Genet Program Evolvable Mach (2014) 15:281–312
DOI 10.1007/s10710-013-9195-8

Software mutational robustness

**Eric Schulte · Zachary P. Fry · Ethan Fast ·
Westley Weimer · Stephanie Forrest**

Received: 30 November 2012 / Published online: 28 July 2013

Abstract Neutral landscapes and mutational robustness are believed to be important enablers of evolvability in biology. We apply these concepts to software, defining *mutational robustness* to be the fraction of random mutations to program code that leave a program's behavior unchanged. Test cases are used to measure program behavior and mutation operators are taken from earlier work on genetic programming. Although software is often viewed as brittle, with small changes leading to catastrophic changes in behavior, our results show surprising robustness in the face of random software mutations. The paper describes empirical studies of the mutational robustness of 22 programs, including 14 production software projects, the Siemens benchmarks, and four specially constructed programs. We find that over 30 % of random mutations are neutral with respect to their test suite. The results hold across all classes of programs, for mutations at both the source code and assembly instruction levels, across various programming languages, and bear only a

E. Schulte (✉) · S. Forrest
Department of Computer Science, University of New Mexico, Albuquerque, NM, USA
e-mail: eschulte@cs.unm.edu

S. Forrest
e-mail: forrest@cs.unm.edu

Z. P. Fry · W. Weimer
Department of Computer Science, University of Virginia, Charlottesville, VA, USA
e-mail: zpf5a@cs.virginia.edu

W. Weimer
e-mail: weimer@cs.virginia.edu

E. Fast
Department of Computer Science, Stanford University, Palo Alto, CA, USA
e-mail: ethaen@stanford.edu

S. Forrest
Santa Fe Institute, Santa Fe, NM, USA

limited relation to test suite coverage. We conclude that mutational robustness is an inherent property of software, and that neutral variants (i.e., those that pass the test suite) often fulfill the program's original purpose or specification. Based on these results, we conjecture that neutral mutations can be leveraged as a mechanism for generating software diversity. We demonstrate this idea by generating a population of neutral program variants and showing that the variants automatically repair latent bugs. Neutral landscapes also provide a partial explanation for recent results that use evolutionary computation to automatically repair software bugs.

Keywords Mutational robustness · Genetic programming · Mutation testing · Proactive diversity · N-version programming · Neutral landscapes

1 Introduction

The ability of biological organisms to maintain functionality across a wide range of environments and to adapt to new environments is unmatched by engineered systems. Understanding the intertwined mechanisms and evolutionary drivers that have led to the robustness and evolvability of biological systems is an important subfield of evolutionary biology. In this paper we focus on neutral landscapes and mutational robustness, applying these concepts to software.

Today's software arose through fifty years of continued use, appropriation, and refinement by software developers. The tools, design patterns and codes that we have today are those that have proven useful and were robust to software developer's edits, hacks and accidents, and those that survived the economic pressures of the marketplace. We hypothesize that these evolutionary pressures have caused software to acquire mutational robustness resembling that of natural systems. Mutational robustness in biological systems is believed to be intimately related to the capacity for unsupervised evolution and adaptation. We posit that software mutational robustness points to the potential for powerful methods of unsupervised software enhancement and evolution.

Robustness is important in software engineering, especially as it relates to reliability, availability or dependability. Here we focus on genetic robustness, defining software *mutational robustness* in terms of changes to computer code. In this context, we define a *neutral mutation* to be a random change applied to a program representation (source code, abstract syntax tree, assembly, binary, etc.) such that the mutated program's behavior remains unchanged on its regression test suite.[1] Thus, software fitness is assessed by the program's performance on a set of test cases. In Sect. 6.1 we discuss the use of test suites to assess software fitness. In Sect. 3.1 we present a formal definition of software mutational robustness as the fraction of mutations to a program which do not change its correctness as assessed by a set of regression tests.

Software mutational robustness measures the fraction of software mutants that are neutral. Neutral variants are equivalent to the original program with respect to

[1] This definition is not to be confused with the "equivalent mutants" of mutation testing, see Sect. 2.3.1.

the test suite. They may or may not be semantically equivalent (compute the same function) to the original program, they may or may not have the same non-functional properties (run-time, memory consumption, etc.), and they may or may not satisfy the specification (required behavior) of the original designers. Empirically, we find that the program's test suite is an acceptable proxy for the program specification. We find many neutral variants that are both semantically distinct from the original program and still satisfy the original program's specification or intended behavior.

This result can be understood more easily when one considers that there are an infinite number of ways to encode any algorithm in software. For example, consider this fragment of a recursive quick-sort implementation:

```
if (right > left) {
  // code elided ...
  quick(left, r);
  quick(l, right);
}
```

Swapping the order of the last two statements to `quick(l, right);` `quick(left, r);` changes the run-time behavior of the program without changing the output, giving an alternate implementation of the specification. We find that neutral mutations are prevalent in software and contribute to evolvability, as discussed in Sect. 4.4.

Our mutation operators (delete, copy, and swap) are described in detail in Sect. 3. They are notable because they do not create new code de novo. Delete and copy are both plausible analogs of genetic operations on DNA, and all three are edit operations that are routinely performed by programmers. They are also related to operators commonly used in the genetic programming community, although we note that we do not have an explicit terminal set as typical in genetic programming. In effect, our terminal set corresponds to all of the statements contained in the program being studied.

We are interested in the extent to which mutational robustness enables software evolvability by which we mean the use of automated methods for software development and maintenance. In particular, there is increasing interest in automatic program repair, and many of the more promising approaches rely on unsound program transformations (Sect. 2.4.1). These may involve both source-level edits (e.g., [16, 36, 92, 93]) and modification to program state at run-time (e.g., [66]). Mutational robustness may help explain why program transformations, such as swapping two statements [92] or clamping an integer value [66], can produce acceptable program behavior.

The primary contributions of the paper include:

1. The empirical measurement of software mutational robustness in a large collection of off-the-shelf software, demonstrating that mutational robustness is prevalent. We find largely uniform mutational robustness scores with an average value of 36.8 % and a minimum across all software instances of 21.2 %. We evaluate this claim using 22 programs involving over 150,000 lines of code and 23,151 tests.

2. An application of software mutational robustness to repair unknown software defects proactively. As an illustration, we seeded bugs into 11 programs, generated populations 5,000 of neutral variants using mutation, and studied the behavior of the variants on the seeded bugs using test cases withheld during neutral variant generation. In eight of the programs, the neutral population contained at least one variant that "repaired" the latent bug, passing the withheld test case.
3. An alternate interpretation of the software engineering technique known as mutation testing to include neutral mutations and software robustness. We discuss the relation between software functionality, software test suites and specifications, demonstrating the value of neutral mutations, both for proactively repairing unknown bugs and as a likely enabler of automated software evolution techniques.

In the remainder of the paper, we review related work in Biology and Software Engineering in Sect. 2. We then describe our software representations and mutation operators in Sect. 3. The experimental design and experimental results are given in Sect. 4. We present a practical application of software mutational robustness in Sect. 5. Finally, we analyze our results and discuss potential threats to validity and implications in Sects. 6 and 7.

2 Background

The previous work most closely related to software mutational robustness includes work on neutral theories in biology, investigations of the effect of neutrality in fitness landscapes in evolutionary computation and the field of mutation testing in software engineering. In the following three subsections we highlight some of the most relevant aspects of these fields.

2.1 Biology

Biological evolution is understood in terms of the interplay between genotype and phenotype. The genotype is the informational representation that specifies the organism, and the phenotype is the physical appearance and behavior of organisms interacting with their environment. There is a corresponding type of robustness for each of these levels of description: *mutational robustness* and *environmental robustness* respectively [45]. Mutational robustness is the organism's ability to maintain phenotypic traits in the face of internal genetic mutations, and environmental robustness is its ability to maintain functionality across a wide range of environments [87].

The two types of robustness are closely related. Many of the causes of mutational robustness are also causes of environmental robustness [52]. It is thought that the pervasive mutational robustness observed in biological systems may have arisen as a by-product of evolutionary pressure for environmental robustness [54]. However mutational robustness has been shown to be beneficial in its own right, especially in its impact on an organism's evolvability [13, 63].

Over time, populations of organisms accumulate mutations in their genome. Of the many mutations that occur in a single individual, only a tiny fraction spread to *fixation* in the population. Mutations accumulate at a fairly constant rate known as the *genetic clock* [98]. Initially, only those mutations which increased fitness were thought to become fixed in the population, an idea known as selectionism [21]. In 1968 Kimura suggested that because populations have finite size, the majority of accumulated mutations might be effectively neutral, with no impact upon fitness [42]. Kimura noted that as a consequence of neutral mutation, "we must recognize the great importance of random genetic drift due to finite population number in forming the genetic structure of biological populations."

Recent work [19] estimates that roughly 50 % of the fixed mutations provide a selective advantage in Drosophila fruit flies, which have effective population sizes on the order of 10^6, while in hominids, with effective populations sizes of 10^4, a tiny percentage of fix mutations were adaptive. In this study, roughly 16 % of non-equivalent mutations in Drosophila were found to be effectively neutral compared to roughly 30 % of non-equivalent mutations in hominids.

The variants of an organism produced through neutral mutations are called "neutral neighbors." Connected sets of neutral neighbors are called "neutral spaces" [43] and can occupy sizeable regions of an organism's fitness landscape [74]. Mutational robustness and the resulting neutral spaces in fitness landscapes are believed to contribute to a population's ability to evolve [32].

Mutational drift through neutral spaces gives populations access to new phenotypes located along the mutational border of the neutral space. Neutral spaces thus allow populations to increase diversity and to accumulate new genetic material through drift. This accumulation of genetic material has been shown to be required for large evolutionary innovations [13, 55, 63, 88].

In effect, mutational robustness of an organism is a metric of its fitness landscape. A number of metrics of fitness landscapes have been devised in an attempt to statistically characterize landscapes [29] and to directly measure those properties of a landscape that encourage evolution [80]. Our proposed software mutational robustness begins the work of applying such metrics to real-world software.

2.2 Evolutionary computation

The role of neutrality in evolutionary computation has been explored in several specific contexts. In simple GP systems whose fitness landscapes have similarities to those of RNA, neutral mutations have been shown to enhance exploration in evolutionary searches [6]. Neutrality was shown to be beneficial for evolving digital circuits, both in retrospective analysis of successful experiments [28], and in directed experiments using synthetic fitness landscapes designed with variable amounts of neutrality [85]. Some GP methods such as Cartesian genetic programming [56] have been explicitly designed to leverage neutrality in genetic search.

