A Study of Equivalent and Stubborn Mutation Operators using Human Analysis of Equivalence

Xiangjuan Yao
College of Science, China University of Mining and Technology, China

Mark Harman
CREST Centre, University College London, UK

Yue Jia
CREST Centre, University College London, UK

ABSTRACT

Though mutation testing has been widely studied for more than thirty years, the prevalence and properties of equivalent mutants remain largely unknown. We report on the causes and prevalence of equivalent mutants and their relationship to stubborn mutants (those that remain undetected by a high quality test suite, yet are non-equivalent). Our results, based on manual analysis of 1,230 mutants from 18 programs, reveal a highly uneven distribution of equivalence and stubbornness. For example, the ABS class and half UOI class generate many equivalent and almost no stubborn mutants, while the LCR class generates many stubborn and few equivalent mutants. We conclude that previous test effectiveness studies based on fault seeding could be skewed, while developers of mutation testing tools should prioritise those operators that we found generate disproportionately many stubborn (and few equivalent) mutants.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Mutation Testing, Equivalent Mutant, Stubborn Mutant

1. INTRODUCTION

In mutation testing, faults are deliberately inserted into a program to create a mutant version; the mutant simulates the effect of a real fault [29]. Mutation testing has been widely studied as a means of validating the fault finding ability of test suites by seeding faults [11, 17, 43]. Mutation testing has also been used as a means of generating test suites that find these seeded faults [19, 24, 46]. We believe that this paper's findings have implications for both mutation based test assessment and test data generation.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568265

A test input that reveals a difference between the bahaviour of a mutant and the original program from which it is constructed is said to 'kill' the mutant. The motivation for all approaches to mutation testing is that test data that kills mutants should hopefully also detect real faults [29], thereby making mutation testing useful for both test effectiveness measurement and test data generation.

The two problems that have traditionally inhibited wider application of mutation testing are the equivalent mutant problem and the large number of possible mutants that might need to be considered. The problems posed by the large number of mutants have been partly dealt with, through selection [10], mutant schema [51] and search [28, 54], but the problem of mutant equivalence remains unsolved.

For some mutations, the semantics of the mutant and original will turn out to be equivalent, even though their syntax is different. It is undecidable whether a mutant is equivalent to the program from which it is constructed [12, 42]. This is the equivalent mutant problem: A tester is never sure whether unkilled mutants are merely hard to kill (more test effort is required) or equivalent (attempts to kill them are futile). Mutation operators that tend to generate disproportionate numbers of equivalent mutants will thus bias results, and much effort will be wasted trying to kill them.

In this paper we want to better understand the relationship between equivalent mutants and so-called stubborn [26] (aka super-hero [20]) mutants. That is, assuming testing has been reasonably thorough, then the set of mutants that remain unkilled will fall into one of two categories:

1. **Equivalent**: The mutants in this set cannot be killed because they are equivalent to the original program. No possible test input exists that can distinguish their behaviour from that of the original program.

2. **Stubborn**: The mutants in this set *can* be killed. Each stubborn mutant does have a test input that distinguishes its behaviour from that of the original program. However, none of these distinguishing test inputs has yet been found.

Naturally, the definition of a stubborn mutant depends upon the definition of 'reasonably thorough' testing. As we increase the fault revealing power of the testing process, there will be consequently fewer stubborn mutants. In the limit, if we define reasonably thorough testing to be exhaustive testing, then no mutant will be stubborn, since any unkilled mutant must be equivalent. However, exhaustive testing is just as infeasible as the equivalent mutant problem itself.

In this paper we define a mutant to be stubborn if it remains unkilled by a test suite that covers all feasible branches. Like the equivalent mutant problem, branch feasibility is also undecidable. Therefore, for each of the 18 programs we study, we also use human effort and ingenuity to generate a test suite that covers all (feasible) branches. We believe this to be 'reasonably thorough' because branch coverage is the ultimate goal (as yet not fully realised) of many test data generation systems [4, 13, 21, 33].

We present the results of a study of equivalent and stubborn mutants for the 5-selective set of mutation operators. This is a widely used [29] category of mutation operators identified by previous work on selective mutation [45]. The set consists of five collections of mutation operators, each of which involves different numbers of specific mutation operators. In total our study considers 58 mutation operators to provide the following primary contributions:

1. Distributions: We report on the numbers and distributions of equivalent and stubborn mutants over the set of 58 operators and 18 programs, ranging from previously studied (tiny) programs through to medium sized real world programs. We find that the equivalent mutant problem affects programs of all sizes.
2. Stubbornness and Equivalence Relationship: We study the relationship between equivalence and stubbornness, identifying operators that create high numbers of equivalent yet few stubborn mutants. Such operators are less effective in test assessment and generation and so their continued use is called into question by our findings. Our results also provide evidence that some operators generate many stubborn yet few equivalent mutants. These might be prioritised in future work on mutation testing.
3. Causes of Equivalence: We report on the causes of equivalent mutants and their distribution over the consequent categories of equivalence. We find that surprisingly few equivalent mutants are caused by dead code. We also find that the largest single cause of equivalence is mutants that cannot even be killed by weak mutation testing. Furthermore, those that cannot be killed, even by strong mutation testing, are largely the result internal state changes of little consequence (i.e., never output), rather than those that fail to propagate to an output.
4. Size Effects: We report on the relationship between program size and the numbers of equivalent and stubborn mutants and the statistical correlations observed between mutant equivalence and stubbornness. We find that size and numbers of mutants are strongly correlated with equivalence (expected) but not with stubbornness (unexpected). We also found that the operators of the ROR class exhibit a moderately strong correlation between equivalence and stubbornness (but no other class did).

The rest of this paper is organised as follows. Section 2 motivates the research questions that we investigate, while Section 3 explains, in detail, the manual decision procedures used to answer them. Section 4 provides the results of the study, answering the research questions, while Section 5 draws out the actionable findings and recommendations for mutation testing that accrue as a result of our study. Section 6 considers the threats to the validity of our findings and to those of previous studies revealed by our work. Section 7 describes related work on equivalent mutants, while Section 8 concludes with a summary of our findings.

2. RESEARCH QUESTIONS

This section presents the research questions concerning equivalent and stubborn mutants (and the relationship between them), for which Section 4 provides the answers.

All mutation testing is affected by the possible distorting effects that the presence of equivalent mutants can have on the mutation score reported. Therefore, the natural first research question we consider concerns the prevalence of equivalent and stubborn mutants over the programs considered:

RQ1: What are the numbers and proportions of equivalent and stubborn mutants found overall and per program studied?

RQ2: What is the contribution of each mutant operator to the proportion of equivalent mutants found and the relationship between equivalent and stubborn mutants?

The relationship between equivalent mutants and stubborn mutants is interesting because they may be connected. In all mutation testing, it is natural to include as many stubborn mutants as possible (since these drive testing hardest), while also hoping to exclude as many equivalent mutants as possible.

A stubborn mutant is very hard to kill, while an equivalent mutant is impossible to kill. Perhaps all mutants lie on a spectrum of killability from easy to kill to impossible to kill. Alternatively, perhaps some mutation operators generate more stubborn mutants than equivalent mutants or vice versa.

