Software Qual J (2013) 21:421–443
DOI 10.1007/s11219-013-9208-0

Current challenges in automatic software repair

Claire Le Goues · Stephanie Forrest · Westley Weimer

Published online: 7 June 2013

Abstract The abundance of defects in existing software systems is unsustainable. Addressing them is a dominant cost of software maintenance, which in turn dominates the life cycle cost of a system. Recent research has made significant progress on the problem of automatic program repair, using techniques such as evolutionary computation, instrumentation and run-time monitoring, and sound synthesis with respect to a specification. This article serves three purposes. First, we review current work on evolutionary computation approaches, focusing on GenProg, which uses genetic programming to evolve a patch to a particular bug. We summarize algorithmic improvements and recent experimental results. Second, we review related work in the rapidly growing subfield of automatic program repair. Finally, we outline important open research challenges that we believe should guide future research in the area.

Keywords Automatic program repair · Software engineering · Evolutionary computation

1 Introduction

Program evolution and repair are major components of software maintenance, which consumes a daunting fraction of the total cost of software production (Seacord et al. 2003). Although there are many tools available to help with bug triage (e.g., Anvik et al. 2006), localization (e.g., Jones and Harrold 2005; Saha et al. 2011), validation (e.g., Yin et al. 2011), and even

C. Le Goues · W. Weimer (✉)
University of Virginia, Charlottesville, VA 22904, USA
e-mail: weimer@cs.virginia.edu

C. Le Goues
e-mail: legoues@cs.virginia.edu

S. Forrest
University of New Mexico, Albuquerque, NM 87131, USA
e-mail: forrest@cs.unm.edu

confirmation (e.g., Liblit et al. 2005), generating repairs remains a predominantly manual, and thus expensive, process. The trend is clear: There is a pressing need for automatic techniques to supplement manual software development with inexpensive tools.

Research in *automated program repair* has focused on reducing repair costs by enabling continued program execution in the face of run-time errors [e.g., Juzi (Elkarablieh and Khurshid 2008), ClearView (Perkins et al. 2009), ARMOR (Carzaniga et al. 2013), or Demsky et al. (2006)], using code contracts or formal specifications to synthesize repairs [e.g., AutoFix-E (Wei et al. 2010), Axis (Liu and Zhang 2012), AFix (Jin et al. 2011), SemFix (Nguyen et al. 2013), or Gopinath et al. (2011)], or using evolutionary computation (EC) (e.g., by coevolving test cases and repairs (Arcuri 2011; Wilkerson et al. 2012), or via language-specific operators and representations (Orlov and Sipper 2011; Schulte et al. 2010). In this latter category, we introduced *GenProg* (Le Goues et al. 2012a, b, c), which uses genetic programming (GP) to repair a wide range of defect types in legacy software (e.g., infinite loops, buffer overruns, segfaults, integer overflows, format string vulnerabilities, and general incorrect output) without requiring *a priori* knowledge or formal specifications.

The breadth and depth of recent activity in this area is exciting. Automatic repair work has been evaluated by DARPA red teams (Perkins et al. 2009) and won awards for human-competitive results produced by genetic and evolutionary computation (Koza 2009). Harman (2010) sums up the challenge succinctly: "If finding bugs is technically demanding and yet economically vital, how much more difficult yet valuable would it be to automatically fix bugs?"

This article provides a high-level overview of the state of current research and existing challenges in automatic program repair, making several contributions. We begin with an update on the GenProg tool (in Sect. 2) and provide an overview and summary of recent experimental results (in Sect. 3) In conjunction with an overview of related work (Sect. 4), we use our experience to motivate a discussion of open research problems (Sect. 5), which we outline as challenges to the field. We conclude in Sect. 6.

2 GenProg

Over the past several years, we have described and evaluated several versions of *GenProg* (Le Goues et al. 2012a, b, c) an automated method that uses *genetic programming (GP)* (Koza 1922; Forrest 1993) to search for a source-level patch[1] that causes an off-the-shelf program to avoid a known defect while retaining key functionality. GP is a search technique based on the principles of biological evolution. As applied to program repair, GP maintains and evolves populations of program patches, seeking a patch that repairs the buggy behavior. In this section, we describe the current state of the algorithm, summarizing previously published work, and highlighting recent improvements that enable GenProg to efficiently repair real bugs in large real-world programs.