Using a population genetics model, varying levels of mutational robustness have been shown to either inhibit or encourage evolution, depending on population size, mutation rate, and fitness landscape [18]. Some studies (e.g., using a GA to optimize robot movement [81]) suggest that with certain complex genotype-phenotype mappings, periods of neutral evolution do not measurably increase a population's evolvability. Although none of this prior work studies neutrality in software per se, it does suggest that success in evolving software (e.g., repairing bugs) may be related to neutral landscapes in the space of program representations.

2.3 Software engineering

Three subfields of software engineering, namely mutation testing, n-version programming, and program transformation are most relevant to this work.

2.3.1 Mutation testing

The software engineering community has studied randomly generated program mutants for over 30 years under the mantle of "Mutation Testing"; however, the interpretation and use of program mutants has been limited to measuring test suite adequacy. In their landmark review of mutation testing Jia and Harmon [35] describe the field as follows.

> Mutation Testing is a fault-based testing technique which provides a testing criterion called the "mutation adequacy score." The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect [mutants] faults.

In mutation testing, non-equivalent mutants are presumed to indicate a fault (either in the mutant or the original program). Thus, mutants that pass a program's test suite indicate test suite failures and lower the test suite's "mutation adequacy score." Mutation testing recognizes the existence of "equivalent mutants" which are semantically identical to the original program (cf. the *Equivalent Mutant Problem* [12]) and viewed as problematic for the mutation testing paradigm. The mutation testing literature, however, does not recognize the possibility of valid neutral mutants; i.e., mutants that are semantically distinct from the original program but still satisfy the original program's specification (and its test suite).

Figure 1 shows the syntactic space surrounding a program. This is similar to a fitness landscape; each point in the space represents a syntactically distinct program, and each program is associated with a semantic interpretation although that is not shown in the figure. Randomly mutating a program's syntactic representation can have several possible semantic effects, which are shown in the figure.

Our results are based on the following insight: for every specification there exists multiple non-equivalent correct implementations. This emphasizes a different view of software than is implicit in the mutation testing technique, namely, that for every program specification all correct implementations are semantically equivalent.

To see how this follows from mutation testing, assume that $\exists$ programs a and b s.t. a is not equivalent to $b(a \not(-1\,\mathrm{cm} : 1.5\,\mathrm{cm})circle(1\,\mathrm{cm})b)$ and both a and

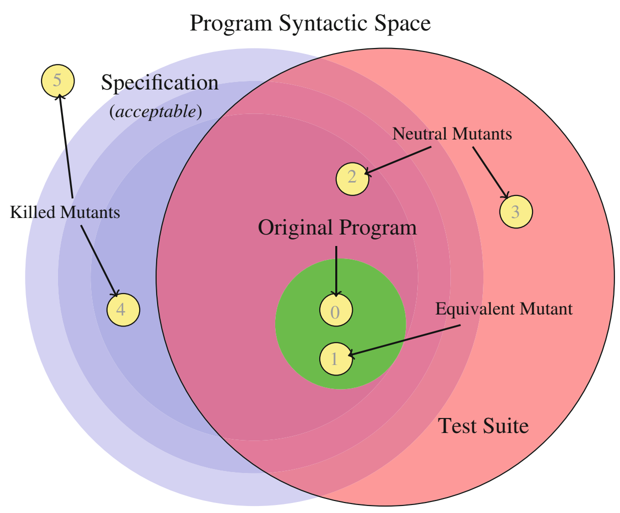


Fig. 1 Syntactic Space of a Program. The set of programs satisfying the program specification are *shaded blue* (*left*), the set of programs passing the program's test suite are *shaded red* (*right*), and the set of equivalent programs are shown in *green* (*center*). The relative size of these spaces and of their intersection is unknown. Three classes of mutants are shown and labeled. This work leverages the existence of valid neutral mutants such as 2 which lie in the intersection of the specification and the test suite (Color figure online)

b satisfy specification S. Without loss of generality let *a* be the original program and *b* be a mutant of *a*. Let T be a test suite of S. According to Offut [60] there are two possibilities when T is applied to *b*. Either "the mutant is killable, but the test cases is insufficient" or "the mutant is functionally equivalent." The former case is impossible because b is assumed to be a correct implementation of S and thus should not be killed by any test suite of S. The later case is impossible because we assume $a \not(-1\,\text{cm} : 1.5\,\text{cm})\mathit{circle}(1\,\text{cm})b$. By contraction, ∀ *a* and *b* satisfying the same specification S, *a* (− 1 cm:1.5 cm) *circle* (1 cm) *b* or ∀ specification S ∃! *a* s.t. a satisfies S.

Taking this perspective, by requiring test suites to kill all non-equivalent neutral mutants, mutation testing could lead to test suites that are more restrictive than the specification. To illustrate this point, consider the specification and programs in Fig. 2. Programs *a* and *b* both satisfy the specification *S*, yet they are not equivalent (notably on many four element arrays). Any test suite for *S* which kills one of these implementations will be overly restrictive.

In the following sections, we study non-equivalent neutral mutants and investigate their relative frequency in our benchmark programs. Our study extends work on mutation testing by emphasizing non-equivalent neutral mutations (cf. neutral mutants 2 and 3 in Fig. 1 which behave identically to the original program with respect to a given test suite), discussing how they can be leveraged, and by providing a biological interpretation for the phenomenon of neutral mutations.

```
/*
 * Spec (S):
 *    Pre: parameter P is an array of three integer elements
 *    Post: returns the smallest of the three input elements
 */

int a(int p[]) {
  if (p[0] <= p[1] && p[0] <= p[2]) return p[0];
  if (p[1] <= p[2] && p[1] <= p[0]) return p[1];
  else return p[2];
}

int b(int p[]) {
  sort(p, "ascending");
  return p[0];
}
```

Fig. 2 Specification *S* and two correct, non-equivalent implementations

2.4 N-version programming

There has been considerable research on the use of automated diversity in security, for example, the special issue of *IEEE Computer Security* devoted to IT Monocultures [33]. Common mechanisms for introducing diversity include Address Space Randomization [10, 23] and Instruction Set Randomization [9, 40], among many others. In these applications, diversity is introduced to reduce the risk of widely replicated attacks, by forcing the attacker to redesign the attack each time it is applied [96]. Our proposed use of diversity, outlined in Sect. 5, is closer in spirit to *n*-variant systems [15], where multiple variants of a program are run in parallel, giving each variant identical inputs and checking that they all behave similarly before forwarding the output to the user.

Our proposed application also resembles *n*-version programming [53] where multiple independent, or quasi-independent, manually written implementations of critical programs reduce the risk of implementation errors going undetected. Our approach differs from *n*-version programming: Instead of relying on teams of human programmers to generate full implementations independently, we use lightweight mutation operators to automatically generate variants that are semantically similar to the original. This addresses the cost issue identified in earlier studies [46] and potentially addresses the assumption that independent teams of humans are likely to generate programs that will fail independently; studies suggest that this assumption does not hold in practice [48]. Because we generate variants automatically, there is a better chance of achieving independence among the variations, either with the mutation operators we describe here or with others to be developed in the future.

2.4.1 *Unsound program transformation*

Traditionally, automated program transformation techniques (e.g., compiler optimizations) refrain from altering the semantics (or behavior) of the original program. Such program transformations are “sound” because they are guaranteed to preserve program semantics. Recent work has experimented with “unsound” program transformations, which do not necessarily preserve the exact semantics of the original program.

Self-healing systems [26] acknowledge that proactive developer efforts at providing fault tolerance and reliability are often insufficient, suggesting instead the idea of *reactive* methods of runtime protection. For example, the Assure system [77] adds rescue points to software which catch otherwise fatal errors, and handle them by re-purposing error handling code already present in the application. Similarly, *failure oblivious* computing [68], handles common memory errors, such as out-of-bounds reads and writes, by simply ignoring them or automatically re-mapping invalid memory addresses to arbitrary valid memory address, allowing the program to continue execution. Such trade-offs of correctness for robustness are desirable for applications such as web servers where availability is paramount.

Loop perforation [57] is a run-time method that sacrifices exact program accuracy in return for reduced running time or energy consumption. Loop perforation eliminates some of the computation specified in the original program by dropping some iterations of selected program loops.

Another important class of unsound program transformations are automated repair techniques. These techniques share a common approach: defining a notion of correct and incorrect program behavior (e.g., from test cases [66, 92], implicit specifications [36], or explicit specifications [93]); generating a set of candidate repair transformations (e.g., at random [92], by constraint solving, or from an established set [36, 66]), and validating the candidates produced by the transformations until a suitable repair is found.

Evolutionary methods have been widely used in this work, both to repair bugs seeded into constructed programs using a subsets of the Java programming language [5] and to repair real programs written in C [50, 92], including a systematic study of over 100 bugs mined from open-source software repositories [51]. Recent work emphasizes evolving software patches directly, rather than complete program representations [2, 51], evolving repairs at the assembly and binary code levels [72, 73], and evolving the Java Byte code compiled from extant programs using specialized mutation operators including a technique of *compatible crossover* [64, 65]. Finally, Wilkerson used GP to co-evolve C++ applications in competition with sets of test cases [95].

3 Technical approach

In this section, we define *software mutational robustness*, describe the program representations used in our experiments, and for each representation specify the representation-specific mutation operations.

3.1 Software mutational robustness

We formalize software mutational robustness with respect to a software program P (a member of the set of all software programs $\mathcal{P}$), a set of mutation operators

M (where each $m \in M$ is a function mapping $\mathcal{P} \rightarrow \mathcal{P}$), and a test suite $T : \mathcal{P} \rightarrow \{\mathsf{true}, \mathsf{false}\}$. A program P is said to pass the test suite iff $T(P) = \mathsf{true}$.

Given a program P, a set of mutation operators M, and a test suite T such that $T(P) = \mathsf{true}$, we define the *software mutational robustness*, written $MutRB(P, T, M)$, to be the fraction of all direct mutants $P' = m(P)$, $\forall m \in M$ which both compile and pass T:

$$MutRB(P,T,M) = \frac{|\{P' | m \in M.P' = m(P) \wedge T(P') = \mathsf{true}\}|}{|\{P' | m \in M.P' = m(P)\}|}$$

Based on this definition, software mutational robustness depends on three parameters, P, T and M. Perhaps surprisingly, the empirical results of Sect. 4 show that software robustness does not depend strongly on P or T for human-constructed software systems. Our mutation operators M, described below, are adapted from genetic programming and are simple and natural analogs of biological mutation. We believe that they are also general and appropriate to software. For example, earlier work has shown that the set M, together with crossover, is sufficiently strong to generate successful repairs for a wide variety of defects in a wide variety of software [51, 50, 92]. Additionally, they reflect common human edit operations.