Knowing about these properties of mutation operators will be useful in the design of mutation testing systems and choices of operators to be used. We examine the contribution of each mutation operator and class of operators and report on a statistical analysis of the correlations between stubborn and equivalent mutants.

RQ3: What are the causes of mutant equivalence?

We need to study the causes of equivalence because not all equivalent mutants are equal. For example, those that are caused by the inability of any test case to even *execute* the mutated code are merely a manifestation of 'dead code' in the program studied. They can be regarded as entirely a property of the program studied rather than the mutation operator concerned; all mutation operators applied at the same unreachable point would yield equivalent mutants. If we had a better understanding of the causes of mutant equivalence then this may help us to design mutation tools that can ameliorate their pernicious effects.

Finally, we ask whether there are any differences in equivalent and stubborn mutants based on the sizes of the programs studied:

RQ4: Does program size or the number of mutants correlate to the number of equivalent and number of stubborn mutants found?

We might expect that the more mutants we generate, the more stubborn and equivalent mutants we will find. We might also expect that larger programs (which tend to have more possible mutants) would also tend to have more equivalent and stubborn mutants.

Table 1: Subject programs ordered by Size in Lines of Code ('#LoC'). The '#Bran' column shows the number of branches in each program.

Name	#LoC	#Bran.	Description
`Min`	10	2	Minimum of two integers
`Bubble Sort`	16	6	Sorting routine
`Profit`	24	10	Salesperson's commission
`Mid`	26	10	Median of three integers
`Prime_num`	27	10	Prime number listing
`Triangle`	35	16	Widely studied program
`Insert`	35	12	Insertion sort
`Day`	42	23	Day order for given year
`Calendar`	137	29	Calendar for given year
`Carsim`	171	22	Driving simulator
`Tcas`	173	66	Aircraft collision avoidance
`Defroster`	230	82	Car defroster controller
`Schedule`	412	66	Priority scheduler
`Hashmap`	455	82	Information Management
`Replace`	564	66	Pattern matching
`Space`	9,564	1,190	Array language interpreter
`Flex`	10,459	1,286	Unix lexer utility
`Make`	35,545	3,838	Unix compilation utility

3. MANUAL ANALYSIS OF STUBBORN & EQUIVALENT MUTANTS

This section explains how we set up and carried out the procedures required to determine equivalence and stubbornness of mutants so that we can answer the four research questions set out in the previous section. Section 3.1 describes the 18 programs used in the study, while Section 3.2 describes the mutation operations applied to these programs. Section 3.3 explains, in detail, the procedure we adopted for determining whether a mutant is equivalent. Finally, Section 3.4 describes the procedure used to construct branch adequate test suites for each of the 18 programs studied, and how this determines whether a mutant is stubborn.

3.1 Subjects

Eighteen C programs were chosen for the investigation of equivalence and stubbornness. Information about each program, including name, size, number of branches, and a simple description of its functionality are presented in Table 1. This table is sorted by the sizes of the programs, measured in lines of code.

The programs `Min`, `Bubble_Sort`, `Profit`, `Mid`, `Prime_num`, `Triangle`, `Insert` and `Day` are eight trivial programs studied in previous work on equivalent mutants [41, 42]. Their inclusion allows us to investigate whether the size of a program influences the prevalence of equivalent mutants or their relationship to stubborn mutants.

The three programs `TCAS`, `Schedule` and `Replace` are part of the Siemens Suite (available from SIR [16]). These programs come with existing test suites. However, we did not use these existing suites because 1) they do not yield 100% coverage of feasible branches and 2) we wanted the same test suite construction method for all subjects.

Finally, we also include six 'real world' programs that vary in size from tiny to relatively more substantial. `Calendar` is the simple UNIX utility program and `Carsim` car cruise-control simulation component used by Mouchawrab et al. [37]. Though 'real', both are relatively small.

The program `Defroster` is an embedded system that implements the controller for the rear window defroster of an automobile. It is closed-source production software from DaimlerChrysler, generated as C code from a state-based model. The program `Hashmap` is an open source program for managing a hash table. The program `Space` is the widely-studied European Space Agency program (also available from SIR [16]). The programs `Flex` and `Make` are the well-known UNIX utilities.

The size, data types, and functionality of these programs are wide and varied. We conduct mutation testing by applying mutants to the whole program, at every point at which a mutation operator applies in all but the largest three cases (`Space`, `Flex` and `Make`).

It is infeasible to consider every possible mutant of these largest three programs. Instead, we adopted the following process for determining the parts of the program to be studied: Starting with the top level procedure m, we include all procedures transitively called from each of the procedures called by m, terminating when we have accrued at least 10 procedures. Using this selection procedure we identified 20 procedures from `Space` (824 LoC), 11 procedures from `Flex` (924 LoC) and 10 procedures from `Make` (672 LoC). In order to understand whether or not a mutant is equivalent and to construct test cases, we consider the entire program, but only these 41 selected procedures are actually mutated.

3.2 Mutation Operators Studied

A program can be mutated by applying a set of mutation operators to it. King and Offutt presented the first set of proposed mutation operators, each of which is represented by a three letter acronym [31]. However, subsequent work on selective mutation by Offutt and Lee, resulted in the identification of five classes of operators (ABS, AOR, LCR, ROR, and UOI) which were deemed to be sufficient to achieve almost full mutation coverage [40].

Many subsequent authors [29] have used only operators from these five classes, called the 'five-selective' mutation operators [40]. Therefore, we only use these five operator classes for our experiments too. A full description of each class of operators and the specific mutations that they involve can be found in Table 2. In total, we consider 58 specific mutation operators, covering all those found in these five classes.

To aid replication, we state exactly how the operators were applied to the program, since these details can vary from one approach to another: The mutation operators from the class LCR, ROR and AOR were applied to every expression and predicate. The ABS and UOI class of operators were applied only to the variables that occurred in expressions and predicates.

No mutation operators were applied to lvalue uses of variables so, for example, the lefthand side of an assignment statement is not mutated, but the expression on the right-hand side is. Operators are applied recursively to all sub expressions.

3.3 Manual Equivalent Mutant Decision Procedure

Since the determination of whether a mutant is equivalent is undecidable, we used a manual decision procedure in order to decide the answer to this question. Our decision procedure, implemented by purely manual inspection of code, is outlined in Figure 1 and defined in more detail below.

