MUTATIONAL ROBUSTNESS AND AUTOMATIC PROGRAM REPAIR

ETHAN FAST

Mutational robustness describes how likely a variant's phenotype is to remain constant in response to mutations applied within its genotype. This measure has not been evaluated generally across software, nor more specifically in the context of a genetic programming (GP) approach to automated program repair. We provide an analysis of this metric across eight benchmark programs, representing each program's genotype as an AST, and each phenotype with a regression test suite. First, we quantify the robustness of these programs, with respect to the mutational operators used in existing GP program repair methods, and analyze the extent that these results can be applied to software more broadly. Next, we evaluate the relationship between program robustness and repair success in existing GP techniques. Finally, we identify a class of program faults which inhibit robustness under current repair methods, and propose changes in variant representation under GP that enable the repair of these more difficult bugs.

1. Introduction

Fixing bugs requires a great deal of time, money, and human effort [7]. In fact, for major software projects, over 70% of development costs are allocated to maintaining and repairing faulty code [8], and frequently, a lack of available resources prevents a development team for addressing all bugs, both known and unknown. [1]. Moreover, bugs are inevitably discovered after a project has left production, and such faults accumulate too quickly for all of them to be addressed [2]. Automated repair techniques have the potential to alleviate this burden, saving developers both time and money. In particular, automatic repair via genetic programming (GP) has proved effective at repairing bugs across a variety of programs [10].

We hypothesize that this success in program repair is in large part due to the robustness of C-programs. That is, in making a random changes to a program, such changes will not impact program functionality, with some significant probability. This property, more widely known as *mutational robustness* has not been analyzed with

respect to real-world software, and appears more commonly in the biological sciences. Here our primary purpose is to explore robustness in the context of program code and automatic program repair.

Moreover, an examination of program robustness may provide insight into potential improvements for automatic program repair. Although existing GP repair techniques are quite powerful in addressing many kinds of faults, they may have particular difficulty in generating repairs in strong *non-robust code*. By this term we refer to repairs that are impossible to generate without breaking program functionality on some intermediate step. Some idea of how commonly such non-robust code appears in software will inform the practicality of these program repair methods. Further, one might develop program representations designed specifically to mitigate the difficulties of such code — that is, contracting representations that transform formerly non-robust code into a representation with a greater degree of robustness.

To this end, an important concept to consider is the neutral fitness landscape associated with each for existing benchmark program. This landscape is closely related to the concept of mutational robustness, defining a region of different variant genotypes with equivalent functional behavior. That is, it defines regions of dissimilar genotypes but equivalent phenotypes. We hypothesize that programs and program representations with more extensive regions of neutral fitness are more likely to evolve repairs, particularly in cases for which a repair requires a progression of accumulated changes.

As a complete examination of these neutral landscapes is computationally unfeasible, a heuristic must be used to uncover parts of the landscape. For this purpose, and so here we use mutational robustness — the likelihood that a change in variant genotype will affect variant phenotype — to examine the properties of neutral fitness landscapes across a few benchmark programs. For instance, a program exhibiting high mutational robustness is likely to have a more extensive region of neutral fitness than a program exhibiting low robustness, as in undergoing mutation, the former is less likely to change its behavior than the latter. Likewise, we can use mutational

robustness to order the affects of varying genotypic representations on the extent of the neutral landscape. It is possible that different ways of representing the variant throughout the repair process (e.g. AST vs ASM) may affect properties of neutral fitness regions, and so impact the algorithms ability to find a repair. This paper offers the following contributions:

1). A quantification of mutational robustness for a variety of programs across a number of representations

2). An evaluation of mutational robustness on the success of automatic program repair with genetic algorithms

3). Exploratory research into forming new representations (genotypes) for non-robust code.

The structure of this paper is as follows. In *Section 2* we introduce background information relating to GP program repair, neutral fitness landscapes, and mutational robustness. *Section 3* provides a motivating example for how the properties of neutral fitness landscapes may be relevant in finding program repairs. In *Section 4*, we discuss experiments quantifying mutational robustness and the affect of such robustness on GP's search. Finally, in *Section 5* we end with our conclusions.

2. Background

Here we review automatic program repair with genetic programming, as well as *mutational robustness* and *neutral fitness landscapes*. Further, we discuss how these latter relate to the idea of *evolvability*.