3.2 Representation and operators

We consider two levels of program representation: abstract syntax trees (AST) based on high-level program source code, and low-level assembly code (ASM). We use the CIL toolkit [58] to parse and manipulate ASTs of C source code. CIL simplifies some C constructs to facilitate manipulation by computational tools and supports source to source translations such as our mutation operations. The ASM representation is the linear sequence of instructions taken directly from the compiled `.s` assembly code file produced by standard compilation (i.e., “gcc -O2 -S”) on a 64-bit Intel platform, which is split on line breaks [73], but with directives and other pseudo-operations protected from mutation. Our choice of one tree-based and one linear representation increases our confidence that the results do not depend on such representation details. For example, our AST representation is at the statement level. That is, each node in the AST corresponds to a legal statement in C. This relatively coarse representation level provides a distinct contrast to the fine-grained ASM representation, where approximately three assembly statements represent each C statement.

Given a source code or assembly language program, we consider three simple language-independent mutation operators: *copy*, *delete* and *swap*. Copy duplicates an AST statement-level subtree or assembly instruction and inserts it at a random position in the AST or immediately after a randomly chosen instruction. Delete removes a randomly chosen statement-level AST subtree or assembly instruction. Swap exchanges two randomly chosen statement-level AST subtrees or assembly instructions. Figure 3 illustrates these operators. Because AST mutations manipulate subtrees, a large amount of code might be inserted or deleted by a single mutation, depending on how high in the tree the mutation is applied. In the experiments, mutations modify only AST statements or ASM instructions that are actually visited by the test suite. This restriction is similar to mutating only those parts of genome that are known to be involved in the phenotype being assayed. Mutations to untested statements would likely be neutral under our metric, unfairly biasing the results towards overly high estimates of mutational robustness.

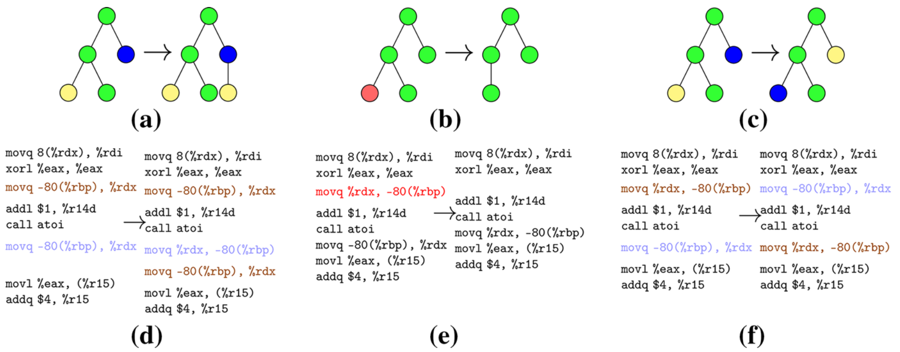
4 Experimental results

We report results for five experiments on the mutational robustness of programs in both representations (AST and ASM). Throughout this section we remain conscious of the threat that our results may measure the poor quality of program test suites instead of the intrinsic mutational robustness of software. Much of our experimental design addresses this particular concern.

We investigate: (1) the level of mutational robustness of several computer programs (we specifically select benchmark programs with the highest quality test suites available); (2) the extent to which mutational robustness depends on or can be explained by test suite quality; (3) a taxonomy of neutral mutations (indicating which mutations would no longer remain neutral under perfect testing), (4) the effect of cumulative neutral mutations mutations in a simple program, and (5) generality of mutational robustness across multiple programming languages and paradigms.

4.1 Benchmark programs

We selected 22 programs for our experiments (Table 1). Fourteen are off-the-shelf programs selected to measure mutational robustness in real-world software. Four are taken from the Siemens Software-artifact Infrastructure Repository, created by Siemens Research [31] and later modified by Rothermel and Harold [70] until each “executable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 tests.” The `space` test suite, which was generated by Vokolos [86] and later enhanced by Graves [27], which covers every edge in the control flow graph with at least 30 tests. These programs are



(d)
```
movq 8(%rdx), %rdi        movq 8(%rdx), %rdi
xorl %eax, %eax           xorl %eax, %eax
movq -80(%rbp), %rdx      movq -80(%rbp), %rdx
addl $1, %r14d            addl $1, %r14d
call atoi             →   call atoi
movq -80(%rbp), %rdx      movq %rdx, -80(%rbp)
                          movq -80(%rbp), %rdx
movl %eax, (%r15)         movl %eax, (%r15)
addq $4, %r15             addq $4, %r15
```

(e)
```
movq 8(%rdx), %rdi        movq 8(%rdx), %rdi
xorl %eax, %eax           xorl %eax, %eax
movq %rdx, -80(%rbp)
addl $1, %r14d            addl $1, %r14d
call atoi             →   call atoi
movq -80(%rbp), %rdx      movq %rdx, -80(%rbp)
movl %eax, (%r15)         movl %eax, (%r15)
addq $4, %r15             addq $4, %r15
```

(f)
```
movq 8(%rdx), %rdi        movq 8(%rdx), %rdi
xorl %eax, %eax           xorl %eax, %eax
movq %rdx, -80(%rbp)      movq -80(%rbp), %rdx
addl $1, %r14d        →   addl $1, %r14d
call atoi                 call atoi
movq -80(%rbp), %rdx      movq %rdx, -80(%rbp)
movl %eax, (%r15)         movl %eax, (%r15)
addq $4, %r15             addq $4, %r15
```

Fig. 3 Mutation operators: Copy, Delete, Swap

Table 1 Mutational robustness of benchmark programs

Program	Lines of code		Test suite		Mut. robustness	
	ASM	C	# tests	% stmt.	AST	ASM
Sorting algorithms						
bubble-sort	184	34	10	100	27.3	25.7
insertion-sort	170	29	10	100	29.4	26.0
merge-sort	233	38	10	100	29.8	21.2
quick-sort	219	38	10	100	28.9	25.5
Siemens [31][a]						
printtokens	2419	536	4130	81.7	21.2	25.8
schedule	922	412	2650	94.4	34.4	29.1
space	18098	9126	13494	91.1	37.7	32.1
tcas	544	173	1608	96.2	33.5	25.9
Systems programs						
bzip2 1.0.2	18756	7000	6	35.9	33.0	26.1
— *(alt. test suite)*			22	71.0	46.4	23.6
ccrypt 1.2	15261	4249	6	29.5	33.0	69.7
— *(alt. test suite)*			16	40.4	34.6	69.7
grep	28776	10929	119	24.9	50.0	36.7
imagemagick 6.5.2	6128	147	145	0.8	33.3	66.3
jansson 1.3	6830	2975	30	28.8	33.3	28.0
leukocyte	40226	7970	5	45.4	33.3	39.9
lighttpd 1.4.15	34165	3829	11	40.1	61.5	56.9
nullhttpd 0.5.0	5951	5575	6	64.5	41.5	37.8
oggenc 1.0.1	299959	59094	10	38.4	33.4	22.1
— *(alt. test suite)*			40	58.8	40.5	72.3
potion 40b5f03	80406	15033	204	48.4	33.3	48.9
redis 1.3.4	44802	17203	234	9.2	33.3	34.0
sed	17026	8059	360	42.0	33.0	25.6
tiff 3.8.2	22458	1732	10	15.4	33.3	90.4
vyquon 335426d	20567	4390	5	50.6	33.3	69.0
Total or average	664100	158571	23151	40.9	33.9 ± 10	39.6 ± 22

Lines of Code" columns report the size of the program in terms of lines of C source code and lines of compiled assembly code. The "Test Suite" columns show the size of the test suite both in terms of number of test cases and the percentage of all AST level statements in the program that are exercised by the test suite. The "Mut. Robustness" columns report the percentage of all first-order mutations that were neutral. The ± values in the bottom row indicate one standard deviation. For each program, at both the AST and ASM level we generated at least 200 unique variants using each of the three mutation operations (*copy*, *delete* and *swap*). Mutation operations were applied at locations chosen randomly from all those visited by the test cases. For three programs (`bzip`, `ccrypt` and `oggenc`) we also evaluated on three independent alternate test suites

[a] Although the Siemens benchmark suite claims complete branch and statement coverage, we find <100 % statement coverage. This is due to our use of finer-grained Cil statements in calculating coverage. See Sect. 3.2 for discussion of the Cil program representation

included for comparability to previous research and to study the robustness of programs with extremely high-quality test suites. We include four simple sorting algorithms taken from http://rosettacode.org to demonstrate the robustness of programs with full statement, branch-level and assembly instruction test coverage.

Each program has an associated test suite. The tests either came with the program as part of its established regression test (e.g., Siemens, `potion`, `redis`, `jansson`) or were constructed manually (e.g., sorting algorithms, webservers). A number of our benchmarks implement invertible transformations (e.g., compression, encryption, serialization, image manipulation), which form an implicit formal specification and permit simple testing [90]. The three invertible programs (`bzip`, `ccrypt` and `oggenc`) are thus each evaluated on two independently constructed test suites. For `lighttpd` and `imagemagick`, we restricted mutations to `mod_fastcgi.c` and `convert.c` respectively, demonstrating application to modules as well as to whole systems.

4.2 Software mutational robustness

We first demonstrate that a variety of software programs exhibit significant mutational robustness under the mutation operators described in Sect. 3. In this experiment, we measure the percentage of random mutations to code visited by at least one test case which leave the program's behavior unchanged on all of its test cases. In every case, the initial program passes all of its test cases. For each mutation operator we generate program variations making at least 200 copies of the original program and then applying a single random mutation to each copy. We refer to these as *first-order*, mutations. We then run the mutated program on its test suite and count it as neutral if it passes all of its tests.

Table 1 shows the results of this experiment on the benchmark programs. We wish to rule out trivial mutations, such as the insertion of dead code (e.g., statements that appear after a `return`) or the transposition of independent lines, that are visible in the source code but would produce equivalent assembly code. Since program equivalence is undecidable [12], we approximate this by compiling the AST using "gcc -O2 -S," which includes dead code elimination, SSA form, and instruction scheduling. Multiple source code variants that produce the same optimized assembly code (modulo label names and other directives) are counted only once in Table 1. Similarly, any two ASM-level mutations which produce the same executable are counted only once.