2.1 Illustrative example

For the purposes of clarifying insights underlying our approach, we begin by presenting a running example adapted from a publicly available vulnerability report. Consider the

[1] GenProg can also effect repairs in assembly code, binary files, and (recently) the LLVM intermediate representation.

pseudocode shown in Fig. 1a, adapted from a remote-exploitable heap buffer overflow vulnerability in the `nullhttpd v0.5.0` webserver. Function `ProcessRequest` processes an incoming request based on data copied from the request header. Note that on line 13, the call to `calloc` trusts the content-length provided by a POST request, copied from the header on line 7. A malicious attacker can provide a negative value for `Content-Length` and a malicious payload in the request body to overflow the heap and kill or remotely gain control of the running server (Nullhttpd 2002).

This buffer overflow vulnerability can be repaired fairly simply by adding a check on the content-length before using it in the call to `calloc`. This candidate patch to the original program is shown in Fig. 1b.

At a high level, the goal of GenProg is to get from Figs. 1a to 2b automatically. In subsequent sections, we periodically refer back to this example to illustrate the algorithmic presentation and its underlying design insights. GenProg can also successfully repair the defect on which this example is based (Le Goues et al. 2012a).

2.2 Overview

High-level pseudocode for GenProg's main GP loop is shown in Fig. 2. GenProg takes as input a program and a set of test cases that encode the bug (referred to as negative test cases) as well as required behavior that cannot be changed (the positive test cases). GenProg uses the test cases to localize the fault and compute context-sensitive information to guide the repair search (Sect. 2.6). Each individual, or variant, is represented as a *patch*, or a sequence of edit operations to the original program (Sect. 2.3). The goal is to produce a patch that causes the original program to pass all test cases.

Line 1 of Fig. 2 initializes the population by creating a number of random initial patches. Lines 2–6 correspond to one iteration, or *generation*, of the algorithm. On line 3, *tournament selection* (Miller and Goldberg 1996) selects high-fitness (Sect. 2.4)

```
1  char* ProcessRequest() {
2   ...
3   while(l=sgets(l,sock)) {
4    if(l=="Request:")
5     strcpy(req_type,l+12)
6    if(l=="Content-Length:")
7     len=atoi(l+16);
8   }
9   if(req_type=="GET")
10   buff=DoGETReq(sock,len);
11  if(req_type=="POST") {
12   sz=sizeof(char);
13   buff=calloc(len,sz);
14   rc=recv(sock,buff,len)
15   buff[len]='\0';
16  }
17  return buff;
18 }
```

(a) Webserver code snippet.

```
3  ...
4    if(l=="Request:")
5     strcpy(req_type,l+12)
6    if(l=="Content-Length:")
7     len=atoi(l+16);
8   }
9   if(req_type=="GET")
10   buff=DoGETReq(sock,len);
11  if(req_type=="POST") {
12 +  if (len <= 0)
13 +   return null;
14   sz=sizeof(char);
15   buff=calloc(len,sz);
16   rc=recv(sock,buff,len)
17   buff[len]='\0';
18  }
19  return buff;
20 }
```

(b) Patched webserver.

Fig. 1 Pseudocode of a webserver that contains a bug (**a**) and a repaired version of the same program (**b**)

Input: Full fitness predicate FullFitness : $Patch \to \mathbb{B}$
Input: Sampled fitness SampleFit : $Patch \to \mathbb{R}$
Input: Mutation operator mutate : $Patch \to Patch$
Input: Crossover operator crossover : $Patch^2 \to Patch^2$
Input: Parameter $PopSize$
Output: Patch that passes FullFitness
1: **let** $Pop \leftarrow$ map mutate over $PopSize$ copies of $\langle\ \rangle$
2: **repeat**
3: **let** $parents \leftarrow$ tournSelect$(Pop, Popsize,$ SampleFit$)$
4: **let** $offspr \leftarrow$ map crossover over $parents$, pairwise
5: $Pop \leftarrow$ map mutate over $parents \cup offspr$
6: **until** $\exists\ candidate \in Pop.$ FullFitness$(candidate)$
7: **return** $candidate$