2.1. **Genetic Programming and Automatic Program Repair.** Genetic programming (GP) discovers new programs using a computationally-defined analog to the evolutionary process [5]. The process iterates across generations of program variants, each assigned a *fitness*, or approximation of desirability. The high-fitness variants are copied over to the next generation, and population variety is introduced through computational representations of biological crossover and mutation. The

process runs until a pre-defined time limit is reached or an acceptable variant is found.

Earlier work on automated program repair demonstrated a framework for GP-based patch generation [10] and explained the evolutionary characteristics of the approach [4]. In this work, a GP evolves programs that avoid a particular bug. The GP system begins with a working program in C, an input that causes the program to behave incorrectly, and a set of regression tests that the program passes. Individuals in the GP system's population are variants of the original buggy program. An abstract syntax tree (AST) represents each program variant. Test cases serve as the fitness function, where a *test case* consists of input to the program (e.g., an image to be processed, or an HTTP request to be served) and an *oracle comparator* function that defines the correct program response [3]. A program *passes* a test case if it produces the expected output when run on the input, as defined by the oracle comparator; otherwise, it *fails* the test case. A *positive test case* is a standard (regression) test case that encodes correct program behavior; the program's existing test suite comprises the positive test cases. A *negative test case* is a program input that demonstrates the bug and a comparator that detects it. To compute a variant's fitness, its AST is printed as source code, compiled, and then run against test cases in a sand box. The weighted sum of the total number of positive and negative test cases passed is its fitness. A *repair* is a program variant that passes all test cases. As a post-processing step, redundant code can be eliminated from the repair with program minimization techniques (the *final repair*).

2.2. **Mutational Robustness.** Mutational robustness is defined as the likelihood of a change in genotype affecting a change in phenotype. [6] That is, something is mutationally robust if given various mutations to its underlying genotype, its phenotype stays constant. In the context of C-programs, we use the underlying C code as a genotype and program behavior as a phenotype.

Naturally, there are many possible genotypes available to us, as code can be represented at different levels of abstraction (e.g. binary, assembly, string, AST, and so on). In this work, we make use of ASTs as our genotypic representation. In this case, individual pieces of the genotype are simply nodes on the AST in the form of program statements, such as code blocks, variable assignments, or various logical operations.

As a way of encoding proper program behavior — encoding a phenotype — we make use of regression test suites, a set of test cases designed to exercise all desired program functionality. Such test suites are often used in industry to insure that code modifications have not broken other parts of the program.

To measure mutational robustness, we repeatedly apply a single mutational operator to the program in question, and record as a result, how likely a program is to have changed its behavior as a result of this mutation. More specifically, the mutation operators we use are *append* (copy one piece of the genotype elsewhere within the genotype), *swap* (swap two pieces of the genotype) and *delete* (delete a piece of the genotype). A piece of the genotype is simply a node on the AST.

Although, some degree of mutational robustness is necessary for evolution to take place, there does exist a tradeoff between robustness and evolvability. Consider that in the extreme case of a completely robust genotype, no phenotypic phenomena might change, and so no evolution could possibly take place. Nevertheless, in the context of automatic program repair, a potential lack of program robustness is a far greater concern. It is not presumed difficult, after all, to introduce bugs through arbitrary permutations to program source code.

2.3. **Neutral Fitness Landscapes.** A neutral fitness landscape describes the space of different genotypes that share a single phenotype. [9] In this space, two genotypes may differ in any arbitrary number of places, so long as their phenotypes resolve identically. Intuitively, genotypes that are highly mutationally robust will have larger areas of neutral fitness, as more changes can can be made to such genotypes without

altering phenotypic behavior. If one examines the extent of a neutral fitness landscape, one finds much the same tradeoff here as was evident for mutational robustness. A genotype possessing a vast and shallow neutral landscape is quite unlikely to undergo significant evolution, whereas a genotype with a small peaked neutral fitness landscape is likely to remain stuck in a local maxima.

3. Motivation

3.1. **The General Evolution of Program Code.** Finding a high degree of mutational robustness in program code would have strong implications for evolutionary computing and software engineering. Traditionally, software is viewed as quite fragile. It is the common assumption that undirected change — a necessary component of the evolutionary process — will likely, if not inevitably break code's functional behavior. If, rather, it is the case that making small random changes to program source code usually does not result in functional changes, then incorporating broader evolutionary principles into computing seems a far more viable prospect.