Although the results vary by program (e.g., `grep` is more robust than `printtokens`), the results show a remarkably high level of mutational robustness: Across all programs, operators, and representations (source or assembly) 36.8 % of variants continue to pass all test cases with no systematic difference between the AST and ASM representation. In the next two subsections we ask to what extent these results arise from inadequate test suites (4.3) or from semantically equivalent mutations (4.4).

4.3 Does mutational robustness depend on test suite quality?

The results in Table 1 are striking, but they could potentially be entirely the result of inadequate test suites. In Fig. 4 we plot the data from Table 1, showing the mutational robustness of software grouped by the quality of the software test suite. We consider both the quantitative metrics of code coverage of these test suites, as well as qualitative differences between the program test suites by panel.

Qualitatively the program classes in the three panels of Fig. 4 have very different test suites.

Sorting program suites all share a single test suite. This test suite leverages the simplicity of sorting programs to provide complete code coverage (at both the statement and assembly instruction levels) with only 10 test cases.

Siemens program suites taken from the testing community where they have been developed by multiple parties across multiple publications [31, 70] until each

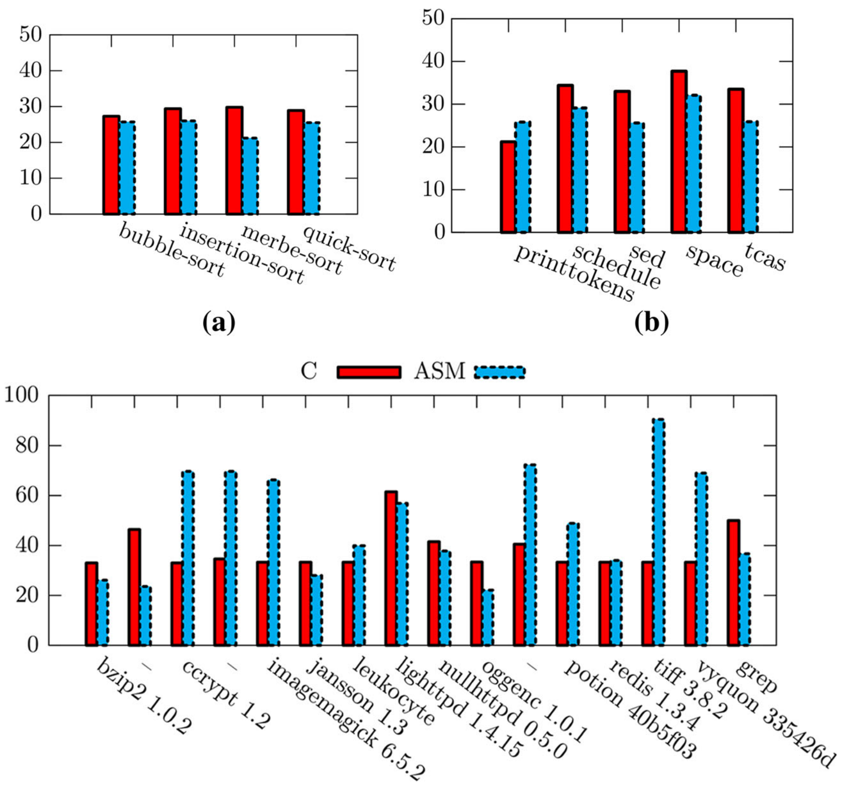


Fig. 4 Mutational robustness shown by quality of test suite. The simple sorting programs in **a** have complete code and ASM instruction coverage with few carefully constructed test cases. The Siemens programs in **b** have complete branch and def-use pair coverage and thousands of test cases for relatively small programs. The Systems programs in **c** have test suites that vary greatly in quality

"executable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 tests."

Systems program suites are taken directly from real world software development projects. These test suites reflect the great range of test suites used in practice.

Quantitatively, the sorting programs in Panel 4a have 100 % code coverage and the Siemens benchmark programs in Panel 4b have close to 100 % code coverage. By contrast the Systems programs in Panel 4c have 37.63 ± 19.34 coverage. Despite this wide range of coverage, the average mutational robustness varies relatively little between Panels at 26.7 % (Panel 4a), 29.8 % (Panel 4b) and 43.7 % (Panel 4c) respectively, and the minimum mutational robustness measurements for each Panel are even closer at 21.2, 21.2 and 22.1 % respectively.

The lack of correlation between code coverage and mutational robustness is not surprising, both because we explicitly limit the mutation operators to code that is visited by test suites, and because statement coverage is known to be an insufficient metric of test suite quality [30].

In fact, the lack of correlation between mutational robustness and test suite quality is not true in either limit. At one extreme, even high-quality test suites (such as the Siemens benchmarks, which were explicitly designed to test all execution paths) and test suites with full statement, branch and assembly instruction coverage have over 20 % mutational robustness. At the other extreme, a minimal test suite that we designed for bubble sort, which does not check program output but requires only successful compilation and execution without crash, has only 84.8 % mutational robustness.

These results show that, in practice, for both real programs and comprehensively tested ones, mutational robustness is not fully explained by test suite quality or inadequacy.

4.4 Taxonomy of neutral variants

To provide insight into our results, we chose bubble sort as an example of an easy-to-understand and easy-to-test program. We then studied the effect of 35 first-order neutral mutations on bubble sort at the AST level, developing a taxonomy of the neutral mutations. After manual review, all 35 neutral variations were confirmed to be valid implementations of the sorting specification. We next categorized them with respect to their operational differences from the original. The results are shown in Table 2.

These categories have different effects on program execution. Only categories 1 and 5 affect the externally observable behavior of the program by changing output and return values in ways not specified by the program specification. Categories 2, 3, 4 and 7 may affect the running time of the program. Category 2 includes the removal of unnecessary variable assignments, re-ordering non-interacting instructions and changing state which is later overwritten or never again read. Many changes, such as types 2, 3 and 4, produce programs that will likely be more robust to further manipulation by inserting redundant (occasionally diverse) control flow guards (i.e., conditionals that control if statements) and variable assignments.

Table 2 Taxonomy of 35 neutral first order AST variations of bubble sort

#	Functional category	Frequency/ 35
1	Different whitespace in output	12
2	Inconsequential change of internal variables	10
3	Extra or redundant computation	6
4	Equivalent or redundant conditional guard	3
5	Switched to non-explicit return	2
6	Changed code is unreachable	1
7	Removed optimization	1

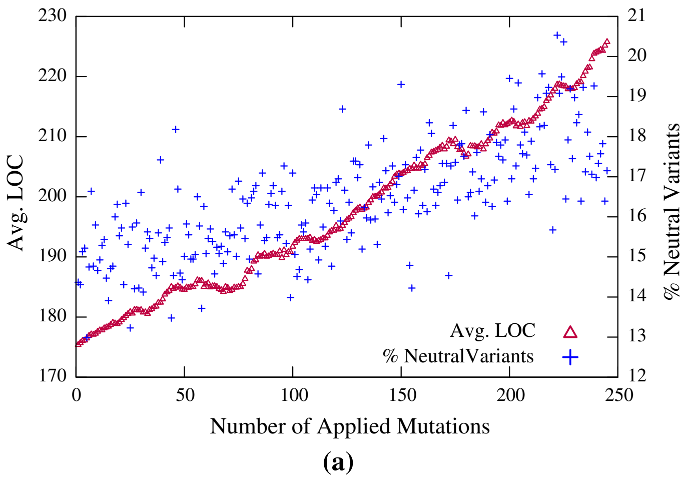
Categorized by manual review. "Different whitespace in output" describes variants whose output differs from the original program only in whitespace characters which are not detected by the test suite. "Inconsequential change of internal variables" describes variants whose mutations change the behavior of the program while executing but do not change the tested output of the program, e.g., mutations which change the values of variables in memory which do not later affect program output or variants which change the ordering of non-interacting instructions

Across most of these categories we find alternative implementations that are *not* semantically equivalent to the original but which conform to the program specification.

Of these classes of neutral programs, only classes 4, 5 and 6 could possibly have no impact on runtime behavior and could possibly be equivalent. Under the mutation testing paradigm, tests would be constructed to "kill" the remaining 29 of 35 neutral mutants. Given the sorting specification used (namely to print whitespace separated integer inputs in sorted order to `STDOUT` separated by whitespace), *none* of these classes of neutral mutations could be viewed as faulty implementations. Consequently, any such extra tests constructed to tell them apart (e.g., for mutation testing) would *over-specify* the program specification. Rather than improving the test suite quality, such over-constrained tests would potentially judge future correct implementations as faulty.

4.5 Cumulative robustness

The previous experiments measured the percentage of first-order mutations that are neutral. This subsection explores the effects of accumulating cumulative neutral mutations in a small assembly program. We begin with a working assembly code implementation of insertion sort. We apply random mutations using the ASM representation and mutation operations defined in Sect. 3.2. After each mutation, the resulting variant is retained if neutral, and discarded otherwise. The process continues until we have collected 100 neutral variants. The mean program length and mutational robustness of the individuals in this population are shown as the leftmost red and blue points respectively in Fig. 5a. From these 100 neutral variants, we then generate a population of 100 second order neutral variants. This is accomplished by looping through the population of first-order mutants, randomly mutating each individual once retaining the result if it is neutral and discarding it otherwise. Once 100 neutral second-order variants have been accumulated, the procedure is iterated to produce higher-order neutral variants. This process produces neutral populations separated from the original program by successively more neutral mutations. Figure 5 shows the results of this process up to 250 steps producing a final population of 100 neutral variants, each of which is 250 neutral mutations away from the original program.



