the term 'selective mutation': for four programs the average mutation score for test suites that killed these mutants was over 95% [25].

4.2 Equivalent mutants

In strong mutation testing a mutant m of a program p is an equivalent mutant if m and p are functionally equivalent and similar definitions hold for weak

and firm mutation testing. Equivalent mutants can be problematic since it is impossible to find test data to kill them. Thus, if the set M of mutants used for program p contains equivalent mutants then we need to determine which mutants are equivalent in order to find the mutation score $E_f(\mathcal{T}, M, p)$ of a test suite $\mathcal{T}$.

Unfortunately, it has been found that the standard mutation operators can lead to many equivalent mutants. For example, with the triangle program it was found that 109 of the 951 mutants produced using Mothra were equivalent mutants [21]. Frankl et al. also found that equivalent mutants create a significant burden [6]:

> "Even for these small programs the human effort needed to check a large number of mutants for equivalence was almost prohibitive."

The significance of the equivalent mutation problem has motivated two main complementary approaches to reducing this problem: to automatically detect equivalent mutants or to produce fewer equivalent mutants.

DeMillo and Offutt [21] showed that the problem of producing test input that kills a mutant can be represented as a constraint satisfaction problem. As a result, it is possible to automatically generate test data to kill a mutant and also to automatically determine that some mutants are equivalent mutants. Naturally, since this problem is generally uncomputable it is unreasonable to expect any technique to detect all equivalent mutants.

Offutt and Craft [18] noted that some equivalent mutants are generated by mutations that, because of the nature of the program, are similar to optimisation

techniques used by compilers. They showed that, as a result, it is possible to automatically detect some equivalent mutants using relatively simple and inexpensive analysis.

Hierons et al. [9] showed that it is possible to predict in advance that some mutants will be equivalent mutants and used an approach based on program slicing in order to achieve this. This makes it possible to produce fewer equivalent mutants without eliminating any of the mutants that provide value: those that are not equivalent.

We can conclude that there has been significant progress in the equivalent mutant problem. This is promising, especially since there has also been progress in automated reasoning techniques such as constraint satisfaction and model checkers and a significant increases in the computational power that is available.

5 Mutating models

The original work on mutation testing mutated the source code of a program. However, if we have a specification or model representing some aspect of the required behaviour of the SUT then we can mutate this. Mutants produced from the source code can be seen as representing programming mistakes. In contrast, models and specifications are at a higher level of abstraction and so the mutants of such documents can be seen as corresponding to design errors.

In order to automate the mutation of a document we at least require a formal syntax and in order to reason about the relationship between a model

and a mutant we require a formal semantics. Mutation testing with models and specifications has therefore focussed on classes of models that have a formal definition. There has been particular interest in state machines since these are relatively straightforward and are widely used. In particular, the UML contains a state machine language, UML Statecharts, and Statecharts [7] are widely used for designing embedded systems. Most work on mutation testing from state based models has used either *finite state machines* (FSMs) or *extended finite state machines* (EFSMs).

Throughout this section N denotes the deterministic and completely specified FSM $(S, s_0, X, Y, \delta, \lambda)$, where S is a finite set of states, $s_0 \in S$ is the initial state, X is the finite input alphabet, Y is the finite output alphabet, δ is the state transfer function of type $S \times X \rightarrow S$ and λ is the output function of type $S \times X \longrightarrow Y$. Given state s and input x, if N receives input x when in state s it moves to state $s' = \delta(s, x)$ and outputs $y = \lambda(s, x)$. Here the tuple $(s, s', x/y)$ is a *transition* and so a finite state machine is defined by a finite set of states, one of which is the initial state, and a finite set of transitions. One of the benefits of mutating models rather than code is that the mutations we make are much closer to mutating the semantics since typically the relationship between syntax and semantics is relatively simple. This is particularly the case when mutating an FSM since mutation operators change the transitions and the transitions contribute directly to output.