Table 2: The Five Mutant Operator Classes and the Specific Mutation Operators they Contain

Mutation operator class	Description	Specific mutation operators	Number of operators
ABS	Absolute Value Insertion	$\{(e,\texttt{abs}(e)), (e,\texttt{-abs}(e))\}$	2
AOR	Arithmetic Operator Replacement	$\{(x,y) \mid x,y \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%}\} \wedge x \neq y\}$	20
LCR	Logical Connector Replacement	$\{(x,y) \mid x,y \in \{\texttt{\&\&},\texttt{||}\} \wedge x \neq y\}$	2
ROR	Relational Operator Replacement	$\{(x,y) \mid x,y \in \{\texttt{>},\texttt{>=},\texttt{<},\texttt{<=},\texttt{==},\texttt{!=}\} \wedge x \neq y\}$	30
UOI	Unary Operator Insertion	$\{(v,\texttt{--}v), (v,v\texttt{--}), (v,\texttt{++}v), (v,v\texttt{++})\}$	4
Total			58

Though our process is entirely manual, it is not arbitrary; we followed a set procedure for identification of equivalent mutants and this allowed us to categorise the reason for equivalence. It is well known [52] that there are three necessary criteria for a mutant to be killed: Reachability, Infection and Propagation (RIP), each of which subsumes the preceding condition(s):

1. **Reachability (R)**: The mutation must be executed by a test case; the mutant is 'reached'. For non-equivalent mutants, the reachability criterion can be achieved by any branch adequate test set.

2. **Infection (I)**: Immediately after the execution of the mutant, the state must be infected. That is, the state after the execution of the mutation and the corresponding state in the original program must differ. A test case that achieves infection is said to 'weakly kill' the mutant [15, 29, 36].

3. **Propagation (P)**: The infected state must propagate to some point in the program at which it can be observed, such as an output or `return` statement. A test case that achieves propagation is said to 'strongly kill' the mutant [15, 29, 36].

This 'RIP framework' defines the three necessary conditions for killing a mutant and, implicitly, it also defines the conditions for a mutant to be equivalent; one or more of the necessary conditions must fail for a mutant to be equivalent. This gives us three broad categories of equivalent mutant, depending on whether they are equivalent because they cannot be reached, cannot infect the state or cannot propagate an infection to an output.

A mutant that cannot be reached by any possible test input also, by definition, cannot infect and also cannot propagate (so we denote this case ¬R¬I¬P). A mutant that can be reached by at least one test input but subsequently fails to infect the state for any reaching input cannot, by definition, propagate (so we denote this case R¬I¬P). A mutant that can be reached and can infect the state for at least one test input, but cannot propagate the infection to an output, will be denoted RI¬P.

Within these three broad categories of reasons for equivalence, we distinguish sub-categories which capture, in more detail, the reason for the failure of the necessary killing condition. This allows us to report, in more detail, the distribution of reasons for mutant equivalence. We adopted the human-evaluated decision procedure for mutant equivalence outlined in Figure 1. Each of the seven categories of equivalence is defined as follows:

Case 1.1 (¬R¬I¬P normal): The mutated predicate or statement is unreachable. Such a statement of predicate is dead code and so no test case can even executed the mutant. The number of mutants in this category is thus partly related to the amount of 'dead code' present in the programs under investigation. Previous work indicates that we might expect a small amount of dead code (perhaps about 2% [8]).

Case 1.2 (¬R¬I¬P short-circuit): We expected that most of the non-reaching mutants would fall into the category defined by Case 1.1. However, there is a special case, which applies to predicates only: the predicate is reached and evaluated, but the mutated part of the predicate is never evaluated due to 'short circuit' evaluation of predicate subexpressions. That is, in C, the boolean operators `&&` and `||` are evaluated using 'short circuit' evaluation. This introduces a sequence point [27], such that the righthand side of the boolean expression is not evaluated when the outcome of the overall expression can be determined from the lefthand side evaluation alone. For example, in the expression p `&&` q, suppose p turns out to be false, then q (and any mutant it might contain) will not be evaluated. Therefore, should it turn out that all reaching inputs cause p to evaluate to false, then any mutation to q will be equivalent.

Case 2.1 (R¬I¬P context free): In this situation the mutated expression is always equivalent to the unmutated expression, no matter what the program context in which the expression occurs. For example, if we mutate `abs(x)` to `abs(abs(x))` then this will result in an equivalent mutant, regardless of the program context in which it occurs. It is useful to distinguish this sub-case because it is far simpler to detect by manual inspection: since it is context free we need not be concerned with the state on each path that reaches the mutant.

Case 2.2 (R¬I¬P context sensitive): In this situation the mutant is executed but, for each test input that reaches and executes the mutant, it so-happens that the state is always one in which the mutant and the original yield identical results. For example, if we mutate `x+2` to `x+x` this is not equivalent according to Case 2.1 above. However, suppose it turns out that all reaching inputs arrive at the mutant in a state in which the value of `x` happens to be `2`. In this execution context the mutant will be equivalent; we say that the equivalence is 'context sensitive'.

Case 2.3 (R¬I¬P subpath equivalence): There is a special case of R¬I¬P that applies only to predicates: a mutated predicate is evaluated and yields a different result in the mutant and the original. However, all paths from the predicate yield identical states.

Case 1: ¬R¬I¬P	**Mutant cannot be reached by any test input**
	Case 1.1 (normal): Mutated statement or predicate cannot be reached
	Case 1.2 (short-circuit): Mutated sub-expression never evaluated though its containing predicate is
Case 2: R¬I¬P	**Mutant is reached by at least one test input, but no test causes state infection**
	Case 2.1 (context free): Infection can never occur in any state
	Case 2.2 (context sensitive): Not context free, but infection cannot occur in any *reaching* state
	Case 2.3 (subpath equivalence): Mutation changes path executed, but all paths are equivalent
Case 3: RI¬P	**Mutant is reached and infects the state, but no infection propagates to an output**
	Case 3.1 (unobservable): No output statement mentions an infected variable
	Case 3.2 (observable): Outputs mention infected variable(s), but infection fails to reach any

Figure 1: A Summary of the Human-Evaluated Decision Procedure for Mutant Equivalence

Case 3.1 (RI¬P unobservable): In this case, the mutant is executed and infects the state, but there simply is no output statement that mentions any infected part of the state. In this situation, the infection may reach the end of the program execution, but remains unobserved and so the mutant is not (strongly) killed.

Case 3.2 (RI¬P observable): In this case, there is an output statement that yields the result of a variable infected by the mutant. However, all paths to all such statements also contain an assignment statement that 'squeezes' [14] the state, so that any differences in the infected state and the original state are removed before they reach the output statement. Such a squeezing statement could be a killing assignment, like `x=1;` in the sense of 'killing assignment' used in the literature on compilers [3]. That is, a statement that always assigns a constant and so 'kills' the value of the variable to which it assigns. However, an assignment need not be a killing assignment to 'squeeze out' the value of the infected variable. For example, suppose the mutant affects only the least significant bit of the value of the variable `x`. For this mutant, the assignment `x=x<<1` (which shifts the contents of `x` one bit left) is one that removes all trace of the infection, yet it is not a killing assignment.

3.3.1 Manual Analysis

The seven classes defined above were used to categorise equivalent mutants according to the reason for their equivalence. We devoted 6 person months continuous effort to the determination of equivalence according to our human-evaluated decision procedure. That is, one of the authors (Xiangjuan Yao) was devoted to this task exclusively for 6 months.

In order to retain consistency, we did not use any automated tools to determine equivalence, other than running the test suite. All mutants which were not killed by the test suite were checked manually for equivalence.

