Dissertation Proposal: Evolving Robust Software

Eric Schulte

August 6, 2012

Contents

1 Introduction

Over the past fifty years software developers have been selecting, reusing and modifying efficient and robust software development tools, code and design patterns. This history of development, through a process mirroring natural selection, results in some surprisingly biological features of software which this proposed work will investigate and exploit.

The dual goals of this investigation are increased understanding of existing software, and new tools for software development and maintenance. I will focus on properties that are exhibited by evolved complex systems and arise through natural selection and adaptation to survival in the world. Some properties, such as the amenability to random evolutionary processes and robustness to variability in constituent parts, will be shown to be applicable to engineered software systems.

My hypothesis is that similar properties lead to the persistence and replication of both living and engineered systems. These properties include robust functioning in an environment and the ability to adapt (or be adapted) to new environments. Specifically, like their biological counterparts, software artifacts which have stood the test of time including applications, interfaces, operating systems, programming languages, compilers and linkers all display robustness and adaptability.

I discuss recent methods of software development and maintenance that make use of software robustness, and I propose new tools and techniques which leverage this natural robustness of software. By extending our understanding of the similarities between engineered software systems and evolved biological systems, software may be engineered to be more robust and evolvable, benefiting software developers and consumers.

Motivation: Over the previous half century the production and maintenance of software has emerged as an important industry, consuming the efforts of 1.3 million software developers in the US alone in 2008, a number projected to increase by 21% by 2018 [5]. Investigating the evolved properties of software systems will challenge commonly held folk wisdom of software fragility, leading to new tools and techniques such as those discussed in the preliminary and proposed work below. My research goals are to produce increasingly robust software and to reduce the cost of software maintenance.

Investigating software robustness could shed light on fundamental properties of the robustness of evolved systems. The field of *digital evolution* encompasses the design and analysis of computational models of biological evolution, permitting evolutionary time frames, controls, and metrics that are not feasible in-situ. Real world engineered software offers not just a model, but an *instance* of a robust evolved system [45]. Real world software is more complex than artificial models, has non-contrived fitness functions, and a development shaped through real world competition.

Opportunity: Recent developments in software engineering and evolutionary biology make this an opportune time to pursue this topic. The two fields increasingly overlap, with biologically inspired computational techniques being applied to the maintenance of software systems [54], and software performance being discussed in biological terms [1].

The software engineering community is moving from ideas of provable correctness towards notions of sufficient correctness and adaptability [43, 42, 40, 44], and my proposed research is well poised to take advantage of and promote this shift in priorities.

Overview: In the remainder of this proposal I first review related work (Section 2). The research plan is divided into two chapters, focusing on an investigation of software mutational robustness (Section 3) and development of applications that leverage software robustness (Section 4). A workplan and timeline for completion of the proposed research is presented (Section 5), and the proposal concludes with a discussion of the expected impact of this research and possible future extensions (Section 6).

2 Related Work

This research draws from previous studies of both biological and computational systems. The study of evolvability and robustness in biology provides the concepts and terminology with which I will investigate these aspects of computational systems.

I will make use of previous applications of computation to biological systems, and of biological processes to computational systems. The field of *digital evolution* has demonstrated the use of software systems to sheds light on the nature of robustness and evolution in biology. The field of evolutionary computation (EC) has successfully applied a computational analog of natural selection to the design, repair and optimization of software systems. This proposed work will extend these fields through a focus on evolution in extant software within the standard software development tool chain rather than on systems specifically designed to study evolution.

Recent work in software engineering has shifted focus from formally provable properties of software towards empirically observable properties. Due to our use of loose fitness metrics and unsound program transformations, our work may be seen as part of this growing trend. I now review the most relevant results from each of these fields, moving from the biological to the computational.

2.1 Robustness and Evolvability in Biology

The ability of living systems to maintain functionality across a wide range of environments and to adapt to new environments is unmatched by man-made systems. A great deal is known about the relationship between robustness and evolvability in living systems. This section reviews the basics of this field of study highlighting particularly relevant elements of this large body of work.

Living systems include both a *genotype* and a *phenotype*. The *genotype* is the genetic information which specifies a living organism. The resulting organism and its interaction with the world is the organism's *phenotype*.

Both the genetic and phenomenal realms have associated types of robustness. Genetic robustness, or *mutational robustness*, is the ability of a genotype to consistently produce the same phenotype in the face of perturbations to its genetic material. This robustness can be achieved in different ways and on different levels. At the lowest level, important amino acids are over represented in the space of possible encodings and similar codings give rise to similar amino acids [28]. At higher levels, vital functions are often degenerate (i.e., supported by diverse partially redundant systems) [13] e.g., in the nervous system no two neurons are *equivalent* but no single neuron is *necessary*. Finally, many mechanisms have evolved that buffer changes, e.g., metabolic pathways whose outputs are stable over wide ranges of inputs [51].

Many of the biological mechanisms responsible for mutational robustness also contribute to *environmental robustness* [32], or the ability of a phenotype to maintain its behavior in the face of environmental perturbations. There is a strong correlation between genetic and environmental robustness, likely because both types of robustness are effects of the robustness inherent in underlying biological mechanisms [26].

Mutational robustness appears to be an evolved feature [8, 56], that is, evolution tends to increase the mutational robustness of important biological components. Although it is unlikely that mutational robustness is explicitly selected for as a protection against mutation (due to the low mutation rates in most populations), it seems likely that it arises as a side-effect of evolution for environmental robustness [50]..

The interplay between mutational robustness and evolution has many facets. On its face, mutational robustness inhibits evolution by making it less likely that any given genetic modification will have phenotypic effects. While true in the extreme, moderate levels of mutational robustness increase the genetic diversity of populations enabling evolution [53].

A mutationally robust organism has many genotypes that map to the same phenotype. Consider the *space* of genotypes to be a high-dimensional discrete space in which each point is a genotype, and the immediate neighbors of each point are the genotypes that are reachable through a single mutation. Regions of this space constituting genotypes with the same fitness are called *neutral spaces*. Populations tend to spread out in large neutral spaces via *drift*, accruing variety and novel genetic material. This accrued genetic material is hypothesized to play an important role in evolutionary innovation [7, 52, 33, 37], and provides the genetic fodder for large evolutionary changes.