There are a number of natural mutation operators for an FSM N and these involve mutating the transitions of N. For example, we can change the target

state of a transition or change the output on a transition. One of the benefits of using FSMs is that it is possible to decide whether an FSM N and a mutant N' are equivalent and, if they are not, to find an input sequence that distinguishes them. In addition, this can be achieved in time that is (low order) polynomial in terms of the number of states and inputs of N and N'. Thus, the equivalent mutant problem is much less significant when considering FSMs. Interestingly, there are several techniques for generating a test suite that kills all mutants of an FSM defined by a fault domain without explicitly generating these mutants (see, for example, [2, 8, 11]).

While FSMs are straightforward to analyse, they are also relatively inexpressive and are not appropriate for describing systems that have data as well as state. Instead, it is possible to use EFSMs, which extend the FSM formalism by including an internal memory represented by a set of variables and transitions that can access and change the values of the internal variables and can have preconditions/guards that refer to the input and internal variables. Languages such as Statecharts [7], SDL [14], and Estelle [13] essentially describe EFSMs with additional constructs such as hierarchy. Researchers have therefore defined mutation operators for the data aspects of an EFSM and the additional syntactic features of these languages [22]. For example, operations on the transitions are typically expressed using a syntax similar to procedural code and thus it is possible to adapt many of the traditional mutation operators that mutate assignments. In addition, the guards are predicates and so we can apply the relevant previously developed mutation operators. Unfortunately, if the full

features of these languages are used then the equivalent mutant problem returns: if unbounded types are allowed then equivalence is undecidable but even if bounded types are used the time involved in deciding equivalence depends on the product of the sizes of the domains of the internal variables and so there can be a combinatorial explosion.

Interestingly, recent work has looked at probabilistic FSMs (PFSMs) in which there can be multiple transitions with input x leaving a state s and each is given a probability [10]. The use of probabilities is relevant when there is a sharing of resources, as encountered if there are Quality of Service requirements. In addition, if the SUT is implemented through several subsystems communicating via an unreliable medium then the behaviour is probabilistic and there are several probabilistic algorithms. The mutation operators are essentially those used for FSMs plus operators that change probabilities or introduce new transitions. Naturally, if the probability of a transition from state s with input x is changed then the probabilities of at least one other transition from s with input x must be changed in order to ensure that the probabilities associated with s and x still sum to 1. Interestingly, the equivalent mutant problem can be solved in polynomial time for PFSMs [10]. PFSMs contain non-functional information, probabilities, and so this work appears to be the first that has mutated non-functional properties.

6 Conclusions

In mutation testing we take some artifact, usually a program, and produce mutants by introducing small changes into it. This is done using a set of mutation operators, which are ways of introducing changes. If a test suite distinguishes between the original program and a mutant then it kills the mutant. The idea is that if our test suite kills the mutants then it is likely to be good at distinguishing between our program and other programs. Thus, if our program is faulty then the test suite is likely to be able to distinguish between our program and a correct program and it can only do this by leading to a failure. Mutation testing is thus a conceptually appealing approach to assessing the quality of a test suite since it relates to the ability of a test suite to find faults (as represented by mutants). Mutation testing was initially introduced for testing procedural code and the original 22 Mothra mutation operators reflect this. These have been complemented by operators developed for object-oriented features of languages such as Java.

While mutation testing is conceptually appealing and powerful, there are several practical issues that have led to it mainly being used for assessing other test techniques. First, for even a small program we obtain many mutants. As a result, subsets of the standard mutation operator sets have been proposed and there is empirical evidence that suggests that relatively little is lost in using these subsets. An additional issue is that changes to the syntax of a program do not necessarily change its semantics: mutants may be functionally equivalent to the original program and such mutants are called equivalent mutants. It can

be extremely expensive to determine which mutants are equivalent mutants but if we do not do this then it is hard to interpret any measure of the number of mutants killed by a test suite. Unfortunately, the problem of deciding whether a mutant is an equivalent mutant is generally undecidable but there are approaches that either reduce the number of equivalent mutants produced or use methods such as constraint satisfaction to automatically determine that some of the mutants are equivalent mutants and eliminate others.