Evolutionary processes, after all, have the potential to be used for more than repairing bugs. They might be used for such diverse purposes as security (computer immune systems), code diversity, and perhaps even the development of new code functionality. Evidence that software is robust, while far from sufficient, is necessary for such evolution to take place.

3.2. **Automatic Program Repair.** Investigations into mutational robustness may provide insight into repairing a certain class of difficult program bug. That is, there exists a kind of program fault for which the sequence of mutations necessary to a repair produces an entirely broken program — failing all positive regression tests — at some intermediate stage. For the most difficult members of this class of bugs, any possible permutation of the repair sequence produces a broken intermediate program state.

As this class of program faults is distinguished by such distinctly non-robust properties, investigation into mutational robustness should shed light on various aspects to this class, and may provide a means for judging among possible improvements to GP to account for the difficulty of this class. To better illustrate the problem here, consider the following program source code:

Figure 1: *Three-step non-robust bug*

```
1  int main(int argv, char * argc[]){
2    int x = atoi(argc[1]);
3    int p1 = 0;
4    int p2 = 0;
5    int p3 = 0;
6    int now = p1+p2+p3;
7    // Positive test evaluates x = 1
8    if( x == 1 ){
9      printf("%d:%d:%d\n",x,p1-p2-p3,now==p1+p2+p3);
10   }
11   // Negative test evaluates x = 666
12   if( x == 666) {
13     printf("%d:%d:%d:%d\n",x,p1+p2+p3,p1-p2-p3,now==p1+p2+p3);
14   }
15   p1 = 7;
16   p2 = 3;
17   p3 = 4;
18 }
19 }
```

To resolve this bug, program statements 15, 16, and 17 need to be moved after statement 5. However, simply moving one statement at a time will produce a program

that fails all positives tests at some intermediate state. This is because moving the necessary statements, one by one, fails the test of equality required of the positive test case, while not yet meeting the equality required of the negative test. It is this kind of non-robust bug for which an analysis of mutational robustness may provide some insight.

4. Experiments

4.1. **Benchmarks.** Our experiments make use of a diverse set of 8 benchmark programs, covering a wide range of application areas and code sizes. The programs *deroff*, *look*, and *uniq* are commonly used unix utilities between 1000 and 2000 lines of code. A simple implementation of *gcd* calculates the greatest common denominator of two numbers in only 22 lines. We have a computational biology program, *leukocyte* of approximately 400 lines, two programming language interpreters, *potion* (50,000 lines) and *vyquon* (4,000 lines), as well as a key-value store, *redis* (36,000 lines) which is commonly used in industrial practice.

4.2. **Mutational Robustness on Weighted Path.** A first experiment seeks to quantify the degree of mutational robustness in programs undergoing the automatic program repair process. We calculate the percentage of mutations on each selected benchmark program that are neutral. These *neutral mutations* signify no change in behavior as detected by our regression test suites. Notably, we continue to make use of the weighted path in this experiment, and thus the mutations each program undergoes are concentrated on those program statements most likely to affect its negative test case. The results follow:

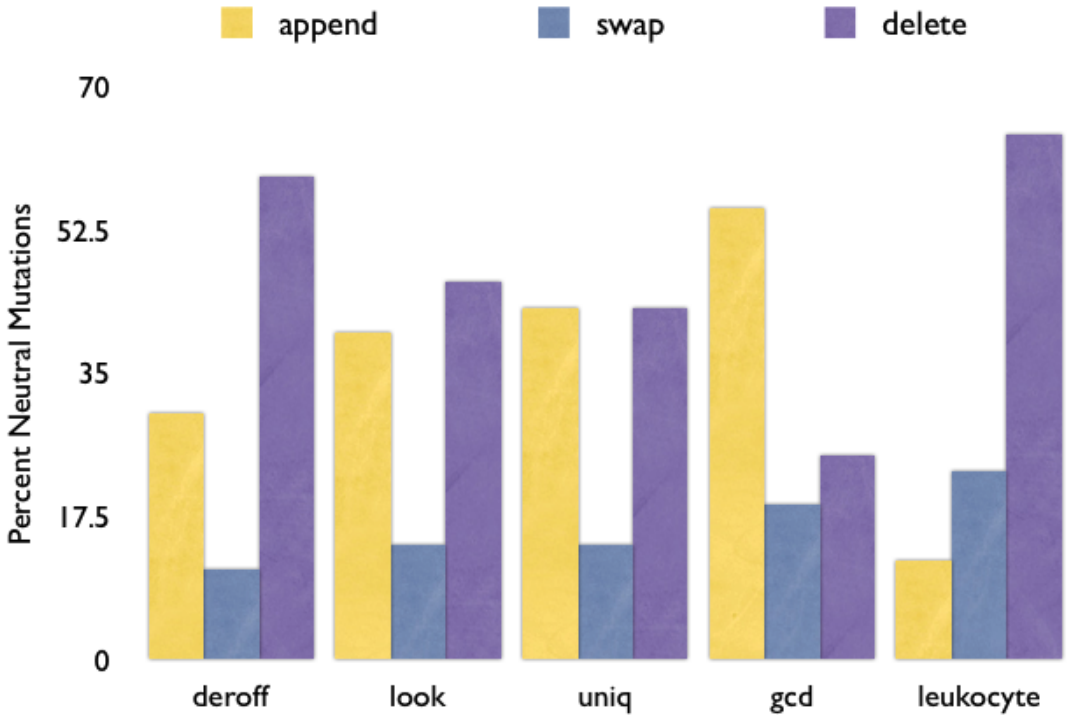
Figure 2: *Mutational Robustness with Path Weighting*