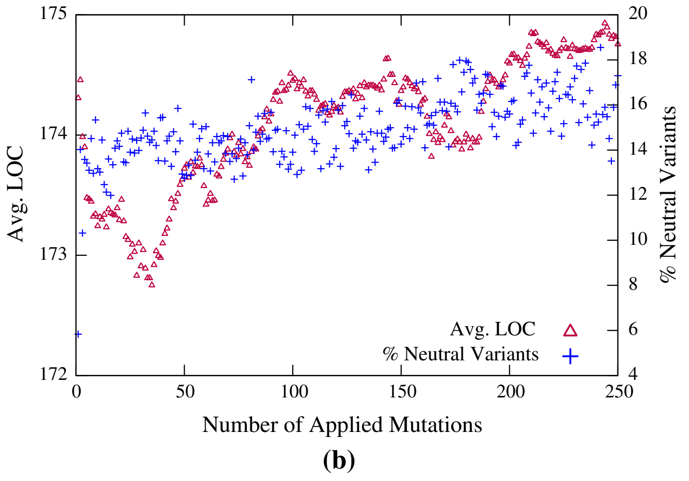


Fig. 5 Random walk in neutral landscape of ASM variations of Insertion Sort. A series of populations of 100 neutral variants from 1 to 250 edits away from the original program. At each step on the X-axis, the mean number of lines of code and mutational robustness of the members of the population are shown. In Panel **a** the size of neutral variants is allowed to vary, while Panel **b** allows only variants that are less than or equal to the length of the original program (in ASM LOC). On the 32-bit machine used for this experiment `insertion-sort` compiles to 175 assembly LOC. Both the average size and the mutational robustness of mutant variants are shown on the Y-axis, the X-axis shows the cumulative number of mutation operators applied

The results show that under this procedure mutational robustness increases with the mutational distance away from the original program. This is not surprising given that at each step mutationally robust variants are more likely to produce neutral mutants which will be included in the subsequent population. We conjecture that this result corresponds to the population drifting away from the perimeter of the program's neutral space. Similar behavior has been described for biological systems, where populations in a constant environment experience a weak evolutionary pressure for increased mutational robustness [87, Ch. 16].

The average size of the program also increases with mutational distance from the original program (Fig. 5a), suggesting that the program might be achieving robustness by adding "bloat" in the form of useless instructions. To control for bloat, Fig. 5b shows the results of an experiment in which only individuals that are the same size or smaller than (measured in number of assembly instructions) the original program are counted as neutral. With this additional criterion, mutational robustness continues to increase but the program size periodically dips and rebounds, never exceeding the size of the original program. The dips are likely consolidation events, where additional instructions are discovered that can be eliminated.

This result shows that not only are there large neutral spaces surrounding any given program implementation (in this instance, permitting neutral variants as far as 250 edits removed from a well-tested <200 LOC program), but they are easily traversable through iterative mutation. Figure 5b shows a small increase in mutational robustness even when controlling for bloat. Further experimentation will determine if these results generalize to multiple programs.

4.6 Multiple languages

The source language and compilation process impose important regularities on the assembly representations of programs. In this section we investigate how mutational robustness varies across assembly code derived from different languages. We evaluated the mutational robustness of sorting algorithms compiled from five languages that span three programming paradigms (imperative, object-oriented, and functional). We focused on sorting algorithms because they are small and sufficiently well-understood to test comprehensively: Test suites were hand-crafted to cover all executable assembly instructions, branches, and corner cases.

The results shown in Table 3 demonstrate that mutational robustness is found across multiple programming languages and paradigms. It is striking that even for a small, comprehensively tested sorting program, on average 28 % of the first-order mutations did not change the program's functionality.

This experiment addresses the question of whether our results depend on idiosyncrasies of a particular language implementation or programming paradigm, and they support the claim that mutational robustness occurs at a significant rate in a wide variety of software.

Table 3 Mutational robustness of sorting algorithms at the assembly instruction level with 100 % test suite coverage, for different algorithms and source language

	C	C++	Haskell	OCaml	Avg.	Std. dev.
bubble	25.7	28.2	27.6	16.7	24.6	5.3
insertion	26.0	42.0	35.6	23.7	31.8	8.5
merge	21.2	46.0	24.9	22.7	28.7	11.6
quick	25.5	42.0	26.3	11.4	26.3	12.5
Avg.	24.6	39.5	28.6	18.6	27.9	
Std. dev.	2.3	7.8	4.8	5.7	3.1	

5 Application of mutational robustness to proactive bug repair

The previous sections demonstrated the prevalence of mutational robustness in software, independent of language, algorithm, or test suite coverage. Section 4.5 suggests that multiple mutations can be accumulated in the same program without loss of functionality. This section provides one example of how we might leverage software mutational robustness in a practical application proactively repairing unknown bugs that are not covered by a program's test suite.

We generate a population of multiple variations of a program, which contain nontrivial changes to the algorithm or implementation but remain within the program's neutral space. Although these variants are *neutral* to the original program with respect to the existing test suite, they may not be neutral to the original if the test suite were later enhanced (e.g., by including tests for as yet unknown bugs). We find that some of these program variations are immune to latent bugs in the original program, and they can be used to automatically pinpoint repairs when new bugs arise.

The term *artificial diversity* describes a wide variety of techniques for automatically randomizing non-functional properties of programs with the goal of disrupting widely replicated attacks. Many methods have been proposed, including stack frame layouts, instruction set numberings, or address space layouts [7, 23, 34]. Although the idea of diversifying certain aspects of a program's implementation has been previously proposed [8], neutral mutations provide a much more general and practical approach. We extend earlier work in this area by generating program variants that are distinct algorithmically but neutral with respect to the test suite. These distinct variants could be used in an *n*-variant system [15], in which a diverse population of programs is run simultaneously on the same inputs. When the neutral variants contain algorithmic or implementation changes (rather than simple remappings, as in address space layout randomization or instruction set randomization), the approach is known as *implementation diversity* [14]. Such a system could be used to flag potential bugs when there are discrepancies in observed behavior, and may be more robust to bugs which only affect subsets of the population.

This use of mutational robustness is analogous to software mutation testing, with the critical differences that (1) neutral mutants are retained rather than manually examined; (2) the test suite is not augmented to kill all mutants; and (3) the set of mutation operators considered is different. The commercial practice of mutation testing has been limited by the significant effort required to analyze mutants that pass the test suite. Such mutants must be manually classified, either as fully equivalent to the original program or non-equivalent, and the latter further classified as buggy or as superior to the original program (cf. *human oracle problem* [94]).

Our proposed alternative to the traditional mutation testing practice amortizes these labor intensive steps by retaining a population of all such *neutral* variants. When a bug is encountered in the original program, it will be detected if some members of the population behave anomalously with respect to the result of the population, and the non-failing variations need only then be analyzed to suggest a repair. This approach of deferring analysis until a potentially beneficial variation is found may be more feasible than traditional mutation testing, because it does not require exhaustive manual review of large numbers of program variants.

5.1 Repairing bugs

We first demonstrate that it is possible to construct variants in the neutral fitness landscape that can repair unknown bugs while retaining required functionality. To do this, we seeded each of the programs in Table 4 with five random defects following an established defect distribution [25, p. 5] and fault taxonomy [47] (e.g., “missing conditional clause,” “extra statement,” “constant should have been variable,” “wrong parameter”). The defects were seeded in advance and without

Table 4 Bugs proactively repaired by neutral variants

Program	Fraction of bugs fixed	Bug fixes
`bzip`	2/5	63
`imagemagick`	2/5	8
`jansson`	2/5	40
`leukocyte`	1/5	1
`lighttpd`	1/5	73
`nullhttpd`	1/5	7
`oggenc`	0/5	0
`potion`	2/5	14
`redis`	0/5	0
`tiff`	0/5	0
`vyquon`	1/5	1
Average	1.0/5	18.8

Five unique bugs were seeded in each subject program according to a defect distribution taken from the Firefox open source project. Five thousand neutral variants were created for each program through AST level mutation, without regard to the seeded bugs. Each variant passed all visible tests. The “Unique Bugs Fixed” column counts the number of seeded bugs fixed by at least one variant. The “Bug Fixes” column counts the number of variants that fixed at least one bug regard to the mutation operators. For each defect we produced a held-out test to verify its presence or absence.

For each program, we generated 5,000 first-order neutral variants using the mutation operators defined in Sect. 3.2. We then noted which of these passed any of the five held-out test cases for the seeded bugs. In practice, 5,000 neutral variants proved sufficient to generate at least one bug-repairing variant for most programs, and if such a variant was generated at all, it was within the first 5,000 neutral variants. We tested this by searching up to 20,000 variants, which did not improve performance. Only variants that passed the original test suite were retained; the mutation process did *not* have access to the held-out test cases for the seeded bugs.

Table 4 shows the results. We say that a variant repairs an unknown bug if it passed all original test cases and the held-out test case associated with that bug. We found repairs which exactly revert the original bug as well as *compensatory* mutations which do not touch the bug itself but repair or avoid the bug by changing other parts of the program. Specifically, we found that 3 % of the proactive repairs changed the same line of code in which the original bug was seeded and 12 % of repairs affect code within 5 lines of the seeded bug. The remaining 88 % of repairs could be considered compensatory in that they do not affect the bug directly but rather change other portions of the program so that the bug is not expressed. In nature compensatory mutations are much more common than mutations which directly repair a given fault [67].

These proactive repairs can be used to *pinpoint* the bug, because the `diff` between them and the original program locates either the bug itself or relevant portions of the program code. Previous work demonstrated that software engineers take less time to address defect reports that are accompanied by such machine-generated patch-like information [91], which provides evidence that these proactive repairs would be useful in practice.

We observe some common trends when examining Table 4. The bugs repaired most easily were those that naturally mirror the mutation strategies employed by our technique. For example, we found multiple examples of the repairs that deleted problematic statements or clauses, corrected an incorrect value for a constant, changed a relational operator (for instance $\leq$ to $<$), inserted clauses or statements to test for extra conditions, changed a parameter value in a function call, etc. However, there was significant overlap between the *types* of bugs that were proactively repaired and those that were not, suggesting that additional tuning might improve results on these currently unrepaired classes of bugs. One bug that was never repaired in our experiment involved an “incorrect function call.” To repair this bug using our technique would require finding the “correct” function elsewhere in the program with exactly the correct parameters, while avoiding extraneous mutations that change behavior on regression test cases. In this experiment, we focused on discovering the proactive repair, and we leave for future work the problem of automatically deciding how to resolve discrepancies that are discovered among selected variants, either with an automated repair or by generating additional test cases.

We predict that the more latent bugs there are in a program, the more likely it is that at least one of them will be repaired through proactive diversity. This is relevant because most deployed programs have significantly more than five outstanding defects (e.g., 18,165 from October 2001 to August 2005 and 2,013 open bug tickets May 2003 to August 2005 for Eclipse (V3.0) and Firefox (V1.0) respectively [4]). To test this prediction, we seeded one of our test programs, `potion`, with ten additional held-out defects. We then generate 5,000 neutral variants which were selected to maximize the number the distinct positions in the original program which they modify. Figure 6 plots the number of distinct bugs repaired by these 5,000 variants as a function of the number of defects seeded, yielding a correlation of 95 %. If the linear relation shown in Fig. 6 applies to the Eclipse and Firefox projects, a population of 5,000 program variants could proactively repair as many as 9,000 and 1,000 latent bugs in those systems, respectively.