Fig. 2 High-level pseudocode for the main GenProg loop; figure adapted from (Le Goues et al. 2012b). Typically, multiple *trials* (instances of the main repair loop) will be run in parallel for a given bug, with each trial initialized with a different random seed. Each trial is run either until a patch is found (*line 6*) or until a resource limit is reached (e.g., a certain number of iterations of the loop). The resource limit is externally checked and does not appear in the pseudocode

individuals to be *parents* in the next generation. In general, fitness evaluation dominates GenProg run-time. The "parents" are selected pairwise at random to undergo crossover, in which a single point is chosen randomly, and the subsequences up to the point in each parent are swapped, creating two new "offspring" variants. Each parent and each offspring are mutated once (mutate), and the result forms the incoming population for the next generation (Sect. 2.5). The GP loop terminates if a patch is found that causes the input program to pass all of its test cases, or when resources are exhausted (i.e., a predetermined time limit is exceeded). If GenProg succeeds in producing a repair, the resulting patch can be minimized in a deterministic post-processing step that combines tree-structured differencing (Al-Ekram et al. 2005) with delta debugging (Zeller 1999). Multiple executions of the algorithm are typically run in parallel for a given bug, each with a distinct random seed. Each execution, or run, is referred to as a *trial*.

In theory, the "best known" patch could be returned if no repair is found within the given time limit. Previous work has shown that developers address bug reports associated with a candidate patch more quickly than when no suggested patch accompanies the bug report, even if the proposed patch is incorrect (Weimer 2006). A "partial solution," or the best known patch found at a certain point, might serve as a useful guide for developers faced with repairing a bug by hand when the automated process fails. In an alternative use case, human advice or input might be solicited by the automated process when it is struggling, perhaps based on the "best known" patch to date. We have not yet investigated these scenarios in detail, but we speculate that they might provide alternative use cases to improve fault localization and debugging processes in practice.

2.3 Representation

The current version of GenProg represents each variant as a *patch*, or a sequence of edit operations with respect to the input program. In earlier work, and in keeping with a considerable proportion of the GP literature, GenProg represented an individual by its entire abstract syntax tree (AST), combined with a novel weighted execution path (Weimer et al. 2009). We subsequently found that the full AST representation limits scalability. For

example, for at least 36 of the 105 defects in our largest dataset of real, historical defects in open-source programs (Le Goues et al. 2012b), a population of 40–80 full ASTs did not fit in 1.7 GB of main memory. Over the same dataset, however, half of patches were 25 lines or less. Thus, two unrelated variants are likely to differ by at most 2×25 lines, with all other AST nodes in common. As a result, we now favor representing individuals as patches to avoid storing redundant copies of untouched code (Le Goues et al. 2012b). This design choice allows each individual in the population to be stored more compactly, and it scales sublinearly with the size of the code to which GenProg is being applied, a clear efficiency advantage.

Returning to our running example, one random individual in the population might correspond to "Delete the statement on line 10." We index statements by assigning them unique integer values when the program is initially parsed, and thus, the candidate patch can be represented as "Delete(N)," where N is a unique identifying integer. This consumes much less storage than an entire secondary copy of the code, with the code from line 10, `buff=DoGETReq(sock,len);`, replaced by an empty block. To evaluate each candidate, the edits are applied to the input program in order to produce a new AST, whose fitness is measured as described in the next subsection.

2.4 Fitness

The *fitness* function guides a GP search. The fitness of an individual in a program repair task should assess how well the patch causes the program to avoid the bug while still retaining all other required functionality. We use test cases to measure fitness by applying a candidate patch to the original program and then rerunning the test suite on the result.

We typically take these test suites from the regression tests associated with many open-source projects. Regardless of its provenance, the input test suite should contain at least one case that initially fails, encoding the bug under repair, as well as at least one (but typically several) that initially pass, encoding required functionality that should be maintained post-patch.

For the running example, we write a test case that demonstrates the bug by sending a POST request with a negative content-length and a malicious payload to the Web server in order to try to crash it, and then, we check whether the Web server is still running. Unmodified `nullhttpd` fails this test case.