The early work on mutation testing concerned the mutation of programs. More recent work has, however, looked at mutating models or specifications. Mutations at this level potentially represent different types of faults since the artifacts considered are at a different level of abstraction. One of the most promising lines of work is the mutation of finite state machines (FSMs) or extended finite state machines (EFSMs). One of the benefits of using FSMs is that equivalence is decidable and can be decided in low order polynomial time. As a result the equivalent mutant problem disappears and, in addition, it is relatively straightforward to automatically generate test cases to kill a mutant. Unfortunately, FSMs are not always sufficiently expressive and the problem of determining whether a mutant of an EFSM is an equivalent mutant is computationally hard. In this line, recent work has looked at the use of probabilistic FSMs and this may be the first piece of work to mutate non-functional properties of an artifact.

Mutation testing has progressed significantly since the early work on mutating FORTRAN programs. We now have some empirical evidence regarding

its effectiveness, techniques that help reduce the number of mutants and help tackle the equivalent mutant problem, and approaches to mutating models and specifications. In addition, the progress in automated reasoning in areas such as constraint satisfaction and model checking, combined with increases in computational power, significantly increase the scope for automation. These factors may well combine to make mutation testing an important practical test technique rather than an approach mainly used to assess other test techniques.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th International Conference on Software Engineering (ICSE 2005)*, pages 402–411. ACM Press, 2005.

[2] T. S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

[3] R. DeMillo, D. Guindi, K. King, M. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151. IEEE Computer Society Press, 1988.

[4] R. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practical programmer. *IEEE Computer*, 11:31–41, 1978.

[6] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems Software*, 38:235–253, 1997.

[7] David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[8] F. C. Hennie. Fault-detecting experiments for sequential circuits. In *Proceedings of 5th Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, 1964.

[9] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Journal of Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[10] R. M. Hierons and M. G. Merayo. Mutation testing from probabilistic finite state machines. In *3rd Workshop on Mutation Analysis (Mutation 2007)*, pages 141–150. IEEE Computing Society Press, 2007.

[11] R. M. Hierons and H. Ural. Optimizing the length of checking sequences. *IEEE Transactions on Computers*, 55(5):618–629, 2006.

[12] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8:371–379, 1982.

[13] ESTELLE: A formal description technique based on an extended state transition model, 1989. IS-9074. International Standards Organization.

[14] ITU. Recommendation Z.100: CCITT Specification and Description Language (SDL), 1992.

[15] K. N. King and A. J. Offutt. A FORTRAN language system for mutation-based software testing. *Software Practice and Experience*, 21:686–718, 1991.

[16] Y.-S. Ma, M.-J. Harrold, and Y. R. Kwon. Evaluation of mutation testing for object-oriented programs. In *28th International Conference on Software Engineering (ICSE 2006)*, pages 869–872. ACM Press, 2006.

[17] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, 1992.

[18] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[19] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5:99–118, 1996.

[20] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon. The class-level mutants of MuJava. In *Workshop on Automation of Software Test (AST 2006)*, pages 78–84. ACM Press, 2006.

[21] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability*, 7(3):165–192, 1997.

[22] T. Sugeta, J. C. Maldonado, and W. E. Wong. Mutation testing applied to validate SDL specifications. In *16th IFIP International Conference on Testing of Communicating Systems (TestCom 2004)*, volume 2978 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2004.

[23] J. Voas and G. McGraw. *Software fault injection: inoculating programs against errors*. John Wiley and Sons, 1998.

[24] K. S. How Tai Wah. A theoretical study of fault coupling. *Software Testing, Verification and Reliability*, 10(1):3–45, 2000.

[25] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[26] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*. IEEE Computer Society Press, 1988.