2.2 Evolutionary Computation

Some of the properties of evolution mentioned above (e.g., [37]) are not based on direct observation of biological systems, but rather on computational models of biological systems. The evolutionary time frames and the degree of environmental controls required for such experiments are simply not experimentally achievable with biological systems. In *digital evolution* computational models of evolving populations represent genotypes using specialized assembly languages in environments in which their execution (phenotype) determines their reproduction or extinction [38].

In addition to illuminating evolution in biological systems, this work has generated insight into those properties of programming languages that might be amenable to evolution [36]. Although the languages (or "chemistries") studied in these computational environments are far from traditional programming languages, some insights do transfer, such as the brittleness of absolute versus symbolic addressing. Work in the virus community [24] has produced similar claims about the evolvability of traditional `X86` assembly code. One of the contributions of this proposed research will be to confirm some of these claims and intuitions and to disprove others.

The evolution of computer programs in a process mimicking natural selection is known as Genetic Programming (GP) [21, 29], and has been applied to a number of real world problems [41, Chapter 12]. GP languages are often much simpler than those used by human programmers and rarely resemble regular programming languages, although in some cases machine code has been evolved directly [30].

Recently, GP techniques have been applied to the repair of extant real world programs [54]. This proposed research extends this work, incorporating new program representations and applications of evolution to program modification and synthesis within traditional software development environments and programming languages.

2.3 Software Engineering

This proposed research can be seen as part of a larger trend in Software Engineering to emphasize acceptable performance over formal correctness. I review recent work in this area, and highlight the tools and metrics most applicable to our proposed work.

In *failure oblivious* computing [44] common memory errors such as out-of-bounds reads and writes are ignored, or handled in unsound ways which are often sufficient to continue operating. Examples of such techniques include wrapping out-of-bounds memory references modulo the available memory address range, treating memory as a hash table in which addresses are merely keys, creating new entries when needed, and returning random values from reads from uninitialized memory. By preventing common errors such as buffer overruns these techniques have been shown to increase the security and reliability of some software systems. In many cases security and reliability may be more important than correctness.

Beal and Sussman take a similar approach proposing a system for increasing the robustness of software by pre-processing program inputs [1]. Under the assumption that most software operates on only a sparse subset of the possible inputs, they propose a system for replacing aberrant or unexpected inputs with fabricated inputs remembered from previous normal operation. This system of input "hallucination", is shown to improve the robustness of a simple character recognition system.

While the previous systems learn and enforce invariants on program input, the clearview system [40] learns invariants on trace data extracted from a running binary using Daikon [14]. When these invariants are violated by an attack, bug or an exploit, the system automatically applies an invariant-preserving patch to the running binary which ensures continued execution. This technique was evaluated against a hostile red-team and was able to successfully repair seven out of ten of the red team's attacks [40].

All of the approaches mentioned above are applied to executing software systems or software phenotypes. There are also techniques which apply to the pre-compiled software source-code, or genotype. One such technique is *loop-perforation* [35] and *dynamic knobs* [20]. In loop-perforation software is compiled to a simple intermediate representation (IR), looping constructs are found in this IR and modified execute the loop fewer times by e.g., skipping every nth loop execution. This technique can be used to reduce energy and runtime costs of software while maintaining probabilistic bounds of expected correctness. This work is notable for introducing program transformations that are not formally correct but are rather predictably probabilistically accurate.

Another important class of program transformation techniques, and the topic of much of this proposed work, includes methods of evolutionary program repair and optimization [54, 31, 55]. These approaches apply Evolutionary Computation (EC) techniques to the source code of existing software projects. Test suite based fitness functions are used to evaluate the effects of program transformations, allowing a much larger range of program modification than traditional formally provably semantic-preserving transformations.

For decades the *mutation testing* community has used extant program test suites to evaluate program mutants. Although their techniques closely resemble those discussed herein, their motivations and interpretation are very different. Under the mutation testing paradigm, test suites are evaluated by their ability to detect automatically generated program mutants [25]. This presumes that all program mutants are either buggy or equivalent to the original program — the detection of equivalent programs is a significant open problem for the mutation testing community [4]. By contrast my proposed work will seek to exploit neutral and beneficial variation in program mutants for use in software development and maintenance.

3 Mutational Robustness

In preliminary work, we show software to be *mutationally robust* [45]. We define the mutational robustness of software as the percentage of software mutants that are functionally equivalent to the original program. We demonstrated software to be robust to mutation of its source and assembly code. This insight has multiple implications for both the nature of software and potential new software tools. In the remainder of this section I further explain the concept of *software mutational robustness* and share preliminary results in Section 3.1, and propose ways to extend this insight to increase our understanding of software in Sections 3.2 and 3.3.

3.1 Software Mutational Robustness

Let “syntax space” be a space of the *text* of program source code, and let “semantic space” be a space of the *functionality* of program behavior. Compilation (with a set compiler and flags) will then be a function from syntax space to semantic space. Given a program we apply syntactic mutations and observe the semantics of the resulting mutations.