In total, 4,181 mutants were generated. Of these 1,230 were unkilled by our branch adequate test suites and were thus manually checked using the decision procedure specified above. Of those manually checked, 946 ($\sim$ 23%) were found to be equivalent and 284 stubborn ($\sim$ 7%) according to the decision procedure.

The manual nature of our analysis renders it subject to human error. Of course, where we find that a mutant is not equivalent we can be sure that we are correct in this assertion because each such non-equivalence claim is accompanied by a test case that distinguishes between the mutant and the original program. However, for all cases where we declare a mutant to be equivalent, we cannot be *completely* sure.

Indeed, for these mutants, the undecidability of equivalence makes it unlikely that anyone could ever be completely sure, no matter how well examined. We therefore treat our scores for equivalent mutants as upper bounds and make all data available to support wider scrutiny:

`www.cs.ucl.ac.uk/staff/Y.Jia/projects/equivalent_mutants`

For complete details about the outputs selected, the mutants seeded, the test cases we constructed and the determination of which test kills which mutants (and the mutants we believe to be equivalent), the reader is refereed to this website. We hope that this website will be a useful resource for testing. Other researchers are welcome to use the test suites, mutants and other information in their research, so long as they acknowledge this paper as the source.

3.3.2 Determining what Constitutes an Output

For our study, we regarded an output to be any changes to the console or a file that can be observed as having occurred as the result of execution of any statement that produces such 'output'. However, some of the programs have no such output. Clearly, it would be unreasonable to regard all mutants in these programs as equivalent simply because they fail to contain output statements.

Therefore, we also deem any statement that returns values to the operating system through a `return` statement to be an 'output statement'. Furthermore, for the tiny programs used in some previous studies, the programs contained neither an output statement nor any `return` statement. This issue is reminiscent of the problem of determining the principal variables in slice based measurement of cohesion and coupling [7, 25].

More precisely, the ten programs Min, Bubble Sort, Profit, Mid, Prime_num, Triangle, Insert, Day, Calendar and Tcas all have either output or return statements. For Carsim and Defroster we identified key state variables and regarded these as output. For the other six programs (Schedule, hashmap, Replace, Space, Flex and Make) we compare the output (files, screen, error log etc.) for each task carried out by the system.

3.4 Semi-Manual Generation of Branch Adequate Test Suite

In order to construct a branch adequate test suite for each program studied, we could have used sophisticated automated test data generation techniques [18, 21, 24, 33]. However, though great advances have been achieved in automated test input generation, no tools can generate 100% branch adequate test data; this is theoretically impossible and practically, such tools remain limited [34].

We therefore constructed test suites using pure random testing to find those branches that can be easily covered and then augment the test suite with human generated test case design. We then manually constructed test cases specifically designed to execute any remaining uncovered branches until all branches were covered.

This means that there is one test case per branch uncovered by random testing. A smaller test suite could be found with equivalent coverage [44], but we are not presently concerned with efficiency of testing, only its effectiveness. That is, none of the results reported in this paper depend upon the efficiency of the testing process, but the determination of stubbornness rests of the effectiveness of our test suites.

We choose branch coverage as a measure of test effectiveness, because this is a criterion widely studied by many authors [4, 5, 9, 13, 21, 35]. Branch adequacy also corresponds to a well-defined lower bound for mutation killability: it ensures that the necessary reachability criterion is met by at least one test case for every possible reachable mutant.

We define a mutant to be a *stubborn mutant* if it is not equivalent, but it is not killed by the branch adequate test suite for the program that has been mutated.

4. RESULTS & ANSWERS TO RESEARCH QUESTIONS

This section reports results that provide the answer to each of the research questions. Section 5 draws actionable conclusions from these findings for mutation-based test suite assessment and test data generation. Section 6 considers threats to the validity of our findings (and also potential new threats to validity of previous studies that use fault seeding to assess test technique effectiveness).

4.1 RQ1: Prevalence of equivalent and stubborn mutants

Table 3 shows the total number of mutants and proportions of equivalent and stubborn mutants per program studied over each of the five classes of mutation operator. From this table we see that equivalent mutants, which account for 23% of all mutants, are more prevalent than stubborn mutants, which account for only 7% of all mutants.

We also immediately notice some startling disparities in the numbers of equivalent and stubborn mutants generated by each operator class. For example, while almost half (47%) of all ABS mutants are equivalent, *none* are stubborn, whereas just over a quarter (26%) of all LCR mutants are stubborn, while very few are equivalent (only 2%).

We can see that almost all programs, no matter how small, do possess equivalent and stubborn mutants. Figure 2 shows the proportionate contribution of the mutants from each program to the total number of all equivalent and all stubborn mutants (as a percentage of all mutants drawn from all 18 programs).

We can see that almost a quarter of the equivalent mutants come from the program `Replace`, while `Tcas` contributes most stubborn mutants. However, there is no obvious relationship between the contribution of equivalent and stubborn mutants made by each of the 18 programs. We can see no obvious trend in Figure 2 and found no high Spearman rank correlations between the two.

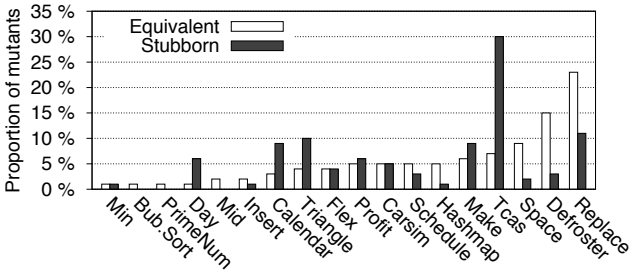


Figure 2: The proportionate contribution of the mutants from each program to the total number of all equivalent and all stubborn mutants (as a percentage of all mutants)

4.2 RQ2: Relationship of Equivalent and Stubborn Mutants for Each Operator

From Figure 2, we can see that there does not appear to be a relationship between overall numbers of mutants and the proportion which are equivalent or stubborn for each program. However, consider Figure 3, which shows the proportion of equivalent and stubborn mutants over all mutation operator classes (thereby visualising the summary data for each operator class from Table 3). This figure reveals some interesting findings: it is very clear that the ABS operators are problematic for mutation testing; they generate a great many equivalent mutants and no stubborn mutants. By contrast, the LCR operators generate stubborn mutants, but very few equivalent mutants.

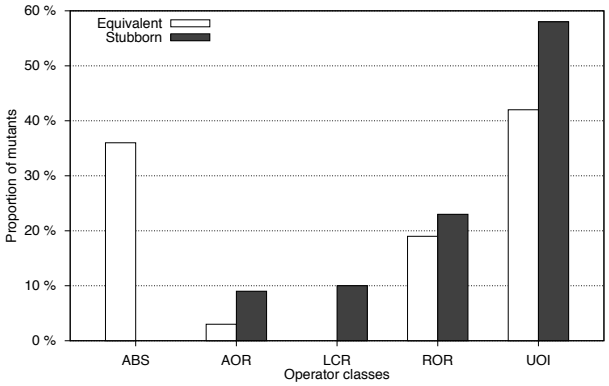


Figure 3: Proportion of equivalent and stubborn mutants for each of the five operator classes

The other three classes of mutants generate both equivalent and stubborn mutants, so we analyse these in more detail: Figure 4 shows the contribution of each operator to the overall number of equivalent and stubborn mutants (as proportions) for each operator.