However, defining desired program behavior exclusively by what we want `nullhttpd` to *not* do may lead to undesirable results. Consider the following variant of `nullhttpd`, created by a patch that replaces the body of the function with `return null`:

```
char* ProcessRequest() { return null; }
```

This version of `ProcessRequest` does not crash on the bug-encoding test case, but it also fails to process any requests at all. The repaired program should pass the error-encoding test case, but it must also retain core functionality before it can be considered acceptable. Such functionality can also be expressed with test cases, such as a regression test case that obtains `index.html` and compares the retrieved copy against the expected output.[2]

Running test cases typically dominates GenProg's run-time, so we use several strategies to reduce the time to evaluate candidate patches. First, test cases can often be evaluated in parallel (Weimer et al. 2009). Second, our problem, like many GP problems, is tolerant of noisy fitness functions (Fast et al. 2010), which allows us to evaluate candidates on

[2] In practice, we use several test cases to express program requirements. We describe only one here for brevity.

subsamples of the test suite. The function SampleFit evaluates a candidate patch on a random sample of the positive tests and on all the negative test cases. For efficiency, only variants that maximize SampleFit are fully tested on the entire test suite (using FullFitness). The final fitness of a variant is the weighted sum of the number of tests it passes, where negative tests are typically weighted more heavily than the positive ones. This biases the search toward patches that repair the defect (Le Goues et al. 2012c). Programs that do not compile are assigned fitness zero.

We have experimented with several test suite sampling strategies and found that a random approach works well: The benefits gained by more precise sampling are outweighed by the additional computation time to select the samples. Sampling introduces *noise* into fitness evaluation, in that the value produced by SampleFit may differ from the value produced by FullFitness for the same individual. Too much noise could lead to more fitness evaluations over the course of the search. Although we have not been able to characterize the amount of noise `SampleFit` introduces across bug or benchmark type, our experiments show that it can vary from around 15 % to as high as 70 %. Overall, we observe the additional cost of increased sampling (more fitness evaluations required to find a successful repair) is strongly outweighed by the much smaller cost per evaluation achieved through sampling.

2.5 Mutation and crossover

Mutation operates on AST nodes corresponding to C statements (e.g., excluding expressions or declarations), which limits the size of the search space. In each mutation, a destination statement d is chosen from the set of permitted statements according to a probability distribution (Sect. 2.6). In GenProg, there are three distinct types of mutation, and the algorithm chooses randomly which one to apply. We have experimented with different mutation operators, but recent versions of GenProg use **delete, insert**, or **replace**. If **insert** or **replace** is selected, a second statement s is selected randomly from elsewhere in the same program. Statement d is then either replaced with s or with a new statement consisting of d followed by an inserted s. These changes are appended to the variant's current list of edits.

Crossover selects two variants and exchanges subsequences between the two list of edits. The motivation for this operator is that valid partial solutions might be discovered by different variants, and crossover can combine them efficiently, helping to avoid local optima in the search. GenProg currently uses one-point crossover (Rowe and McPhree 2001) as follows: Given parent individuals p and q, crossover selects crossover points p_n and q_m. The first portion of p is appended to the second portion of q, and vice versa, creating two offspring, both of which are evaluated by `SampleFit`.

2.6 Search space

Because the space of all possible edits to a program is so large, GenProg restricts the search to a smaller space that is likely to contain a repair. Consider again the bug in `nullhttpd` (Fig. 1a). This code snippet represents only a small portion of the 5,575 line program. Displaying all 5,575 lines is unnecessary, however, because *not all program locations are equally likely to be good choices for changes to fix the bug*. Fault localization reduces the number of destination statements d that can be selected as locations for mutation.

Once a location for the mutation has been chosen, GenProg next selects the source statement s to be used as insertion or replacement code. We observe that *a program that*

makes a mistake in one location often handles a similar situation correctly in another (Engler et al. 2001). As a result, GenProg selects source statements *s* from code found elsewhere in the same program. This approach applies to `nullhttpd`. Although the POST request handling in `ProcessRequest` does not perform a bounds check on the user-specified content-length, the `cgi_main` function, implemented elsewhere, does:`502 if (length <= 0) return null;`

This code can be copied and inserted into the buggy region, as shown in the repaired version of the program (Fig. 1b).