Program	Neutral*	Deletes	Appends	Swaps
deroff	31%	59%	30%	11%
look	43%	46%	40%	14%
uniq	34%	43%	43%	14%
gcd	34%	25%	55%	19%
leukocyte	39%	64%	12%	23%
average	36%	47%	36%	16%

* Percent neutral mutations on 1000 trials

We find, to our surprise, that over 36% of mutations are neutral mutations — mutations that do not affect program behavior. We next deconstruct the neutral mutations by operator, measuring what percentage of these mutations occur for each distinct mutation operation (append, swap, or delete).

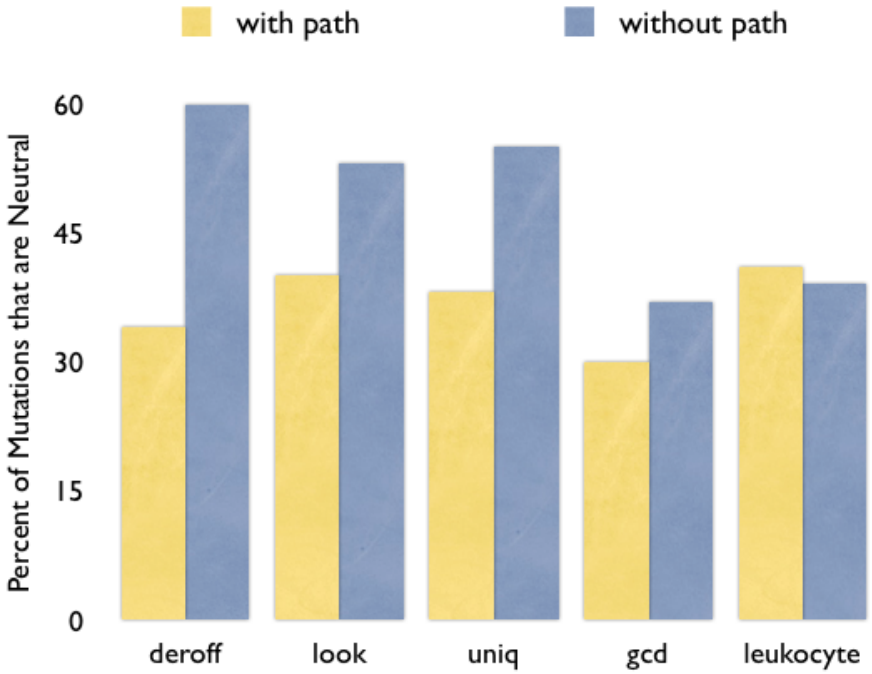
Figure 3: *By Mutation Operators*



We find that the majority of neutral mutations are made up of deletes, followed by appends, and lastly by swaps. The presence of deletes in the majority is particularly surprising, as it suggest as a potential implication that programmers write unnecessary code. A minority for swaps, however, is not so unexpected, if one considers that a swap has the more complicated property of altering the program at two points.

4.3. **Mutational Robustness off the Weighted Path.** Next we run a similar experiment without using the weighted path. Thus, here we consider each program statement as equally likely to undergo mutation, with no preference for those statements implicated in the negative test case. This second experiment, then provides something more akin to a universal benchmark of robustness, as we do not take into account those special properties of program representation found in the GP repair process.