We can see that post increment/decrement operators are problematic (they generate a preponderance of the equivalent mutants and fewer stubborn mutants), while pre increment/decrement operators behave in exactly the opposite manner and may be thus much more practical.

In Figure 4, operators are ordered in descending order of their contribution to the number of equivalent mutants overall. Notice that ROR operators show some degree of correlation between equivalent and stubborn mutants, which is not present in the AOR category. We used a Spearman rank correlation test to explore this correlation further.

Table 3: Total mutants (T), proportion of equivalent mutant (E) and stubborn mutants (S)

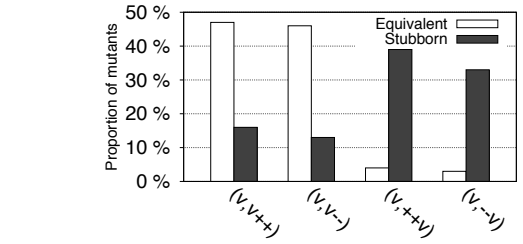
Program	Mutation Operator Class															Totals		
	ABS			AOR			LCR			ROR			UOI					
	T	E	S	T	E	S	T	E	S	T	E	S	T	E	S	T	E	S
	#	%-age		#	%-age		#	%-age		#	%-age		#	%-age		#	%-age	
`Min`	6	0	0	0	0	0	0	0	0	5	20	0	20	40	10	31	29	6
`Bubble Sort`	6	0	0	4	50	0	0	0	0	5	20	0	20	30	5	35	26	3
`Profit`	24	50	0	100	5	0	0	0	0	25	20	0	84	29	19	233	20	7
`Mid`	12	0	0	0	0	0	0	0	0	25	20	0	68	21	0	105	18	0
`Prime_num`	6	50	0	12	17	0	0	0	0	10	20	0	24	8	0	52	17	0
`Triangle`	18	50	0	4	0	0	2	0	0	40	18	8	104	23	25	168	24	17
`Insert`	14	14	0	0	0	0	0	0	0	10	10	10	40	40	5	64	30	5
`Day`	4	50	0	16	0	50	3	0	0	25	16	16	28	29	18	76	18	22
`Calendar`	22	50	0	76	0	18	2	0	0	20	20	15	60	27	13	180	17	14
`Carsim`	12	42	0	56	13	5	0	0	0	45	18	7	64	38	11	177	25	7
`Tcas`	54	48	0	4	0	25	17	0	59	70	29	26	76	26	74	221	30	38
`Defroster`	6	100	0	24	58	0	8	0	25	105	49	2	152	47	3	295	48	3
`Schedule`	52	50	0	8	0	0	2	0	50	15	13	7	84	24	7	161	30	5
`Hashmap`	46	43	0	28	0	0	3	0	67	60	12	0	144	14	0	281	17	1
`Replace`	274	48	0	160	0	0	25	8	8	235	2	3	540	14	4	1,234	17	3
`Space`	80	50	0	8	0	0	14	0	7	205	16	2	48	29	0	355	25	2
`Flex`	42	57	0	56	0	0	17	0	41	100	16	4	24	0	0	239	17	5
`Make`	42	50	0	40	0	0	10	0	20	90	12	17	92	30	10	274	22	9
Grand Total	720	47	0	596	5	4	103	2	26	1,090	17	6	1,672	24	10	**4,181**	**23**	**7**

Ordering the 20 operators in the ROR class by proportion of equivalent and stubborn mutants exhibits a rank correlation ($\rho = 0.82$), whereas ranking the 30 operators of the AOR class in the same way yields no such correlation ($\rho = -0.25$). Over all 58 mutation operators there is also no correlation between the ranking of operators by their contribution to equivalence and stubbornness ($\rho = 0.45$).

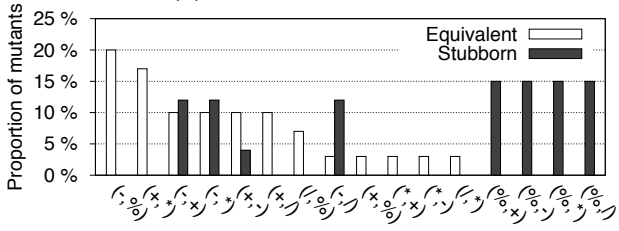
We conclude that one cannot have stubborn mutants without equivalent mutants in the ROR class and that there is a tendency for the ROR operators that generate high numbers of stubborn mutants to also generate high numbers of equivalent mutants. By contrast, we find that some of the AOR operators ($\{(\%,+),(\%,-),(\%,*),(\%,/)\}$) are (perhaps surprisingly) good at generating stubborn mutants (but no equivalent mutants), while others are, unfortunately, good at generating equivalent mutants. That is, the four AOR operators that replace '`-`' with one of $\{+,*,/,\%\}$ generate about half of all AOR's equivalent mutants.

Finally, checked the correlation between the numbers of equivalent and stubborn mutants, overall and within each operator class. We speculated that equivalence might be merely an 'extreme form' of stubbornness and we believed other researchers may have made the same implicit assumption. Our results surprised us and challenged this assumption: they revealed no evidence for any such correlation.

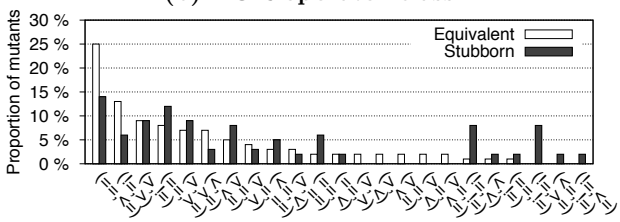
The strongest correlations were for the operator (`!=,>`) for which $\rho = 0.61$ and for (`v,v++`) for which $\rho = 0.51$, neither of which is particularly strong. It should be noted that for many (35 of the 58 operators), there were simply too few mutants to compute a meaningful ρ value, while for those with sufficiently many data points, these two reported above were the only results that indicated even a *mild* correlation. We conclude that we can find little evidence of any noticeable correlation between equivalence and stubbornness.



(a) UOI operator class



(b) AOR operator class



(c) ROR operator class

Figure 4: Stubbornness and Equivalence for the AOR, ROR and UOI Classes. AOR and ROR operators not listed in this figure generate no stubborn and no equivalent mutants.

4.3 RQ3: Causes of Equivalent Mutants

Figure 5 shows the proportion of each category of equivalent mutants over all equivalent mutants. This reveals a number of interesting observations about the causes of equivalence. We see that there are no equivalent mutants due to unreachable code.