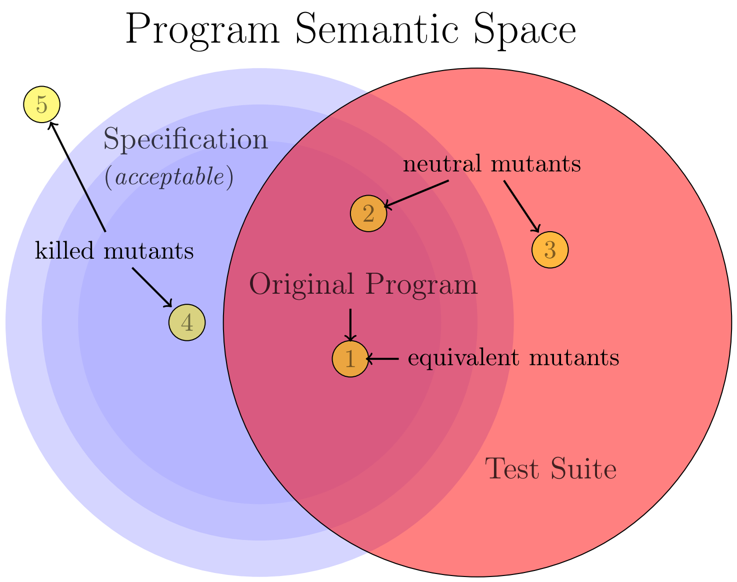


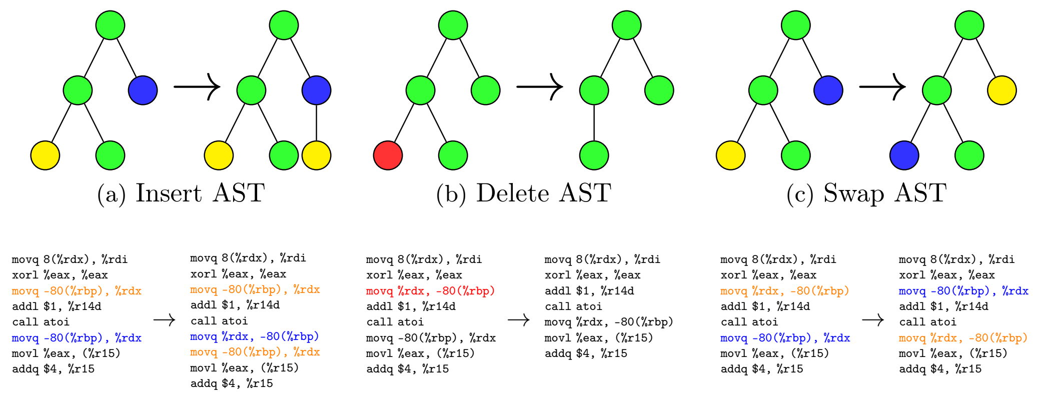
Figure 1: Semantic Space around a program showing the specification (*lavender*) the test suite (*red*) and various program mutants (*yellow*).

Figure 1 shows the semantic space surrounding a program. The lavender “Specification” region contains those programs with acceptable semantics, whether defined through a formal specification or loosely by the developers or users of the program. The red region holds those programs with acceptable behavior as defined by the test suite. Ideally the original program lies within the intersection of these two regions, and their exclusive disjunction is small

The mutation testing community uses mutation operators specifically designed to produce faulty program mutants. Mutation testing assumes that all mutants are either faulty or equivalent, i.e., the lavender *Specification* region in Figure 1 does not extend beyond the point of the original program. A “mutation adequacy score” [25] accesses the quality of the test suite as the percentage of non-equivalent mutants which fail some test. Through seeking to optimize this score mutation testing drives the *Test Suite* region to similarly approach the single point of the original program in semantic space.

By not allowing for the possibility of non-faulty non-equivalent mutants (i.e., by not acknowledging mutants 2 or 4 in Figure 1), the mutation testing approach ignores the functional (or neutral) variants which much of this work will seek to exploit[1].

[1]By pointing out this oversight I do not mean to diminish the importance of the mutation testing community, or its impact on both Software Engineering research and industry practice of the previous 30 years. My


(a) Insert AST (b) Delete AST (c) Swap AST

```
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)
movl %eax, (%r15)               movq -80(%rbp), %rdx
addq $4, %r15                   movl %eax, (%r15)
                                addq $4, %r15
```

(d) Insert ASM

```
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 %rdx, -80(%rbp)
movq -80(%rbp), %rdx            movl %eax, (%r15)
movl %eax, (%r15)               addq $4, %r15
addq $4, %r15
```

(e) Delete ASM

```
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
```

(f) Swap ASM

Figure 2: Mutation operators: Insert, Delete and Swap applied to both abstract syntax trees whose nodes are C statements parsed from source code and to linear arrays of assembly instructions parsed from compiled assembly code.

We define software mutational robustness as the percentage of random mutations to a given software instance that leave the observable functionality unchanged. We assume that mutants which satisfy the test suite will also satisfy the specification, and thus determine mutant fitness using the software's existing test suite. Only those variants that successfully compile and pass all tests in the test suite are considered *neutral*. Mutation operations are restricted to those portions of the program exercised by the test suite to ensure that all mutations will have some affect on software runtime behavior as exercised by the test suite.

In preliminary work we tested mutational robustness at two levels of software representation; abstract syntax trees parsed from source code and linear sequences of compiled assembly code instructions. The three mutation operations are shown applied to both of these representations in Figure 2.

3.1.1 Experimental Results

We tested the mutational robustness of 22 programs including production software projects, the Siemens benchmark suite [22], and a small number of hand-crafted exhaustively tested programs. Each of the three classes of programs serves a different purpose. The sorting algorithms have complete test suite coverage ensuring that each statement, branch, and assembly instruction is evaluated by the test suite. The Siemens programs provide for comparison of our work to the large amount of previous work in mutation testing. The Siemens programs also demonstrate that our results apply to *extremely* well tested software in which each branch and def-use pair is covered by at least 30 test cases. The third class of programs demonstrates that mutational robustness is a property of real world software programs used in everyday systems. The results of these experiments are shown in Table 1.

For each program, at both the AST and ASM level, we generated at least 200 unique program variants using each of the three mutation operations (*insert*, *delete* and *swap*). To

work builds upon the existing mutation testing research program and seeks to provide new applications for many existing mutation testing tools, e.g., Compiler-Integrated Techniques [10] or *super-mutants* [49] could be used to efficiently ship and run entire populations of diverse program variants (Section 4.1).

Program	ASM LOC	C LOC	# Test	% Cov.	AST Rbst.	ASM Rbst.	Description
Sorting algorithms							
bubble-sort	184	34	10	100	27.3	25.7	integer sorting
insertion-sort	170	29	10	100	29.4	26.0	integer sorting
merge-sort	233	38	10	100	29.8	21.2	integer sorting
quick-sort	219	38	10	100	28.9	25.5	integer sorting
Siemens Benchmarks							
grep	28776	10929	119	24.9	50.0	36.7	text search
printtokens	2419	536	4130	81.7	21.2	25.8	lexical analyzers
schedule	922	412	2650	94.4	34.4	29.1	priority schedulers
sed	17026	8059	360	42.0	33.0	25.6	text manipulation
space	18098	9126	13494	91.1	37.7	32.1	array def. lang. int.
tcas	544	173	1608	96.2	33.5	25.9	collision avoidance
Real World Programs							
bzip2 1.0.2	18756	7000	6	35.9	33.0	26.1	compression
— *(alt. test suite)*			22	71.0	46.4	23.6	
ccrypt 1.2	15261	4249	6	29.5	33.0	69.7	encryption
— *(alt. test suite)*			16	40.4	34.6	69.7	
imagemagick 6.5.2	6128	147	145	0.8	33.3	66.3	image manipulation
jansson 1.3	6830	2975	30	28.8	33.3	28.0	data serialization
leukocyte	40226	7970	5	45.4	33.3	39.9	computational biology
lighttpd 1.4.15	34165	3829	11	40.1	61.5	56.9	webserver
nullhttpd 0.5.0	5951	5575	6	64.5	41.5	37.8	webserver
oggenc 1.0.1	299959	59094	10	38.4	33.4	22.1	audio codec
— *(alt. test suite)*			40	58.8	40.5	72.3	
potion 40b5f03	80406	15033	204	48.4	33.3	48.9	language interpreter
redis 1.3.4	44802	17203	234	9.2	33.3	34.0	key-value data
tiff 3.8.2	22458	1732	10	15.4	33.3	90.4	image manipulation
vyquon 335426d	20567	4390	5	50.6	33.3	69.0	language
total or average	664100	158571	23151	40.9	33.9 ± 10.4	39.6 ± 21.5	