Figure 4: *With and Without Weighted Path*



We find that program robustness increases more-or-less uniformly, with no significant difference in the distribution of neutral mutations across appends, swaps, and deletes. This result is not surprising for two reasons. First, the weighted path is

designed to focus the attention of mutations on those segments of program code implicated most directly in program functionality and behavior, so in taking away the mutational privilege from these statements it is only natural that neutral mutations become more likely. Further, without the weighted path, we open up the possibility of mutation for a wider variety and number of program statements, many of which may exist in less commonly executed run-time paths.

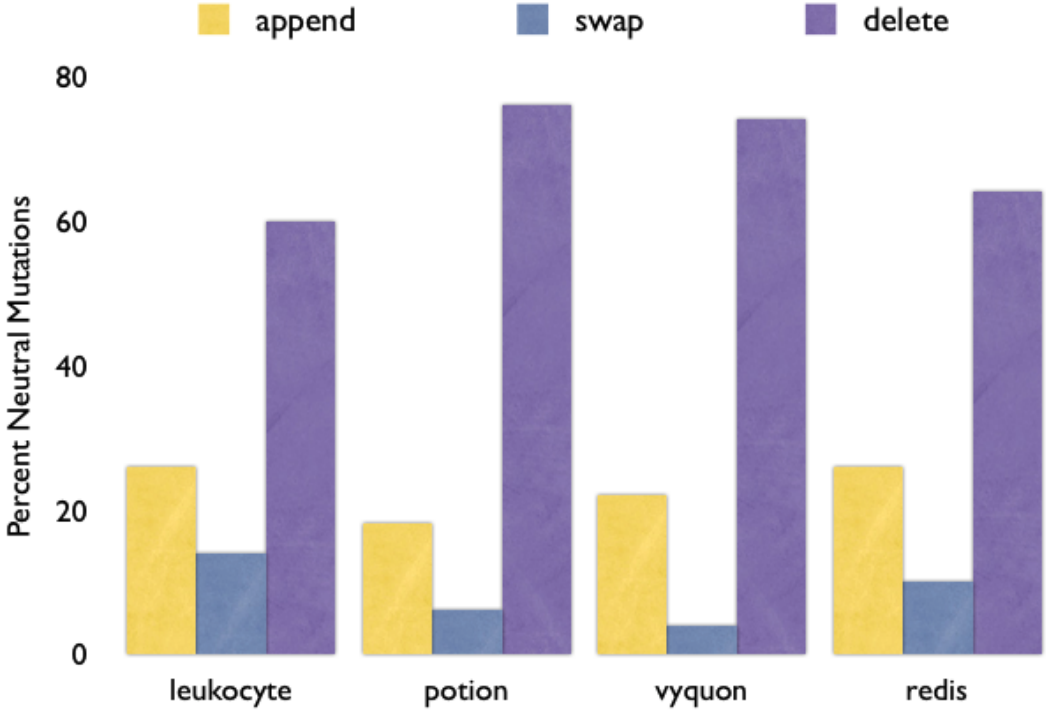
4.4. **Mutational Robustness for Large Test Suites.** Next, we address the worry that the test suites used for our previous benchmark programs are not adequate for evaluating a program's phenotype, and thus produce a flawed measure of mutational robustness. Consider that if the regression test suite in use measured only some very small portion of program functionality, then those mutations that we count as *neutral* may not in fact be neutral at all. Thus, to mitigate these concerns, we measure the mutational robustness of a few additional programs, picked for their more comprehensive regression test suites. We also augment the *leukocyte* program with a larger test suite. Among the two new program are two interpreters and a key-value store. Both interpreters may be downloaded with built in suites designed to exercise their functionality. The *redis* program likewise has a large suite of over 300 test cases, exercising complex program functionality.

Figure 5: *Mutational Robustness: Large Test Suites*

Program	Neutral	Deletes	Appends	Swaps
leukocyte	35%	60%	26%	14%
potion	39%	76%	18%	6%
vyquon	32%	74%	22%	4%
average	35%	70%	22%	8%

We find similar levels of mutational robustness for these programs, suggesting that our measures are reasonable within the earlier benchmark. We next look at the contributions of various mutation operators.

Figure 6: *By Mutation Operators: Large Test Suites*



When breaking down the neutral mutation of these new programs, we find that the distribution of such mutations across our operators has been exaggerated. Deletes contribute even more significantly here, holding a clear majority among those mutations on the neutral landscape. Likewise, swaps become less likely. These curious results once more suggest the odd implication that elements of a program's source code would seem to be redundant or unnecessary.