This is not because our programs contain no unreachable statements; there were unreachable statements in `Tcas` and `Defroster`, for example. However, the unreachable branches contain few statements, and those that they do contain attract no mutation operators because their expressions were merely constants. For example, in the defroster program, there is an unreachable branch:

```
if ((We1_BA_DEF_ev_ctr0 >= ((signed short)3250))
   && error_e>0 && confirmation_e>0)
{
AU8.We11_BLINK_OUT = 0;
}
```

The right hand side of the unreachable assignment to `AU8.We11_BLINK_OUT` is a constant and so it is not mutated.

Similarly, Case 2.1 (context free) contains no equivalent mutants, revealing another interesting property of the programs under study: though they contain redundant (unreachable) code, none of them contains the kind of 'redundant' expression that would lead to a context free equivalent mutant.

The mutation operators studied in this paper are able to generate such equivalent mutants, in principle. For example, should the expression `x+0` occur in the program, then it would be equivalently replaced by `x-0`. However, no such trivially equivalent code was present in any of the 18 subjects studied.

We found that there are cases where the predicate is infected and causes execution to follow a different path, yet no state infection occurs (Case 2.3 of mutant equivalence). This surprised us: it seems that programmers do write programs which, when mutated, can have disrupted control flow, but that this disruption does not affect the state. We observed this behaviour in 4 of the programs (`profit`, `Mid`, `Carsim` and `Space`).

The most common cause of equivalence is Case 2.2, which accounts for 51% of all equivalent mutants. These mutants will be equivalent, even under weak mutation testing, because they fail to infect the state. This is an interesting finding because weak mutation testing was introduced as a weaker alternative to strong mutation testing in the hope that it might overcome some of strong mutation's difficulties [53]. Mutants that are equivalent due to Cases 1.1, 1.2 and 2.1 will also be equivalent under weak mutation. Our results thus suggest that many of the overall population of equivalent mutants (59%) will still be equivalent, even for weak mutation testing.

Mutants that are equivalent because of Cases 3.1 and 3.2 can be killed by weak mutation testing, but will be equivalent for strong mutation testing. Considering these strongly equivalent mutants, it is striking to note how many of them occur due to Case 3.1 (unobservable). This means that the mutants have affected the state, but the effect is not 'important' to the computation. That is, the program never externalises the corrupted part of the state, using an output statement (even given our broad interpretation of the notion of 'output').

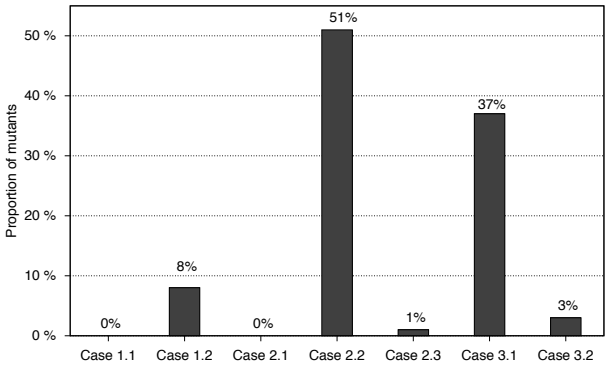


Figure 5: Proportion of All Equivalent Mutants that Result from Each of the Seven Causes of Equivalence

Had the oracle exposed more of the internal state, then more of these mutants would be equivalent. This finding may also have a bearing on the work on fault tolerant computing [38]; it suggests that many faults can be tolerated. That is, Case 3.1 faults are those that corrupt the state, but for which this corruption 'does not matter'.

4.4 RQ4: The effect of Size on Mutant Equivalence and Stubbornness

We found a strong correlation between the overall number of mutants and the numbers of these that are equivalent ($\rho = 0.90$). This was expected because 'more mutants' should surely mean 'more equivalent mutants'. We also expected a similar correlation for stubborn mutants ('more mutants' should surely also mean 'more stubborn mutants'?). However, there was no such correlation ($\rho = 0.47$), indicating that stubbornness appears to be more related to special program circumstances than equivalence.

We also investigated the correlation between the size of the program and the number of equivalent mutants. For this correlation test we ranked only the 15 programs for which we constructed all possible mutants (and for which it is therefore meaningful to consider such a correlation). Once again, we found some degree of Spearman rank correlation between the size of the program and the number of equivalent mutants ($\rho = 0.81$), but not between the program size and the number of stubborn mutants ($\rho = 0.46$).

We already found that for the most operators and operators classes there is little connection between equivalence and stubbornness (ROR being the exception). We now also conclude that there is also no correlation between size and stubbornness, though there is for equivalence, further highlighting their different character. These differences between equivalence and stubbornness are optimistic findings because they indicate that mutation testing systems might be redesigned to reduce the generation of equivalent mutants, as we discuss in the next section.

5. ACTIONABLE FINDINGS

Our results suggest that mutation testing tools should drop or de-emphasise operators that create many equivalent but few stubborn mutants (unless there is some other reason to retain these). For instance, inclusion of the ABS class of operators would require some careful justification since these operators seem to be responsible for large number of equivalent mutants yet very few stubborn mutants.

By contrast, the LCR class of mutant operators may be more valuable than previously realised. Our results suggest that its two logical operators tend to create a relatively large number of stubborn mutants yet mercifully few equivalent mutants. Designers of mutation testing tools might want to favour the inclusion of this operator over others, provided this does not affect mutation effectiveness.

The ROR operators generate many stubborn and also many equivalent mutants. Indeed, our results indicate that there is some correlation between the two. As such, this operator should be retained for thorough testing (since it creates stubborn mutants). However, it might be dropped if the test scenario is particularly sensitive to the pernicious effects of the equivalent mutant problem.

Work on equivalent mutant detection should focus on the ROR class of operators, since the prevalence of equivalent and stubborn mutants appears to be correlated in this class. This correlation may make equivalent mutants unavoidable, without some form of detection technique. That is, when we seek to retain the effectiveness that accrues from stubborn mutants, we may encounter the associated equivalent mutants, which will harm the accuracy of the assessment of test effectiveness.

Our results also indicate that the current widely-used categorisation of mutants into the five classes ABS, AOR, ROR, LCR, UOI needs to be refined. The sub-categories of operator are important in their own right; we found noticeable differences in their behaviours. For example, not all AOR operators are equal: we found that replacing '-' with any of the other four arithmetic operators tended to generate a disproportionately high number of equivalent mutants (with the benefit of comparatively few stubborn mutants). More work is required to determine whether this is a more generalisable phenomenon, in which case some of the AOR class of operators might be prioritised over others.

Finally, in the UOI category, there is a clear distinction in the behaviour of the pre– increment and decrement operators, compared to their post– increment and decrement counterparts. While the post– operators tend to create both equivalent and stubborn mutants, the pre– operators tend to create proportionately more stubborn mutants than the post– operators and very few, if any equivalent mutants. This suggest that the tester might want to favour pre– operators over post– operators.

6. THREATS TO VALIDITY

The most glaring threat to the validity of the findings reported in this paper is the inherent undecidability of the equivalent mutant problem and, by extension, the stubborn mutant problem. No approach to this problem can be free from threats to validity. Despite these threats, we believe it necessary to address the equivalent mutant problem more thoroughly than has been achieved in the literature. Without a better understanding of the properties of mutation operators with regard to equivalence, we cannot design reliable mutation testing systems. Our results may also be affected by our test suite. Further work is required in different test suites since these will affect the determination of whether a mutant is stubborn.