Table 1: Programs with mutational robustness of first-order (one-step) mutations. The "LOC" columns report the size of the program in terms of lines of C source code and lines of compiled assembly code. The "# Test" and "% Cov." 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 "Rbst." columns report the percentage of all first-order mutations that were neutral. The ± values in the bottom row indicate one standard deviation.

generate these variants, mutation operations were applied at locations chosen randomly from those visited by the test cases. On average, roughly 37% of randomly generated program variants in this sample were neutral.

One possible weakness of these results is the use of a program's existing test suite to determine the functionality of program variants. It is often the case that existing program test suites are not sufficient to detect flawed mutants — this intuition underlies the entire field of mutation testing. To address this potential weakness we included programs with very good test suites (the sorting and Siemens programs), and we study how mutational robustness correlates with test suite coverage (Figure 3). A visual inspection of this table shows no obvious correlation, and importantly, even with 100% statement coverage at least 20% of program variants remain neutral.

I propose to extend this work by investigating factors affecting mutational robustness (Section 3.2) and those properties of software that may be related to mutational robustness (Section 3.3).

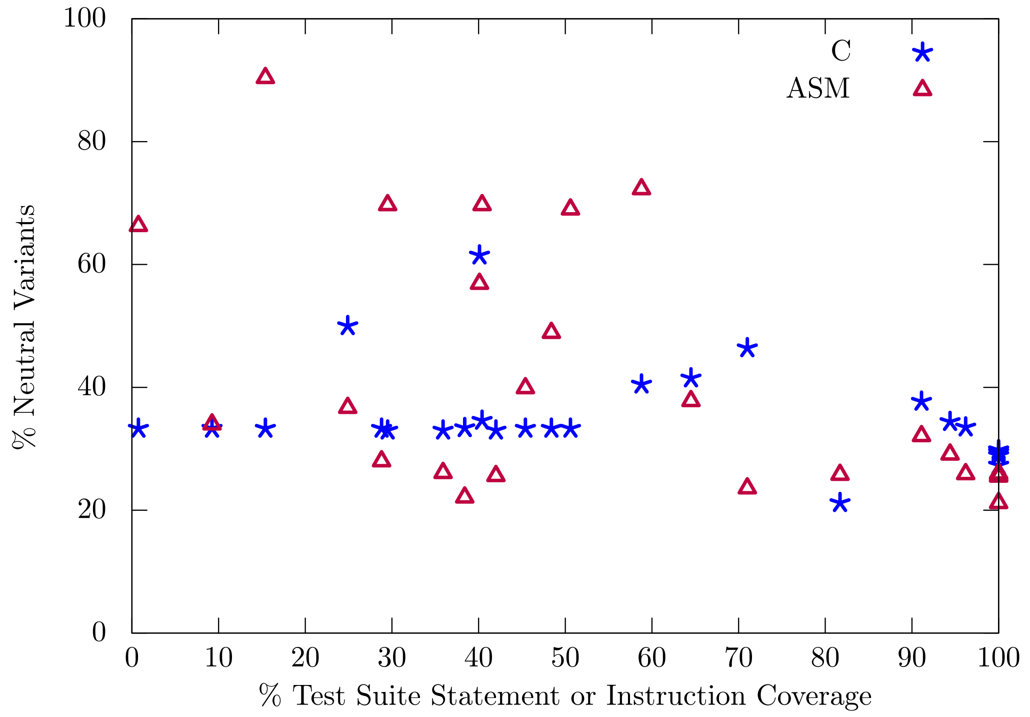


Figure 3: Software mutational robustness from Table 1 shown by % test suite coverage. The Pearson correlation coefficient between coverage and robustness is -0.41 (≈ 17%).

3.2 Causes of Mutational Robustness

There are a number of properties of biological systems thought to cause high levels of mutational robustness. Two such factors which I will discuss below are: the mechanisms of DNA expression and the history of development through evolution.

The mapping from DNA to amino acids has evolved in such a way that similar DNA sequences give rise to similar proteins [28]. Similarly, the mappings from RNA sequences to the functional shapes they produce have properties such as local continuity (promoting incremental change) and global reachability (promoting novel functionality) which are themselves thought to be amenable to evolution. [46, 23].

> If one were to design the ultimate evolvable molecule that carries information and is engaged in functional interactions, it would ideally require two features: (i) capability of drifting across sequence space without the necessity of changing shape; and (ii) proximity to any common shape everywhere. These are precisely the features that statistically characterize the mapping from RNA sequences to secondary structure. [46]

To the degree that such properties are the consequences of the evolutionary origin of DNA and RNA, analogs should be found in other systems shaped by natural selection. The remainder of this section will investigate the hypothesis that as a product of natural selection, the mechanisms of expression of software systems should increase robustness and evolvability. I will then measure the effects of increased evolutionary development on robustness and evolvability in software developed using GP.

3.2.1 Level of Representation: Source VS. Compiled

The processes of compilation and linking of program code can be thought of as roughly analogous to the biological processes of DNA transcription and translation through which a biological *genotype* encoded in DNA is translated into protein structures whose interaction gives rise to a *phenotype* of behavior in the world. In the context of software the *genotype* is encoded in source code which is then compiled to generate assembly or machine code sections which are linked together giving rise to the *phenotype* of an executing program.

The biological processes of transcription and translation both contribute to an organism's mutational robustness. I hypothesize that the same is true of software. If this is the case, then I hypothesize both compilers and linkers contribute to software mutational robustness. In the same way that the mappings from DNA and RNA to proteins and secondary structures buffer small genetic mutations the processes of compilation and linking may buffer the effects of mutations in source code.