We leave as future work the application of this technique to higher order neutral mutants as constructed in Sect. 4.5. While such variants would also greatly increase diversity they would be less useful in directly pinpointing the source code implicated in buggy behavior.

Although preliminary, these results show how the mutational robustness properties of programs might be used to quickly repair programs or as in the scenario outlined in the next subsection.

5.2 Neutral variants for n-version programming

In their classic work on independence in multiversion programming, Knight and Leveson [48] found that distinct teams of humans, when given an identical

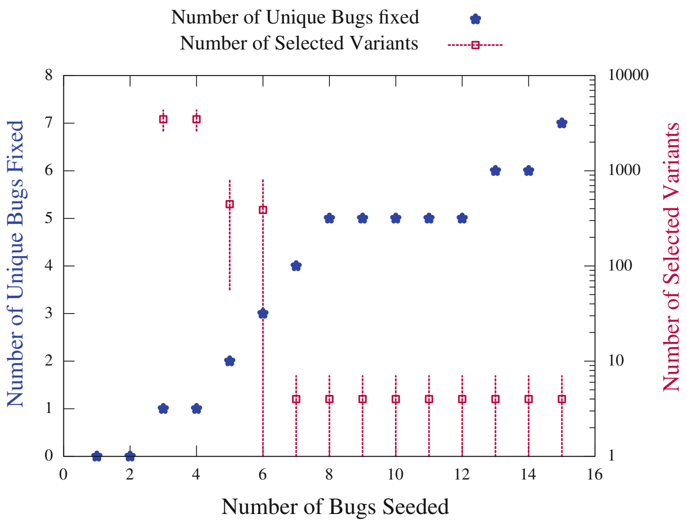


Fig. 6 Number of unique bugs repaired and number of unique variants required to repair at least one seeded bug as a function of number of bugs seeded for the `potion` program. Starting with a population of 5,000 neutral program variants the total number of unique bugs repaired is shown in *blue* and the number of neutral variants needed to find a repair for at least one seeded bug is shown in *red*. The X-axis is the number of bugs seeded in the subject program (`potion`). The Pearson correlation coefficient between the number bugs seeded and repaired is 0.95 (Color figure online)

programming task, produce solutions that have correlated errors (bugs). That is, two independent teams are unlikely to produce two independent implementations, which decreases the potential benefit of N-person programming.

Previous work has shown that GP can serve as a promising tool for N-version programming [20]. In particular, GP can automate the significant task of developing N independent software instances. However, the technique had limited applicability because it relied on de-novo programs evolved in simplified languages specifically designed for GP.

The diverse populations of program variants described in this section could serve as the bases for an N-version programming system. Such a system could potentially address part of the non-independence hurdle because the mutations are generated randomly rather than by people, producing independent algorithmic changes. For example, if we seed 15 bugs in `potion` and then select at random ten neutral variants which each proactively repair one bug, we see that, on average, 1.7 different defects are fixed by those ten proactive repairs, rather than 1.0, which we would expect if they were 100 % correlated.

To put this idea into perspective, for the first month after Firefox 4.0 was released (March 11 through April 10, 2011), the project's Bugzilla database shows that an average of 77 new, non-duplicate bugs were reported per day. Given that this rate of bug reporting is more than ten times the number of bugs seeded in our evaluation, we conjecture that an N-version programming system populated by our technique would be at least as effective in similar real-world systems as it is in our experimental setup.

6 Discussion

The results presented here contradict the prevailing folk wisdom that software is a precise and intentionally engineered mechanism, which is brittle to small perturbations. We find software to be inherently robust to random mutations, malleable within extensive neutral landscapes, and evolvable by combining the mutation operators described here with crossover and selection [50, 51, 92].

This new view of software suggests a number of exciting areas for future work. These include further analysis of the extent, origins and mechanics of software mutational robustness, novel practical applications to software engineering, and new parallels between software applications and evolved biological organisms. We next discuss threats to the validity of our findings, and then explore each of these three areas of future work in turn.

6.1 Threats to validity

The quantitative results that we report depend on the particular choice of mutation operators, and a competent programmer could likely craft operators capable of achieving any pre-set desired level of mutational robustness. However, we believe that the operators used in this paper are sufficiently simple, powerful and general that they expose software mutational robustness as an inherent property of software.

A topic left for future investigation is to define and test a wider variety of mutation operators (including mutation operations taken from the mutation testing community such as Mothra [44]), studying their effect on software mutational robustness.

We use test suites to assess program behavior. Insufficient test suites could artificially inflate our estimates of software mutational robustness. Multiple aspects of our experimental design addresses this threat explicitly. In Sect. 4.1 we selected benchmarks programs with a wide variety of test-suite coverage and depth, including both the Siemens programs, whose test suites have been developed by multiple independent researchers to achieve an exceptionally high quality, and small sorting programs, where we can test all corner cases exhaustively. In Sect. 4.3 we analyze mutational robustness across these three categories of programs, finding little variation in the average and minimal mutational robustness measured for each category. As a further step to address concerns about test-suite quality, in Sect. 4.4 we manually categorized neutral variants for a small program, finding that at least 29 of the 35 analyzed variants could not possibly fail any test suite that conforms to the program specification.

In Sect. 5 we demonstrate applications of software mutational robustness to the tasks of software development and maintenance. This demonstration does not address the impact which low-quality test suites may have on the performance of our demonstrated applications. Such an investigation may be required before such techniques are applied.

6.2 Further investigation of software mutational robustness

Although we demonstrate the reach of neutral software landscapes in Sect. 4.5 by evolving neutral variants hundreds of edits removed from an original program, much about these landscapes remains unknown. How densely do the spaces of neutral variants fill the space of all possible programs? Are these neutral spaces connected and traversable, or isolated? Do these spaces extend to all portions of program space (as the neutral spaces of mappings of RNA sequences to RNA structures do [74])?

Measuring the "distance" between neutral program variants may illuminate the effective impact of neutral program mutations. Such distance metrics could be based on information flow through the program during execution [82], or comparison of stack traces or system calls [22].

Recent work has shown that Markov Chain Monte Carlo [3] (MCMC) techniques are capable of traversing neutral program spaces defined by vastly smaller programs on the order of 10 assembly instructions [76]. It remains unclear if these techniques are extensible to the spaces defined by full-size programs with loops.

Software mutational robustness measures the density of neutral variants within a single edit of a program. However, the density of higher-order neutral variants remains unknown. We could possibly use MCMC to sample the space of more neutral variants within a given edit radius or the original program as follows:

1. Calculate the mixing time of an edit-radius constrained neutral variant markkov chain;

2. Record the neutrality of program encountered after the mixing time;
3. Use mark and recapture techniques from population ecology [69] to estimate the total volume of neutral variants.

Performing mark and recapture experiments within neutral landscapes found along the post mixing time walks would allow us to estimate the absolute number of neutral variants within a given edit radius of the original program.

This work does not address execution time software robustness. Examples include: short error propagation distances observed in web servers studied in *failure oblivious* computing [68], and calculations such as those prevalent in the PARSEC benchmark suite [11], which iteratively converge on the correct solutions and which may become more common as future applications leverage multi-core architectures.

6.3 Applications to software engineering

There are an infinite number of possible implementations for any functional program specification, and Table 1 can be interpreted as illustrating how easy it is to find such multiple equivalent implementations. These results also suggest that programs have a significant amount of unidentified and unexploited redundancy. One practical implication of large traversable neutral landscapes would be to search the neutral landscapes for regions that optimize non-functional properties such as program size (as demonstrated in Fig. 5b), memory requirements, runtime, power consumption, or any other non-functional software features. This may explain the success of previous work optimizing graphics shader software [78].

The large neutral landscapes revealed by our study may also explain the success of evolutionary methods, such as GP, on the task of automated program repair [92]. Although this parallel with evolutionary biology is intriguing (Sect. 2.1), we do not yet have definitive evidence that quantifies the role of mutational robustness in automated program repair, a topic that we leave for future work. We suspect that other methods of automated repair (e.g., [16, 36, 66, 92, 93]) may ultimately be understood in the context of software mutational robustness and evolvability.

In addition to the proactive diversity application described in Sect. 5, it might be feasible to incorporate other machine learning methods into software development and maintenance processes; such methods typically work poorly on brittle system and require some degree of robustness while they search for improved solutions.

Our results constitute a fundamentally different interpretation of software mutants than that of the mutation testing community. We do not view software mutants as faulty or neutral mutants as an indicator of an insufficient test suite. Instead we view software mutants as natural and neutral mutants as alternate implementations of software specifications which admit many non-equivalent implementations. The quick-sort example discussed in Sect. 1 demonstrates that simple mutations can yield different fully correct implementations. Yin et al. [97, p. 61] provide another example in which a fully-formally verified implementation of AES encryption is changed based on what was “purely an implementation decision, [where] the specification did not impose any restrictions,” yielding another formally verified implementation.

Such alternate implementations, which may have distinct runtime properties but still conform to the program specification, are *neutral*, but are not *equivalent* in the sense used by the mutation testing community. The authors have performed a thorough review of the mutation testing literature to compare our empirical findings of software mutational robustness to typical frequencies of equivalent mutants in the mutation testing paradigm. The mutation testing community does identify equivalent mutants as one of the fundamental problems in mutation testing (cf. [35, Section II.C]). Although this problem is well known, we were unable to find formal publications that experimentally identify the fraction of equivalent mutants (aside from work explicitly targeting Object-Oriented mutation operators which generate particularly high rates of equivalent mutants [62, 71, 75]).

Through our own review of the mutation testing literature we collected unreported counts of equivalent mutants from a number of papers [17, 24, 59, 61] that all used the Mothra [44] mutation operators and that found equivalent mutant rates of 9.92, 6.75, 6.24 and 6.17 % respectively. Although these works used different mutation operators than those described in Sect. 3.2, the percentages of equivalent mutants found are smaller than the percentage of neutral mutants which we report, and are thus consistent with our results because equivalent mutants are a strict subset of neutral mutants.