However, as our results indicate, previous test effectiveness studies that used fault seeding may be the subject of previously under-appreciated threats to the validity of their findings.

For example, previous work on mutation testing in which the ABS operator is used may suffer disproportionately from threats to the validity of their findings due to the unusually high prevalence of ABS-created equivalent mutants. Also, suppose there are two studies of testing techniques, Study A and Study B. In Study A, the programs just so happened to contain proportionately more relational operators and fewer arithmetic operators compared to Study B. Even if the researchers control for size and use the same set of mutation operators *and the same testing tools*, we can expect (from our findings) that Study A will report much lower mutation scores than Study B.

These threats to validity affect previous work that has attempted to use operators from the five-selective set to assess testing techniques based on fault seeding. However, our own study is also the subject of threats to validity from the equivalent mutant problem. Our claims about equivalence need to be treated cautiously since they are the result of human analysis and not automated and proven-correct algorithms. While humans are error prone, no automated solution can be complete due to undecidability.

In order to ameliorate the effects of this inherent threat, we seek to be cautious in our findings, paying attention only to strongly observed effects. Such stronger effects are less likely to be affected by changes in the details regarding the numbers of equivalent mutants. We also used a statistical analysis of the correlations between stubborn and equivalent mutants, since we can hope that such statistical techniques may prove to yield reliable findings, even in the presence of 'human error noise'.

Finally, we make our programs, mutants, test suites, and all data about equivalence and stubbornness available online to support wider analysis, scrutiny and replication. All test data, mutants and programs can be found on the web at the project website.

Naturally, results also depend on the choice of subject programs, as they do with any study. We partly address this threat by studying a relatively large and varied set of programs. The programs studied are also of different sizes and come from many different domains. However, all our programs are C Language programs. Further generalisability requires additional replication studies, as always with this kind of empirical study.

7. RELATED WORK

There has been much work on techniques to reduce the number of equivalent mutants, either by detecting them or preventing their creation. These approaches have used a variety of techniques, including constraint solving [30, 41, 42], slicing [23, 26, 52], compiler optimisation [12, 39], impact analysis [49, 50] and search based software engineering [2, 47]. Of course, the undeciability of equivalence means that such approaches must be inherently partial and their true performance remains unknown.

Baldwin and Sayward [6] proposed an approach that used compiler optimisation techniques to detect equivalent mutants. Their approach uses six types of compiler optimisation rule. The compiler optimisations were implemented and empirically studied by Offutt and Craft [39] who reported that 10% of all mutants were equivalent mutants for the subject programs they studied. However, the programs were small and the results only applied to mutants that could be detected by the compiler optimisations used.

Offutt and Pan [41, 42] introduced a constraint-based equivalent mutant detection approach. They evaluated it on 11 small programs and found that their approach could find approximately half of the equivalent mutants present. The results were reported to be superior to those achieved using the previously formulated compiler optimisation approach [39].

Program slicing has also been proposed as a way to detect equivalent mutants [23, 26, 52]. Voas and McGraw [52] were the first to suggest that slicing may be helpful, while Hierons et al. were the first to formulate such an approach, which they did in terms of both dependence analysis [23] and amorphous program slicing [26]. Their approach was evaluated using only case studies and remains unimplemented. Adamopoulos et al. [2] proposed a co-evolutionary approach to detect possible equivalent mutants, but this was only evaluated on simulations and not real mutants.

More recently, Grün et al. [22, 48] proposed to measure the impact of a mutant on execution, postulating that the greater the impact the greater the killability. By extension, their work could be thought of a means of approximating the likelihood of equivalence, rather than seeking to provide a partial (but exact) decision procedure, as had been attempted in previous work.

Schuler and Zeller [49] studied the correlation between the impact of various mutations, including impacts on coverage, return values and invariants. This approach can also be used as an approximate estimate of the likelihood of mutant equivalence.

Few authors have attempted to specifically and comprehensively determine which mutants are and which are not equivalent for a set of programs studied. Such an approach requires manual analysis and is tedious and time-consuming. We found ten previous papers in which some form of manual analysis is used to confirm results. A summary of this literature is presented in Table 4.

In Table 4, the first two columns give a reference to the previous work (authors and year), while the final two columns list the number of mutants that were manually examined[1] and the number reported to be equivalent. In the studies from before the year 2000, the programs were tiny. This observation is not intended as a criticism of this foundational work; available human and computing resources of the time did not permit anything else. The purpose of these studies was specifically to examine the prevalence of equivalent mutants and the effectiveness of techniques for detecting them automatically.

The more recent studies were able to include much larger programs, but only a tiny sample of mutants was examined manually. The purpose of this manual analysis was not to answer questions about equivalent mutants but rather, to assess the possible impact that their presence might have on the results reported.

None of the previous studies reports on the relationship between equivalent and stubborn mutants (according to *any* definition of stubbornness). It is this examination of relationships between stubborn and equivalent mutants that is the primary novelty of the present paper and from which we draw most of our actionable conclusions.

[1] The two approximated entries (indicated with a $\sim$ symbol in Table 4) were provided by a private communication with one of the authors of each of the two papers in question (Offutt and Patrick).

Table 4: Comparison to previous manual studies

Author(s) [Reference]	Year	#Studied	#Equivalent
Acree [1]	1980	90	25
Offutt & Craft [39]	1994	∼1000	255
Offutt & Pan [41, 42]	1994-7	695	695
Grün, Schuler & Zeller [22]	2009	20	8
Schuler & Zeller [49, 50]	2010-2	140	63
Patrick, Oriol & Clark [47]	2012	∼500	193
Kintis, Papadakis & Malevris [32]	2012	140	63
Just, Ernst & Fraser [30]	2013	17	9
This paper	2013	1230	946

8. CONCLUSIONS

We have presented a detailed manual study of equivalent mutants and their relationship to stubborn mutants. Equivalent mutants are those that are impossible to kill, while stubborn mutants are those that are killable, but prove hard to kill in practice. It might therefore be natural to assume that equivalence is merely an extreme case of stubbornness.

However, our results contradict this assumption. We found that, though equivalence is correlated with program size (and number of mutants created), stubbornness is not. We also found a correlation between the tendency for operators to generate equivalent and stubborn mutants only for the ROR operator class (for other operators there was no such correlation). In conclusion, though there is clearly some relationship between equivalence and stubbornness, the relationship is more subtle than might previously have been assumed.

Our findings also indicated that some mutation operators (for example ABS and post increment operators) should be used with caution or discarded. They appear to generate many equivalent and few stubborn mutants. We found that other operator classes (such as LCR) are more valuable. They tend to produce proportionately more stubborn, yet fewer equivalent mutants. These findings also applied to sub-classes of operators within the five categories, indicating that tool designers cannot afford to treat operator class members identically.