I propose to test this proposition by comparing the robustness of mutations to source code (before it is compiled and linked to form an executable), assembly code (before it is linked to form an executable) and ELF files (which have previously been compiled and linked). I hypothesize that mutations directly to ELF files will be the least robust and mutations to source code will be more robust.

There are a number of threats to the validity of such an experiment. Most importantly the differences in representation across these levels of representation (e.g., source code is represented using trees while compiled executable code is represented using vectors) will require different mutation operators be used at each level. Any measure of mutational robustness is as much a measure of the mutation operators used as it is of the representation being operated upon. Great care will need to be taken to ensure that the operators are made as similar as possible, and the effects of any difference in operators are understood sufficiently well to ensure that observed differences are in fact due to differences in the inherent robustness of the underlying software representation.

If there are significant differences in the robustness of software across these levels then the next step would be an investigation of the specific mechanisms of compilers and linkers that enhance robustness. As one example, C compilers allow symbolic addressing to be used in source code even though direct addressing is required in assembly code. The use of direct addressing has been shown to limit the robustness of a language to mutation [36].

Preliminary Work In preliminary work, I implemented representations allowing evolution of compiled software assembly code (ASM), and compiled and linked ELF executables. Both representations encode genomes as linear arrays of whole assembly instructions. The mutation operators used over these representations are identical to each other but differ from those used to manipulate source code level abstract syntax trees (Figure 2).

3.2.2 By Provenance: Evolved VS. Engineered

It seems reasonable that the products of an evolutionary process would be both more robust to the changes wrought by evolutionary processes [15], and more amenable to improvement through these processes [9]. If these results generalize to software, then software artifacts programmed using evolutionary technique will be more robust (both mutationally and environmentally) than engineered software artifacts. I propose to test this hypothesis by comparing three types of software artifacts.

1. Those programmed entirely by human engineers.
2. Those programmed initially by human engineers and then incrementally evolved.
3. Those programmed entirely through an evolutionary process.

Given that the third type of software to be examined requires de novo evolution, it will likely be necessary to restrict our investigation to the relatively simple languages and algorithms amenable to development through exclusively evolutionary processes.

If it is the case that an evolutionary provenance increases software robustness, then our work may point towards the possibility of using evolution to enhance and replace components of existing engineered systems, as a means of increasing both mutational and environment robustness, as well as the ability of such systems to be improved and repaired through environmental processes.

3.3 Correlates of Mutational Robustness

Having investigated the possible causes of mutational robustness in software I will look to possible effects or correlates of mutational robustness in software. Specifically I propose to study the relation between mutational robustness and evolvability of software, and the relation between mutational and environmental robustness.

3.3.1 Robustness and Evolvability

In biological systems mutational robustness and evolvability are thought to be inextricably linked. Mutational robustness allows neutral mutations to accrue in a population storing the increased diversity required for large evolutionary breakthroughs [51, 56, 37, 7, 23, 46, 33]. However, excessive mutational robustness renders too many mutations neutral, thus inhibiting evolutionary selection of beneficial mutations.

The tension between these opposing effects has been studied in biological systems [51, 53, 32]. I propose to study the effects of varying levels of mutational robustness on the evolvability of software systems.

Such a study may indicate what level of mutational robustness is desirable in software and in which cases increased mutational robustness would be beneficial (e.g., maintaining critical behavior) or detrimental (e.g., in the face of a rapidly changing specification).

Such an experiment requires metrics of evolvability and of mutational robustness. The techniques required to measure mutational robustness are supplied by previous work (Section 3.1). Measuring evolvability is less straight-forward. Two possible approaches are seeding bugs into software projects and testing their amenability to repair through evolution, and adding test cases for previously unimplemented functionality to software projects, and testing the ability of the software to evolve the newly required functionality. With these metrics in hand, correlations can be measured in existing benchmark suites of software projects.

3.3.2 Mutational and Environmental Robustness

Many biological mechanisms are thought to be common causes of both genetic and environmental robustness. For example, metabolic pathways in biological systems often produce stable output over greatly varying sets of inputs [11]. Such buffering protects these processes from varying levels of inputs whether the cause of the variance is due to mutations elsewhere in the organism or to environmental variation.

I propose to investigate whether mutational robustness is positively correlated with environmental robustness in software. I will define the environmental robustness of software to be its ability to execute successfully in a wide range of computational environments. The computational environment will include inputs to the process (generated using fuzz testing tools), the system resources available to the process (e.g., memory, disk space, open file handles, system threads), as well as the speed and reliability with which system calls are handled.

Two different experiments can be used to test for this correlation.

First, correlation can be measured across multiple extant software artifacts. To ensure comparability of environmental robustness the software artifacts must inhabit the same environment, meaning they must implement the same functionality allowing the use of identical fuzz tests, inputs and test suites. One potential source of such diverse programs implementing a single test suite would be programs implementing widely used standards such as data formats (e.g., JSON or YAML), or communication protocols such as router software. The benchmark set developed for this experiment will be re-used in the investigation of software husbandry (Section 4.3).

Second, using traditional EC methods, I plan to separately evolve variants of existing software for increased mutational and environmental robustness using the program optimization techniques proposed in Section 4.2. The mutational robustness of those individuals evolved for environmental robustness can then be measured and compared to the original software, and vice versa.

If desirable properties of software such as environmental robustness and evolvability are found to correlate with mutational robustness, then automated methods of increasing software mutational robustness (demonstrated in preliminary work [45]) may also be used to increase software environmental robustness and evolvability.

4 Applications of Robustness

This section presents applications of the work described in the previous section. The malleability of software makes plausible a number of techniques for software diversification (Section 4.1), optimization (Section 4.2) and re-combination (Section 4.3) which would previously have seemed unrealistic. I present experiments designed to gauge the potential of each of these techniques below.

4.1 Software Diversity

Software diversity may be a useful goal in and of itself. It has, for example, been leveraged to improve software reliability through N-version systems [6]. I hypothesize that the neutral spaces of software systems could be exploited to generate diverse software variants; a process which when performed by hand can be difficult and often leads to surprisingly similar variants [27].

In previous work in this area Zachary Fry used diverse program populations to preemptively repair withheld bugs [45]. The diverse populations were generated through automated exploration of the neutral space of the original program.