The search space for program repair is therefore defined by the locations that can be changed, the mutations that can be applied at each location, and the statements that can serve as sources of the repair. We parameterize these components of the search along two key dimensions:

Fault space GenProg mutates only statements that are associated with incorrect behavior, and the statements are weighted to influence mutation probability. The input program is instrumented and executed to identify which statements are executed by which test cases. We initially believed that mutation should be biased heavily toward statements visited exclusively by the negative test cases (Weimer et al. 2009). However, we subsequently found that this intuition does not hold on our largest dataset: A uniform weighting, or one in which statements executed by both positive and negative test cases are weighted more heavily, was found to be preferable (Le Goues et al. 2012c), although we do not consider this issue completely resolved.

Fix space We use the term *fix localization* (or *fix space*) to refer to the *source* of insertion or replacement code. Candidate fixes are restricted to those within the original program, and they are currently restricted to statements visited by at least one test case (because we hypothesize that common behavior is more likely to be correct). In addition, GenProg rules out insertions that include variables that would be out of scope at the destination (to avoid type checking errors). Such localization improves search efficiency because it greatly reduces the proportion of generated variants that do not compile (Orlov and Sipper 2009).

3 Evaluation

We have evaluated GenProg along several dimensions. We established *generality* by showing that GenProg can repair many different types of bugs in real-world programs, and we demonstrated *scalability* by showing that GenProg can repair programs containing millions of lines of code, without requiring special coding practices or annotations. We have characterized and improved the algorithm in both of these dimensions. Performing these evaluations has highlighted the challenge of developing benchmarks for automated defect repair, a problem we have approached from multiple angles. In this section, we summarize several of our recent evaluation efforts, focusing on high-level goals, results, and challenges. We elide some details in the interest of space and direct the reader to associated publications where relevant.

3.1 Benchmarks

One of the greatest challenges we have faced in evaluating GenProg has been finding a good set of benchmark bugs and programs. Good benchmarks are critical to high-quality empirical science: “Since benchmarks drive computer science research and industry product development, which ones we use and how we evaluate them are key questions for the community”

Table 1 A set of benchmark programs used in experiments to evaluate GenProg's generality, with size of the program measured in lines of code (LOC)

	LOC	Description	Fault
`gcd`	22	Example	Infinite loop
`zune`	28	Example (BBC News 2008)	Infinite loop†
`uniq utx`	1146	Duplicate text processing	Segmentation fault
`look utx`	1169	Dictionary lookup	Segmentation fault
`look svr`	1363	Dictionary lookup	Infinite loop
`units svr`	1504	Metric conversion	Segmentation fault
`deroff utx`	2236	Document processing	Segmentation fault
`nullhttpd`	5575	Web server	Remote heap buffer overflow (code)†
`openldap`	292598	Directory protocol	Non-overflow denial of service†
`ccrypt`	7515	Encryption utility	Segmentation fault†
`indent`	9906	Source code processing	Infinite loop
`lighttpd`	51895	Web server	Remote heap buffer overflow (vars)†
`flex`	18775	Lexical analyzer generator	Segmentation fault
`atris`	21553	Graphical tetris game	Local stack buffer exploit†
`pphp`	764489	Scripting language	Integer overflow†
`wu-ftpd`	67029	FTP server	Format string vulnerability†
total	1246803		

The dataset contains bugs spanning 8 different fault types. See http://genprog.cs.virginia.edu/ for all benchmarks and source code used in our evaluations. Table adapted from (Le Goues et al. 2012a)

† Indicates an openly available exploit

(Blackburn et al. 2006). A good benchmark defect set should be *indicative* and *generalizable*, and it should therefore be drawn from a variety of programs representative of real-world systems. The defects should illustrate real bugs that human developers would consider important and be easy to reproduce. Existing benchmark suites such as SPEC or Siemens (Hutchins et al. 1994) do not fulfill these requirements. The SPEC programs were designed for performance benchmarking and do not contain intentional semantic defects that are required for the automated repair problem. The Siemens suite does provide programs with test suites and faults. However, it was designed for controlled testing of software testing techniques, and therefore, the test suites maximize statement coverage, the faults are almost exclusively seeded, and the programs are fairly small.