Our re-interpretation of software mutants opens the possibility of re-purposing the many tools developed by the mutation testing community. For example, runtime optimization techniques such as *mutant scheme generation* [83] enable the compilation of "super mutants" capable of executing all first-order mutants of a program. Such a system could potentially be applied to efficiently deploy and run populations of diverse software variants as described in Sect. 5 on end user systems. Techniques for the automatically identifying fully equivalent mutants [59, 60] could be used to differentiate between those neutral mutants that are identical to the original program and those that encode a distinct implementation. These are just two examples of how the extensive toolbox of mutation testing could be re-purposed to leverage the view of software mutants as neutral.

6.4 Comparison to biological mutational robustness

Beyond the significance for computation, we believe that our results are relevant to biologists. We considered the effect of repeated neutral mutations in a single program, showing that robustness can increase systematically through population exploration of the neutral landscape; a property shared with biological systems [89]. We also showed that it is possible to generate programs that are many mutational steps removed from the original while retaining functionality (i.e., without leaving the neutral plateau). These large extended neutral landscapes are thought to be essential to the ability of biological systems to improve through natural selection [74, 84] but difficult to measure experimentally. Our software analogs may eventually provide a useful experimental framework for testing hypotheses about the role of neutrality in biological evolution.

Currently, it is difficult to draw conclusions from a quantitative comparison of the 36.8 % mutational robustness found in software to typical levels of mutational robustness in biological systems (e.g, 30 % mutational robustness in hominids [19], or the almost 40 % neutrality of gene knockouts (deletions) in yeast [79]). There are many drivers of mutational robustness in biological systems; environmental stability influences levels of robustness [55], the centrality of a gene may influence its mutational robustness [38], evolution may either select for mutational robustness [84] or it may be more effective in organisms that are mutationally robust [13]. There are analogs to each of these factors in software systems, which future work may relate to the levels of mutational robustness found in software.

Mutational robustness has many correlates in biological systems. These include a correlation between environmental and mutational robustness [45, 52, 84] and a correlation between mutational robustness and evolvability [89]. The presence of analogous correlations in software systems is an intriguing possibility that could be investigated empirically. Such an investigation could indicate whether these relations are general across complex mutationally robust systems or are specific to biological systems. If such correlations do exist in software, they could lead to new applications, such as methods of automated hardening which automatically increase environmental robustness through increasing mutational robustness (as in Sect. 4.5).

We may use the ratio between genome size and gene number as a proxy for the relative mutational robustness of biological organisms, although this would certainly be a lower bound. However this ratio varies widely from 97:47 % in prokaryotes and viruses, to 87:1 % in eukaryotes. We note that those portions of software that are executed by the test suite (e.g., where we limit our mutation operations) could be compared to those portions of the genome that code for genes. In both cases we can be sure that those portions have a phenotypic effect, and in both cases we can't say for sure that the uncovered or non-coding portions have no effect. Such portions may either regulate expression or affect compilation respectively. Direct software analogs of non-coding regulatory DNA could include type annotations or pragmas.

Similarly, some of the causes of large genomes in biological organisms have immediate analogs in the would of software development. Gene duplication is thought to be a significant contributor to the growth in genome size. The analogous practice of copy-paste programming is a wide spread and common software development technique [37, 39, 41, 49].

Ultimately software may stand with biological organisms as a second example of an evolved system. Albeit one in which humans engineers are the mechanisms of both mutation and selection [1].

7 Conclusion

The previous sections described experimental results, using three simple mutation operators, which show that software is surprisingly robust to random mutations. For the programs we tested, 37 % of the mutations had no effect on software functionality, as measured by the programs test suites. Software mutational robustness, or neutrality, is observed even in programs that are is completely correct according to their specifications. Just as neutrality is believed to enhance evolvability in naturally evolving populations, so may software neutrality enable and explain the evolvability of software, either through automated means (e.g., [92]) or by humans.

Software robustness is potentially useful for enhancing the resilience of software systems. We demonstrate this idea by describing a method that increases software diversity, automatically generating software variants that are immune to as yet undiscovered bugs. The insights into software described here suggest several opportunities for the Software Engineering community, including the following: Creating system diversity, for example, to protect against security exploits; incorporating machine learning methods into software development and maintenance; improving program performance; or developing error-tolerant computations.

We postulate that the presence of mutational robustness in software is not an effect of intentional design, but is rather an effect of software's provenance through natural selection—even though the agents of selection, mutation and reproduction are human engineers. In this way, mutational robustness can be viewed as a property arising through inadvertent selection in both natural and engineered systems. Further study of software as an evolved system may yield new insights into those aspects of evolution that are specific to biological systems and those which are general across other complex evolved, and even engineered, systems. Because software is fundamentally easier to instrument and observe than naturally occurring populations, studying software robustness may lead in the future to an increased understanding of the role of neutrality in natural evolution.

Acknowledgments The authors would like to thank Lauren Ancel Meyers, William B. Langdon and Peter Schuster for their thoughtful comments. This work was supported by the National Science Foundation (SHF-0905236), Air Force Office of Scientific Research (FA9550-07-1-0532, FA9550-10-1-0277), DARPA (P-1070-113237), DOE (DE-AC02-05CH11231) and the Santa Fe Institute.

References

1. D. Ackley, Personal communication (2000)
2. T. Ackling, B. Alexander, I. Grunert, Evolving patches for software repair. in *Genetic and Evolutionary Computation* (2011), pp. 1427–1434
3. C. Andrieu, N. De Freitas, A. Doucet, M.I. Jordan, An introduction to MCMC for machine learning. Mach. Learn. **50**(1–2), 5–43 (2003)
4. J. Anvik, L. Hiew, G.C. Murphy, Coping with an open bug repository. in *OOPSLA Workshop on Eclipse Technology eXchange* (2005), pp. 35–39
5. A. Arcuri, Evolutionary repair of faulty software. Appl. Soft Comput. **11**(4), 3494–3514 (2011)
6. W. Banzhaf, A. Leier, Evolution on neutral networks in genetic programming. in *Genetic Programming Theory and Practice III* (2006), pp. 207–221
7. G. Barrantes, D. Ackley, S. Forrest, D. Stefanovic, Randomized instruction set randomization. ACM Trans. Inf. Syst. Secur. (TISSEC) **8**(1), 3–40 (2005)
8. E.G. Barrantes, D.H. Ackley, S. Forrest, D. Stefanovic, Randomized instruction set emulation. ACM Trans. Inf. Syst. Secur. **8**(1), 3–40 (2005)
9. E.G. Barrantes, D.H. Ackley, T.S. Palmer, D. Stefanovic, D.D. Zovi, Randomized instruction set emulation to disrupt binary code injection attacks. in *Computer and Communications Security* (2003), pp. 281–289
10. S. Bhatkar, D. DuVarney, R. Sekar, Address obfuscation: an effcient approach to combat a broad range of memory error exploits. in *USENIX Security Symposium* (2003)

11. C. Bienia, S. Kumar, J. Singh, K. Li, The parsec benchmark suite: characterization and architectural implications. in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (ACM, 2008), pp. 72–81
12. T.A. Budd, D. Angluin, Two notions of correctness and their relation to testing. Acta Informatica **18**, 31–45 (1982)
13. S. Ciliberti, O. Martin, A. Wagner, Innovation and robustness in complex regulatory gene networks. Proc. Natl. Acad. Sci. **104**(34), 13–591 (2007)
14. C. Cowan, H. Hinton, C. Pu, J. Walpole, The cracker patch choice: an analysis of post hoc security techniques. in *Proceedings of the 23rd National Information Systems Security Conference (NISSC)* (2000)
15. B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, J. Hiser, N-variant systems: a secretless framework for security through diversity. in *USENIX Security Symposium* (2006)
16. V. Dallmeier, A. Zeller, B. Meyer, Generating fixes from object behavior anomalies. in *Automated Software Engineering* (2009), pp. 550–554
17. R. DeMillo, A. Offutt, Constraint-based automatic test data generation. Trans. Softw. Eng. **17**(9), 900–910 (1991)
18. J. Draghi, T. Parsons, G. Wagner, J. Plotkin, Mutational robustness can facilitate adaptation. Nature **463**(7279), 353–355 (2010)
19. A. Eyre-Walker, P. Keightley, The distribution of fitness effects of new mutations. Nat. Rev. Genet. **8**(8), 610–618 (2007)
20. R. Feldt, Generating diverse software versions with genetic programming: an experimental study. IEE Proc. Softw. **145**(6), 228–236 (1998)
21. R.A. Fisher, *The genetical theory of natural selection*. (Oxford, Oxford, 1930)
22. S. Forrest, S. Hofmeyr, A. Somayaji, The evolution of system-call monitoring. in *Annual Computer Security Applications Conference* (2008), pp. 418–430
23. S. Forrest, A. Somayaji, D.H. Ackley, Building diverse computer systems. in *Workshop on Hot Topics in Operating Systems* (1997), pp. 67–72
24. P.G. Frankl, S.N. Weiss, C. Hu, All-uses vs mutation testing: an experimental comparison of effectiveness. J. Syst. Softw. **38**(3), 235–253 (1997)
25. Z.P. Fry, W. Weimer, A human study of fault localization accuracy. in *International Conference on Software Maintenance* (2010), pp. 1–10
26. D. Garlan, J. Kramer, A.L. Wolf, (eds.), *Proceedings of the First Workshop on Self-Healing Systems, WOSS 2002, Charleston, South Carolina, USA, November 18–19, 2002* (ACM, 2002)
27. T. Graves, M. Harrold, J. Kim, A. Porter, G. Rothermel, An empirical study of regression test selection techniques. Trans. Softw. Eng. Methodol. **10**(2), 184–208 (2001)
28. I. Harvey, A. Thompson, Through the labyrinth evolution finds a way: a silicon ridge. in *Evolvable Systems: From Biology to Hardware*, pp. 406–422, (1997)
29. W. Hordijk, A measure of landscapes. Evolut. Comput. **4**(4), 335–360 (1996)
30. W.E. Howden, Reliability of the path analysis testing strategy. Softw. Eng. IEEE Trans. **2**(3), 208–215 (1976)
31. M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments of the effectiveness of dataflow-and control flow-based test adequacy criteria. in *International Conference on Software Engineering* (1994), pp. 191–200
32. M. Huynen, P. Stadler, W. Fontana, Smoothness within ruggedness: the role of neutrality in adaptation. Proc. Natl. Acad. Sci. **93**(1), 397 (1996)
33. IEEE security and privacy, special issue on IT monocultures. **7**(1) (2009)
34. T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, M. Franz, Compiler-generated software diversity. in *Moving Target Defense* (Springer, 2011), pp. 77–98
35. Y. Jia, M. Harman, An analysis and survey of the development of mutation testing. Trans. Softw. Eng. **37**(5), 649–678 (2011). doi:10.1109/TSE.2010.62
36. G. Jin, L. Song, W. Zhang, S. Lu, B. Liblit, Automated atomicity-violation fixing. in *Programming Language Design and Implementation* (2011)
37. E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter? in *International Conference on Software Engineering* (2009), pp. 485–495
38. R. Kafri, O. Dahan, J. Levy, Y. Pilpel, Preferential protection of protein interaction network hubs in yeast: evolved functionality of genetic redundancy. Proc. Natl Acad. Sci. **105**(4), 1243–1248 (2008)