In contrast to the direct benefits of diverse software populations demonstrated by Fry et al., the remainder of this section will explore two methods of leveraging diversity to *enhance* evolutionary techniques of program development and maintenance. Each technique leverages a different cornerstone of the traditional software development environment — compilers and linkers, and version control systems.

4.1.1 Compilers and Linkers

Any given piece of source code may already be used to generate a number of distinct executing programs. The compilation process determines a number of features of the final executable not fully specified by the source code.

Such divergent expressions of a single piece of source code could be used to seed an evolving population of diverse program variants. The neutral reproduction of such a population could be used to mix and match features between different compiled executables, and may evolve new features through mutation.

I propose to use a variety of distinct compilations to seed an evolutionary process with diverse variants of an input program. This increased diversity could be used to *jump-start* the production of N-variant systems, to augment the evolutionary program repair process and to provide increase genetic material to the software optimization (Section 4.2).

In each case the performance of the evolutionary processes seeded using a single program variant will be compared to the performance when seeded with diverse program compilations. This comparison should indicate if increased diversity software improves population evolvability in software systems as it does in biological systems [53].

4.1.2 Program Atavism using Version Control

Biological systems retain genetic information encoding previous phenotypes in such a way that the previous phenotype may be accessible through very minor genetic mutations in a process called *atavism* [9]. I will develop an automated method of program atavism using information stored in version control repositories. Version control information will be encoded into an evolvable representation in such a way that it is not expressed but is easily accessible to GP operators.

Many authors have added memory to EC systems. Their work can be divided into those with implicitly and explicit memory [3]. Implicit memory systems use constructs similar to gene diploidy in which two or more alternatives of a portion of the genome (a gene) are stored and genetic operators may switch which version of the gene is active (dominant). Explicit memory systems maintain a database of whole individuals from previous runs which may be periodically injected back into the population.

The information stored in version control repositories has clear translations to both implicit and explicit memory systems. Individual patches define alternative implementations for specific portions of a program. A version control history may be views as a set of patches, the entirety of which can be stored in a sufficiently large implicit memory program representation. Each particular version in a version control history specifies a whole variant implementation. These implementations can be stored in an explicit memory system and used to seed future populations.

This work will initially focus on implicit memory representations which provide support for *non-coding* genetic material. Although both implicit and explicit memory systems have only shown clear benefits in dynamic fitness environments [3, 19], such an atavistic representation should yield a number of benefits particular to the process of bug repair with operators limited to manipulating the genetic material present in the original program:

- Unless bugs are uniformly distributed through the source code, there will exist locations in the code that are prone to buggy behavior. During regular software maintenance such *buggy* areas are more likely to accrue edits as bug fixes are applied to the code base. I hypothesize that these localized areas with denser edit histories will benefit bug fixing in two ways. First, an increased number of historical alternative implementations will be available for buggy portions of the program. Second, any representation-uniform genetic operators will focus on buggy portions of the code, which will be over-represented due to denser edit histories.
- Through *switching* on and off large sections of non-coding genetic information single mutation operations will result in large jumps through phenotype space. I hypothesize that these jumps will facilitate the exploratory processes of evolution.
- Expanding the genetic fodder available to the mutation operators expands GP's ability to express new variants.

As proposed in the previous section, I will evaluate the performance of an atavistic program representation through comparison to existing software representations. This will be done by testing their respective abilities to evolve software variants for the purposes of increasing diversity, repairing software defects, and enabling software optimization.

These experiments will be limited to programs which have sizable edit histories and, in the case of program repair, have bugs not fixable through operations on existing program representations. The benchmark suite used by Le Goues et al. [18] satisfies both of these requirements and will be used in our experimental evaluation.

4.2 Software Optimization

During compilation and linking, non-functional properties of software such as running time and executable size may be optimized. Techniques for such optimizations have been extensively researched and implementations can be found in many of the cornerstones of modern software development environments such as GCC [48].

Current techniques rely almost exclusively on operations that can be formally proven to preserve program semantics. Using the much looser test-suite based definition of program behavior described above I hope to evolve neutral program variants not reachable through semantic-preserving operations alone. Some of these semantically neutral variants may have desirable characteristics such as faster running times or lower energy consumption. Multi-objective EC may be used to evolve software variants which are semantically neutral yet optimize such non-functional properties.

Such an evolutionary multi-objective optimization technique has been demonstrated in work by Sitthi-Amorn et al. [47] in which simplified variants of pixel shaders were evolved from an extant original program. I propose an extension of this work to optimization of diverse properties of programs aside from pixel shaders.

Modern system emulators [34] allow fine-grained monitoring of many aspects of program execution, which may be difficult to predict a-priori, such as energy consumption and communication overhead. I propose to use such a system to evaluate fitness in a multi-objective EC system for software optimization. I hypothesize a number of benefits to such a system:

- Test-suite defined correctness permits more radical program transformations than available to traditional optimization strategies which are limited to formally semantic-preserving transformations.
- Evaluation using a full system emulator enables optimization of software properties not readily predictable through static analysis, such as aspects of performance based on particulars of the hardware including cache sizes or on chip network speeds.
- The use of a multi-objective fitness function provides a natural method for developers to specify priorities for non-functional optimization.

Specifically this investigation will attempt to optimize the parallel fast Fourier transform (FFT) algorithm. This algorithm is of great importance to scientific computation [39] and it has received much attention and manual optimization [12, 17]. Thus, a competitive evolutionarily optimized FFT implementation would be a significant achievement.

4.3 Software Husbandry

The evolution of diverse populations from single programs raises questions of software identity. Earlier proposed work (Section 3.3.2) described the creation of a benchmark suite composed of multiple distinct programs which all conform to the same test suite. Neutral populations evolved from the members of such a benchmark suite would populate different regions of the same test-suite defined neutral space.

It is possible that individuals from these separate regions may be successfully combined to form new hybrid implementations containing genetic material from both ancestor programs. This amounts to asking if the neutral populations derived from the two original programs are members of the same *species* — abusing the biological definition of the term.

Recombination of programs with no shared ancestry has been observed in the wild in Microsoft word macro viruses [2]. The intentional recombination of ancestrally related programs has been performed at the object level [16]. In this work Foster and Somayaji were able to link libraries from two different versions of a program to generate a new version exhibiting features of both ancestor versions.