We believe that our findings may benefit the design of mutation testing tools; mutants that generate many stubborn, but few equivalent mutants can be prioritised for selection and generation, based on our findings. We also believe that our findings have implications for the validity of empirical and experimental studies of testing techniques that use fault seeding. The mutation scores reported in all such studies may have been adversely affected by the uneven distorting effects of equivalent mutants.

9. ACKNOWLEDGMENTS

Xiangjuan Yao is supported by National Natural Science Foundation of China (No.61203304), Natural Science Foundation of Jiangsu Province (Number BK2012566) and the Fundamental Research Funds for the Central Universities (Number 2012QNA41). Yue Jia is supported by the European Union FITTEST project FP7/ICT/257574 and by the EPSRC project EP/J017515 (DAASE). Mark Harman is additionally partly supported by the UK EPSRC projects EP/I033688 (GISMO) and EP/G060525, a 'Platform Grant' for the Centre for Research on Evolution Search and Testing (CREST) at UCL.

10. REFERENCES

[1] A. T. Acree. *On Mutation*. Phd thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.

[2] K. Adamopoulos, M. Harman, and R. M. Hierons. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'04)*, volume 3103 of *LNCS*, pages 1338–1349, Seattle, Washington, USA, 26th-30th, June 2004. Springer.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques and tools*. Addison Wesley, 1986.

[4] N. Alshahwan and M. Harman. State aware test case regeneration for improving web application test suite coverage and fault detection. In *International Symposium on Software Testing and Analysis (ISSTA 2012)*, pages 45–55, Minneapolis, MN, USA, 2012.

[5] A. Arcuri. It does matter how you normalise the branch distance in search based software testing. In *International Conference on Software testing (ICST 2010)*, pages 205–214, Paris, France, 2010. IEEE Computer Society.

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

[7] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[8] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, 16(2):1–32, 2007.

[9] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. In W. Tracz, M. P. Robillard, and T. Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. ACM, 2012. Article 45(1-10).

[10] L. Bottaci and E. S. Mresa. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, Dec. 1999.

[11] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Proceedings of the Summer School on Computer Program Testing*, pages 129–148, Sogesta, June 1981.

[12] T. A. Budd and D. Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18(1):31–45, March 1982.

[13] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In 33^{rd} *International Conference on Software Engineering (ICSE'11)*, pages 1066–1071, New York, NY, USA, 2011. ACM.

[14] D. Clark and R. M. Hierons. Squeeziness: An information theoretic measure for avoiding fault masking. *Information Processing Letters*, 112(8–9):335 – 340, 2012.

[15] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[16] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405 – 435, Oct. 2005.

[17] 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.

[18] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 416–419. ACM, September 5th - 9th 2011.

[19] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis (ISSTA 2010)*, pages 147–158, Trento, Italy, July 12-16, 2010, 2010. ACM.

[20] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis (ISSTA 2013)*, pages 302–313, Lugano, Switzerland, 2013. ACM.

[21] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[22] B. J. M. Grün, D. Schuler, and A. Zeller. The Impact of Equivalent Mutants. In *Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09)*, pages 192–199, Denver, Colorado, 1-4 April 2009. IEEE Computer Society.

[23] M. Harman, R. M. Hierons, and S. Danicic. The Relationship Between Program Dependence and Mutation Analysis. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 5–13, San Jose, California, 6-7 October 2001.

[24] M. Harman, Y. Jia, and B. Langdon. Strong higher order mutation-based test data generation. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 212–222, New York, NY, USA, September 5th - 9th 2011. ACM.

[25] M. Harman, M. Okunlawon, B. Sivagurunathan, and S. Danicic. Slice-based measurement of coupling. In R. Harrison, editor, 19^{th} *ICSE, Workshop on Process Modelling and Empirical Studies of Software Evolution*, Boston, Massachusetts, USA, May 1997.

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

[27] International Standards Organisation (ISO). International standards organisation: Programming

languages — C. International standard, ISO/IEC 9899: 1990 (E), Dec. 1990.

[28] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 249–258, Beijing, China, 2008. IEEE Computer Society.

[29] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649 – 678, September–October 2011.

[30] R. Just, M. D. Ernst, and G. Fraser. Using state infection conditions to detect equivalent mutants and speed up mutation analysis. In *Proceedings of the Dagstuhl Seminar 13021: Symbolic Methods in Testing*, volume abs/1303.2784, 2013.

[31] K. N. King and A. J. Offutt. A Fortran Language System for Mutation-Based Software Testing. *Software:Practice and Experience*, 21(7):685–718, October 1991.

[32] M. Kintis, M. Papadakis, and N. Malevris. Isolating first order equivalent mutants via second order mutation. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 701–710, Washington, DC, USA, 2012. IEEE Computer Society.

[33] K. Lakhotia, M. Harman, and H. Gross. AUSTIN: An open source tool for search based software testing of C programs. *Journal of Information and Software Technology*, 55(1):112–125, January 2013.

[34] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In 4^{th} *Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART'09)*, pages 95–104, Windsor, UK, 4th–6th September 2009.

[35] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software*, 83(12):2379–2391, 2010.

[36] L. J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[37] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. Di Penta. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *IEEE Transactions on Software Engineering*, 37(2):161–187, 2011.

[38] V. P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, 23(7):19–25, 1990.

[39] A. J. Offutt and W. M. Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, September 1994.

[40] 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.

[41] A. J. Offutt and J. Pan. Detecting Equivalent Mutants and the Feasible Path Problem. In *Proceedings of the 1996 Annual Conference on Computer Assurance*, pages 224–236, Gaithersburg, Maryland, June 1996. IEEE Computer Society Press.

[42] A. J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability*, 7(3):165–192, September 1997.

[43] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26:165–176, 1996.

[44] A. J. Offutt, J. Pan, and J. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the 12th International Conference on Testing Computer Software*, pages 111–123, June 1995.

[45] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In 15^{th} *International Conference on Software Engineering (ICSE 1993)*, pages 100–107. IEEE Computer Society Press, Apr. 1993.

[46] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10)*, California, USA, November 2010.

[47] M. Patrick, M. Oriol, and J. A. Clark. MESSI: Mutant evaluation by static semantic interpretation. In *Proceedings of the 7th International Workshop on Mutation Analysis (MUTATION'12)*, Montreal, Canada, 17 April 2012. IEEE Computer Society.

[48] D. Schuler, V. Dallmeier, and A. Zeller. Efficient Mutation Testing by Checking Invariant Violations. Technique report, Saarland University, Saarbrucken, Telefon, 2009.

[49] D. Schuler and A. Zeller. (Un-)Covering Equivalent Mutants. In *Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST'10)*, Paris, France, 6 April 2010. IEEE Computer Society.

[50] D. Schuler and A. Zeller. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability*, 23(5):353–374, 2013.

[51] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In T. Ostrand and E. Weyuker, editors, *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 139–148, 1993.

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

[53] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, July 1988.

[54] Y. Zhan and J. A. Clark. Search-based mutation testing for simulink models. In H.-G. Beyer and U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference (GECCO 05)*, pages 1061–1068, Washington DC, USA, June 2005.