39. T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. **28**, 654–670 (2002)
40. G. Kc, A. Keromytis, V. Prevelakis, Countering code-injection attacks with instruction-set randomization. in *Computer and Communications Security* (2003), pp. 272–280
41. M. Kim, L. Bergman, T. Lau, D. Notkin, An ethnographic study of copy and paste programming practices in oopl. in *Empirical Software Engineering, 2004. ISESE'04. Proceedings 2004 International Symposium on* (IEEE, 2004), pp. 83–92
42. M. Kimura et al., Evolutionary rate at the molecular level. Nature **217**(5129), 624 (1968)
43. M. Kimura, *The Neutral Theory of Molecular Evolution.* (Cambridge University Press, Cambridge, 1985)
44. K. King, A. Offutt, A fortran language system for mutation-based software testing. Softw. Pract. Exp. **21**(7), 685–718 (1991)
45. H. Kitano, Biological robustness. Nat. Rev. Genet. **5**(11), 826–837 (2004)
46. J.C. Knight, P. Ammann, Issues influencing the use of n-version programming. in *IFIP Congress* (1989)
47. J.C. Knight, P.E. Ammann, An experimental evaluation of simple methods for seeding program errors. in *International Conference on Software Engineering* (1985)
48. J.C. Knight, N.G. Leveson, An experimental evaluation of the assumption of independence in multiversion programming. IEEE Trans. Softw. Eng. **12**(1), 96–109 (1986)
49. J. Krinke, A study of consistent and inconsistent changes to code clones. in *Working Conference on Reverse Engineering* (IEEE Computer Society, 2007), pp. 170–178. doi:10.1109/WCRE.2007.7
50. C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, GenProg: a generic method for automated software repair. Trans. Softw. Eng. **38**(1), 54–72 (2012)
51. C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, A systematic study of automated program repair: fixing 55 out of 105 bugs for 8 each. in *International Conference on Software Engineering* (2012), pp. 3–13
52. R.E. Lenski, J. Barrick, C. Ofria, Balancing robustness and evolvability. PLoS Biol. **4**(12), e428 (2006)
53. B. Littlewood, D. Miller, Conceptual modelling of coincident failures in multiversion software. IEEE Trans. Softw. Eng. **15**(12), 1596–1614 (1989)
54. C. Meiklejohn, D. Hartl, A single mode of canalization. Trends Ecol. Evol. **17**(10), 468–473 (2002)
55. L.A. Meyers, F.D. Ancel, M. Lachmann, Evolution of genetic potential. PLoS Comput. Biol. **1**(3), e32 (2005)
56. J. Miller, Cartesian genetic programming. in *Cartesian Genetic Programming* (2011), pp. 17–34
57. S. Misailovic, D. Roy, M. Rinard, Probabilistically accurate program transformations. in *Static Analysis* (2011), pp. 316–333
58. G.C. Necula, S. McPeak, S.P. Rahul, W. Weimer, Cil: An infrastructure for C program analysis and transformation. in *International Conference on Compiler Construction* (2002), pp. 213–228
59. A. Offutt, W. Craft, Using compiler optimization techniques to detect equivalent mutants. Softw. Test. Verif. Reliab. **4**(3), 131–154 (1994)
60. A. Offutt, J. Pan, Automatically detecting equivalent mutants and infeasible paths. Softw. Test. Verif. Reliab. **7**(3), 165–192 (1997)
61. A. Offutt, A. Lee, G. Rothermel, R. Untch, C. Zapf, An experimental determination of sufficient mutant operators. Trans. Softw. Eng. Methodol. **5**(2), 99–118 (1996)
62. J. Offutt, Y. Ma, Y. Kwon, The class-level mutants of mujava. in *International Workshop on Automation of Software Test* (ACM, 2006), pp. 78–84
63. C. Ofria, W. Huang, E. Torng, On the gradual evolution of complexity and the sudden emergence of complex features. Artif. Life **14**(3), 255–263 (2008)
64. M. Orlov, M. Sipper, Flight of the FINCH through the Java wilderness. Trans. Evolut. Comput. **15**(2), 166–192 (2011)
65. M. Orlov, M. Sipper, Genetic programming in the wild: Evolving unrestricted bytecode. in *Genetic and Evolutionary Computation Conference*, (2009), pp. 1043–1050
66. J.H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.F. Wong, Y. Zibin, M.D. Ernst, M. Rinard, Automatically patching errors in deployed software. in *Symposium on Operating Systems Principles* (2009)
67. A. Poon, B.H. Davis, L. Chao, The coupon collector and the suppressor mutation estimating the number of compensatory mutations by maximum likelihood. Genetics **170**(3), 1323–1332 (2005)

68. M.C. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, W.S. Beebee, Enhancing server availability and security through failure-oblivious computing. in *Operating Systems Design and Implementation* (2004), pp. 303–316
69. D. Robson, H. Regier, Sample size in petersen mark–recapture experiments. Trans. Am. Fish. Soc. **93**(3), 215–226 (1964)
70. G. Rothermel, M. Harrold, Empirical studies of a safe regression test selection technique. Trans. Softw. Eng. **24**(6), 401–419 (1998)
71. D. Schuler, A. Zeller, (un-) covering equivalent mutants. in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on* (IEEE, 2010), pp. 45–54
72. E. Schulte, J. DiLorenzo, S. Forrest, W. Weimer, Automated repair of binary and assembly programs for cooperating embedded devices. in *Architectural Support for Programming Languages and Operating Systems* (2013)
73. E. Schulte, S. Forrest, W. Weimer, Automatic program repair through the evolution of assembly code. in *Automated Software Engineering* (2010), pp. 33–36
74. P. Schuster, W. Fontana, P. Stadler, I. Hofacker, From sequences to shapes and back: a case study in rna secondary structures. in *Proceedings: Biological Sciences* (1994), pp. 279–284
75. S. Segura, R. Hierons, D. Benavides, A. Ruiz-Cortés, Mutation testing on an object-oriented framework: an experience report. Inf. Softw. Technol. **53**(10), 1124–1136 (2011)
76. R. Sharma, E.S., A. Aiken, Stochastic superoptimization. in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, 2013)
77. S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, A.D. Keromytis, Assure: automatic software self-healing using rescue points. in *Architectural Support for Programming Languages and Operating Systems* (2009), pp. 37–48
78. P. Sitthi-Amorn, N. Modly, W. Weimer, J. Lawrence, Genetic programming for shader simplification. ACM Trans. Graph. **30**(5), 152 (2011)
79. V. Smith, K. Chou, D. Lashkari, D. Botstein, P. Brown et al., Functional analysis of the genes of yeast chromosome v by genetic footprinting. Science **274**(5295), 2069–2074 (1996)
80. T. Smith, P. Husbands, P. Layzell, M. O'Shea, Fitness landscapes and evolvability. Evolut. Comput. **10**(1), 1–34 (2002)
81. T. Smith, P. Husbands, M. O'Shea, Neutral networks and evolvability with complex genotype-phenotype mapping. in *Advances in Artificial Life* (2001), pp. 272–281
82. G.E. Suh, J.W. Lee, D. Zhang, S. Devadas, Secure program execution via dynamic information flow tracking. in *ACM SIGPLAN Notices*, vol. 39, (ACM, 2004), pp. 85–96
83. R. Untch, *Schema-Based Mutation Analysis: A New Test Data Adequacy Assessment Method.* Ph.D. thesis. (Clemson University, Clemson, 1995)
84. E. Van Nimwegen, J. Crutchfield, M. Huynen, Neutral evolution of mutational robustness. Proc. Natl. Acad. Sci. **96**(17), 9716 (1999)
85. V. Vassilev, J. Miller, The advantages of landscape neutrality in digital circuit evolution. in *Evolvable systems: from biology to hardware* (2000), pp. 252–263
86. F. Vokolos, P. Frankl, Empirical evaluation of the textual differencing regression testing technique. in *International Conference on Software Maintenance* (1998), pp. 44–53
87. A. Wagner, *Robustness and Evolvability in Living Systems (Princeton Studies in Complexity).* (Princeton University Press, Princeton, 2005)
88. A. Wagner, Neutralism and selectionism: a network-based reconciliation. Nat. Rev. Genet. **9**(12), 965–974 (2008)
89. A. Wagner, Robustness and evolvability: a paradox resolved. Proc. R. Soc. B Biol. Sci. **275**(1630), 91 (2008)
90. K.S.H.T. Wah, A theoretical study of fault coupling. Softw. Test. Verif. Reliability **10**(1), 3–45 (2000)
91. W. Weimer, Patches as better bug reports. in *Generative Programming and Component Engineering* (2006), pp. 181–190
92. W. Weimer, T. Nguyen, C. Le Goues, S. Forrest, Automatically finding patches using genetic programming. in *International Conference on Software Engineering* (2009), pp. 364–367
93. Y. Wei, Y. Pei, C.A. Furia, L.S. Silva, S. Buchholz, B. Meyer, A. Zeller, Automated fixing of programs with contracts. in *International Symposium on Software Testing and Analysis* (2010), pp. 61–72
94. E. Weyuker, On testing non-testable programs. Comput. J. **25**(4), 465–470 (1982)

95. J.L. Wilkerson, D. Tauritz, Coevolutionary automated software correction. in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation* (ACM, 2010), pp. 1391–1392
96. D. Williams, W. Hu, J.W. Davidson, J. Hiser, J.C. Knight, A. Nguyen-Tuong, Security through diversity: leveraging virtual machine technology. IEEE Secur. Priv. **7**(1), 26–33 (2009)
97. X. Yin, J.C. Knight, W. Weimer, Exploiting refactoring in formal verification. in *International Conference on Dependable Systems and Networks* (2009), pp. 53–62
98. E. Zuckerkandl, L. Pauling, Molecular disease, evolution and genetic heterogeneity. in *Horizons in Biochemistry* (1962)