I propose to use the single-test-suite benchmark suite developed in Section 3.3.2 to develop related neutral populations. I will then attempt to combine individuals between these neutral populations using existing crossover operations. If effective, I hypothesize that this technique will have a number of practical applications including:

- the transfer of optimizations between distinct software products
- the transfer of functionality between distinct software products
- an automated method of introducing diversity into program populations

Though outside of the scope of this proposal, such a technique may necessitate new legal tools and definitions related to issues of software identity and copyright, and new limits on the use of compiled program binaries.

5 Work-plan and Timeline

A detailed work plan is show as a gantt chart in Figure 4.

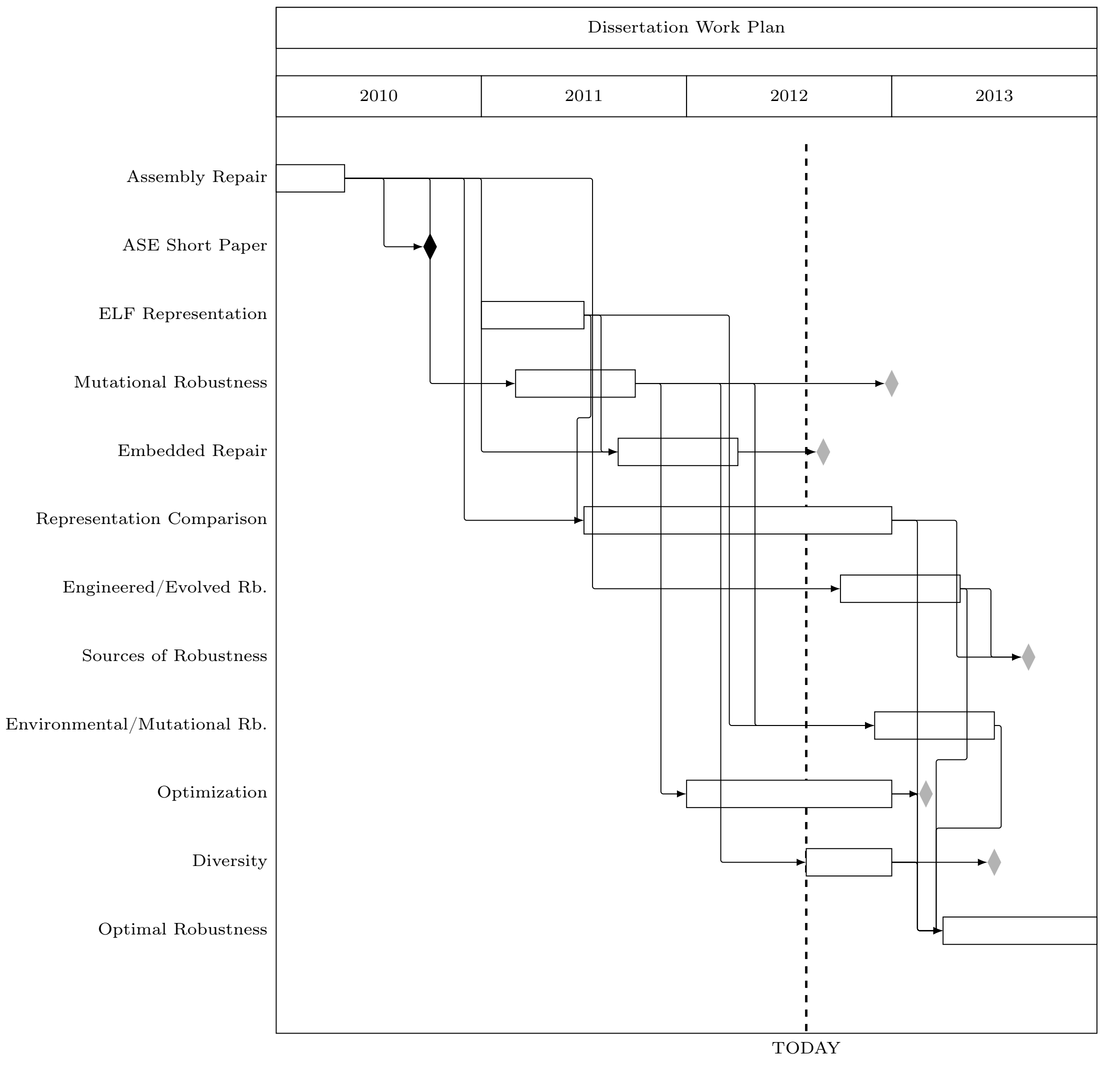


Figure 4: Publications are shown as diamonds with dark diamonds representing published papers and gray diamonds representing prospective publications.

6 Conclusion

Engineered software systems are robust in ways previously thought limited to biological systems. I propose an investigation of the robustness of software systems and a number of tools which leverage this robustness.

Our investigation will attempt to isolate the sources of robustness in software, determine the effects of robustness on the evolvability of software, and seek correlations between software robustness and other desirable software properties.

This work presents the modern software development environment as a product of natural selection. A greater understanding of the evolved properties of software will correct errors in existing folk wisdom of software fragility and will improve the ability of software engineers to reason about software performance and to build more effective tools for software development and maintenance.

References

[1] J. Beal and Gerald Jay Sussman. Engineered robustness by controlled hallucination. November 2008.

[2] V. Bontchev. Macro virus identification problems. *Computers & Security*, 17(1):69–89, 1998.

[3] J. Branke. Memory enhanced evolutionary algorithms for changing optimization problems. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3. IEEE, 1999.

[4] T.A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[5] U.S Department of Labor Bureau of Labor Statistics. Computer software engineers and computer programmers. On the internet, November 2011. http://www.bls.gov/oco/ocos303.htm.

[6] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, 1978.

[7] S. Ciliberti, O.C. Martin, and Andreas Wagner. Innovation and robustness in complex regulatory gene networks. *Proceedings of the National Academy of Sciences*, 104(34):13591, 2007.

[8] S. Ciliberti, O.C. Martin, and Andreas Wagner. Robustness can evolve gradually in complex regulatory gene networks with varying topology. *PLoS Computational Biology*, 3(2):e15, 2007.

[9] Anton Crombach and Paulien Hogeweg. Evolution of evolvability in gene regulatory networks. *PLoS Comput Biol*, 4(7):e1000112, 07 2008.

[10] R.A. DeMillo, E.W. Krauser, and A.P. Mathur. Compiler-integrated program mutation. In *Computer Software and Applications Conference, 1991. COMPSAC'91., Proceedings of the Fifteenth Annual International*, pages 351–356. IEEE, 1991.