4.5. **New Genotype Representations.** Finally, we investigate the impact of a new genotype with a higher degree of mutational robustness on the difficult non-robust bugs described in an earlier section.

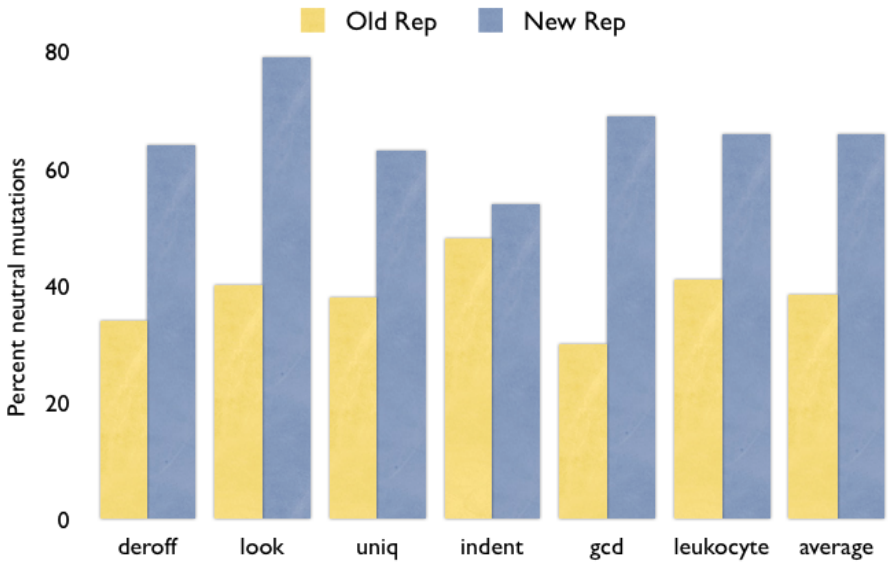
We draw inspiration from the diploid nature of chromosomal structure, and attempt to create a computational analogy for our ASTs. Previously, each individual was

composed of a single program-tree, and we introduce two per individual, where their genetic material must be combined before running the program. This analogy may be implemented in a myriad of ways, and we tested two such approaches.

The first approach arbitrarily declares some statements on each tree to dominant, and some statements to be recessive. Thus, in evaluating a single individual, its two trees would be combined according to its dominant and recessive traits. The second approach applied a 'fallback' strategy, executing the second tree only if the the first tree fails. In both cases, our mutation operators may be applied across both trees.

As a first step in out evaluation of these representations, we measure the new levels of mutational robustness:

Figure 7: *Compare Representations*



Unsurprisingly, we find that the new representations are more robust. Thus, we next run these new representations on 'two-step' and 'three-step' implementations of *non-robust* bugs, one of which was shown previously in figure 1. We find that the new representations are three times more likely to find a repair for the two-step bug, but no approach proved able to solve the three-step problem.

5. Conclusions

We find that programs are far more robust than intuition might suggest. In evaluating random mutations the source code of 9 programs, we find that more than 30% of the mutations are *neutral*, that is, they do not affect program behavior. Further, this number holds for decidedly thorough and complicated test suites.

Although test suites may be inherently limited in their ability to detect program bugs — and thus limited in their ability to serve as a program genotype — we suggest that the levels of mutational robustness we find are high even in light of such limitation. Further, investigations into robustness have shown some promise in resolving existing difficulties for the GP automatic program repair technique. In future work we will attempt to provide a better explanation for the robustness we see, and more practical implications for the GP repair process.

6. Acknowledgements

Thanks to my mentor Stephanie Forrest!

References

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[3] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 1999.

[4] Stephanie Forrest, Westley Weimer, ThanhVu Nguyen, and Claire Le Goues. A genetic programming approach to automated software repair. In *GECCO*, pages 947–954, 2009.

[5] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[6] Rebecca Montville, Remy Froissart, Susanna K. Remold, Olivier Tenaillon, and Paul E. Turner. Evolution of mutational robustness in an rna virus. In *PLOS*, November 2005.

[7] C. V. Ramamoothy and W-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.

[8] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[9] Eric van Nimwegan, James P. Crutchfield, and Martijn Huynen. Neutral evolution of mutational robustness. In *PNAS*, March 1999.

[10] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–367, 2009.