[11] D. Deutscher, I. Meilijson, M. Kupiec, and E. Ruppin. Multiple knockout analysis of genetic robustness in the yeast metabolic network. *Nature genetics*, 38(9):993–998, 2006.

[12] P. Duhamel and M. Vetterli. Fast fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19(4):259–299, 1990.

[13] Gerald M. Edelman and J.A. Gally. Degeneracy and complexity in biological systems. *Proceedings of the National Academy of Sciences*, 98(24):13763, 2001.

[14] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[15] M.A. Félix and Andreas Wagner. Robustness and evolution: concepts, insights and challenges from a developmental model system. *Heredity*, 100(2):132–140, 2006.

[16] Blair Foster and Anil Somayaji. Object-level recombination of commodity applications. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 957–964. ACM, 2010.

[17] M. Frigo and S.G. Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[18] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repairs: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering, 2012. ICSE 2012.* IEEE, 2011.

[19] B. Hadad and C. Eick. Supporting polyploidy in genetic algorithms using dominance vectors. In *Evolutionary Programming VI*, pages 223–234. Springer, 1997.

[20] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and Martin Rinard. Power-aware computing with dynamic knobs. Technical report, Technical Report TR-2010-027, CSAIL, MIT, 2010.

[21] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* The MIT press, 1992.

[22] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[23] M.A. Huynen, P.F. Stadler, and W. Fontana. Smoothness within ruggedness: The role of neutrality in adaptation. *Proceedings of the National Academy of Sciences*, 93(1):397, 1996.

[24] Dimitris Iliopoulos, Christoph Adami, and Peter Szor. Darwin inside the machines: malware evolution and the consequences for computer security. *Virus Bulletin*, pages 187–194, 2008.

[25] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, (99):1–1, 2010.

[26] H. Kitano. Biological robustness. *Nature Reviews Genetics*, 5(11):826–837, 2004.

[27] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Software Eng.*, 12(1), 1986.

[28] R.D. Knight, S.J. Freeland, and L.F. Landweber. Selection, history and chemistry: the three faces of the genetic code. *Trends in biochemical sciences*, 24(6):241–247, 1999.

[29] J.R. Koza. Genetic programming: On the programming of computers by means of natural selection, 1992. *See http://miriad. Iip6. fr/microbes Modeling Adaptive Multi-Agent Systems Inspired by Developmental Biology*, 229, 1992.

[30] F. Kühling, K. Wolff, and P. Nordin. A brute-force approac to automatic induction of machine code on cisc architectures. *Genetic Programming*, pages 288–297, 2002.

[31] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 2011/2012.

[32] Richard E. Lenski, Jeffrey Barrick, and Charles Ofria. Balancing robustness and evolvability. *PLoS biology*, 4(12):e428, 2006.

[33] Lauren Ancel Meyers, Fredric D Ancel, and Michael Lachmann. Evolution of genetic potential. *PLoS Comput Biol*, 1(3):e32, 08 2005.

[34] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[35] S. Misailovic, D.M. Roy, and Martin Rinard. Probabilistically accurate program transformations. *Static Analysis*, pages 316–333, 2011.

[36] Charles Ofria, Christoph Adami, and Travis C. Collier. Design of evolvable computer languages. *Evolutionary Computation, IEEE Transactions on*, 6(4):420–424, 2002.

[37] Charles Ofria, W. Huang, and E. Torng. On the gradual evolution of complexity and the sudden emergence of complex features. *Artificial life*, 14(3):255–263, 2008.

[38] Charles Ofria and Claus O. Wilke. Avida: A software platform for research in computational evolutionary biology. *Artificial Life*, 10(2):191–229, 2004.

[39] A.V. Oppenheim, R.W. Schafer, J.R. Buck, et al. *Discrete-time signal processing*, volume 2. Prentice hall Upper Saddle Riverˆ eN. JNJ, 1989.

[40] J.H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. 2009.

[41] R. Poli, W.B. Langdon, and N.F. McPhee. *A field guide to genetic programming.* Lulu Enterprises Uk Ltd, 2008.

[42] A. Radul and Gerald Jay Sussman. The art of the propagator. Technical report, Technical Report MIT-CSAIL-TR-2009-002, MIT Computer Science and Artificial Intelligence Laboratory, 2009.

[43] Martin Rinard. Survival strategies for synthesized hardware systems. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE'09. 7th IEEE/ACM International Conference on*, pages 116–120. IEEE, 2009.

[44] Martin Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6*, pages 21–21. USENIX Association, 2004.

[45] Eric Schulte, Zachary P. Fry, Ethan Fast, Stephanie Forrest, and Westley Weimer. Software mutational robustness: Bridging the gap between mutation testing and evolutionary biology. http://arxiv.org/abs/1204.4224, April 2012.

[46] P. Schuster, W. Fontana, P.F. Stadler, and I.L. Hofacker. From sequences to shapes and back: A case study in rna secondary structures. *Proceedings: Biological Sciences*, pages 279–284, 1994.

[47] Pitchaya Sitthi-Amorn, Nicholas Modly, Westly Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics (TOG)*, 30(6):152, 2011.

[48] R. Stallman and Mass.) Free Software Foundation (Cambridge. *Using GCC: the GNU compiler collection reference manual.* Free Software Foundation, 2003.

[49] R.H. Untch, A.J. Offutt, and M.J. Harrold. Mutation analysis using mutant schemata. In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 139–148. ACM, 1993.

[50] E. Van Nimwegen, James Crutchfield, and M.A. Huynen. Neutral evolution of mutational robustness. *Proceedings of the National Academy of Sciences*, 96(17):9716, 1999.

[51] Andreas Wagner. Robustness and evolvability in living systems. 2005.

[52] Andreas Wagner. Neutralism and selectionism: a network-based reconciliation. *Nature Reviews Genetics*, 9(12):965–974, 2008.

[53] Andreas Wagner. Robustness and evolvability: a paradox resolved. *Proceedings of the Royal Society B: Biological Sciences*, 275(1630):91, 2008.

[54] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[55] D.R. White, A. Arcuri, and J. Clark. Evolutionary improvement of programs. *IEEE Trans. on Evolutionary Computation, PP (99)*, pages 1–24, 2011.

[56] Claus O. Wilke, Jia Lan Wang, Charles Ofria, Richard E. Lenski, and Christoph Adami. Evolution of digital organisms at high mutation rates leads to survival of the flattest. *Nature*, 412(19):331–333, July 2001.