A number of studies of automatic bug finding, localization, and fixing techniques have used bugs "in the wild," found through case studies, careful search through bug databases, industrial partnerships, and word-of-mouth (e.g., Liblit et al. 2005; Perkins et al. 2009). We have also taken this approach, identifying as broad a range of defects in as many different types of programs as possible to substantiate our claim that GenProg is *general* (Table 1) for the benchmarks used in many of our studies). The programs total 1.25M lines of C code, and the bugs in the dataset cover 8 different fault types; a number are taken from public vulnerability reports (indicated with a † in the table).

To enable large-scale evaluation of GenProg's scalability and real-world utility, we recently developed a larger benchmark defect set, leveraging source control, and regression tests suites of open-source C programs in a systematic way (Le Goues et al. 2012b). Given a set of popular programs from open-source repositories, we searched systematically through each program's source history, looking for revisions that caused the program to

Table 2 A benchmark set of subject C programs, test suites, and historical defects, designed to allow large-scale, indicative, and systematic evaluation of automatic program repair techniques

Program	Description	LOC	Tests	Bugs
`fbc`	Legacy compiler for Basic	97,000	773	3
`gmp`	Precision math library	145,000	146	2
`gzip`	Data compression utility	491,000	12	5
`libtiff`	Image manipulation library	77,000	78	24
`lighttpd`	Lightweight web server	62,000	295	9
`php`	Web programming language interpreter	1,046,000	8,471	44
`python`	General programming language interpreter	407,000	355	11
`wireshark`	Network packet analyzer	2,814,000	63	7
Total		5,139,000	10,193	105

Tests were taken from the most recent version available in May 2011; defects are defined as test case failures that were repaired by developers in previous versions. See http://genprog.cs.virginia.edu/ for all benchmarks and source code used in the evaluations, including virtual machine images and pre-packaged bug scenarios that can be used to reproduce these defects. Table adapted from (Le Goues et al. 2012b)

pass test cases that failed in a previous revision. Such a scenario corresponds to a human-written repair for the bug defined by the failing test case. Table 2 summarizes the programs and defects in this dataset, which allows, to the best of our knowledge, the largest evaluation of automatic program repair to date. In total, it comprises 8 open-source C programs and 105 defects, with at least 2 defects per program.

We used this larger set to evaluate GenProg's real-world utility (Le Goues et al. 2012b) (i.e., what proportion of real bugs can be repaired automatically and the cost of a repair on publicly available cloud compute resources) and to conduct in-depth studies of critical algorithmic choices (Le Goues et al. 2012c). These latter studies allow us to ask questions about the nature of the search, how and why it works, why it does not always work, and how we may improve it, and are ongoing.

3.2 Generality

GenProg repairs all the bugs in Table 1 in 356.5 s, on average, using relatively small sets of regression test cases (automatically or human-generated or taken from the existing test suites) on a machine with 2 GB of RAM and a 2.4 GHz dual-core CPU. These bugs cover a variety of defect types, including one of the earliest reported format string vulnerabilities (`wu-ftpd`). Of the sixteen patches, seven insert code, seven delete code, and two both insert and delete code. We note that patches that delete code do not necessarily degrade functionality, because the code may have been included erroneously, or the patch may compensate for the deletion with another insertion. Similarly, it is also possible to insert code without negatively affecting functionality, because the inserted code can be guarded so it applies only to relevant inputs (i.e., zero-valued arguments or tricky leap years).

Although a comprehensive code review is beyond the scope of this article, manual inspection (and quantitative evaluation, results not shown Le Goues et al. 2012a) suggests that the patches are acceptable, in that they appear to address the underlying defect without introducing new vectors of attack. In our experiments and experience with patches that GenProg has produced, we observe that lost functionality in response to inadequate positive test cases appears more likely than the introduction of new vulnerabilities. Overall, GenProg patches are typically highly localized in their effects.

Using commodity cloud resources, and limiting all repair runs to a maximum of 12 h (simulating an overnight repair run), GenProg repaired approximately half of the bugs in Table 2, including at least one per program. Recall that this dataset was intended to evaluate real-world cost and expressive power. Modifying certain parameter values or changing various selection probabilities in the algorithm can influence GenProg's ability to find a repair, especially for the more "difficult" repair scenarios (that is, those on which GenProg's random success rate is lower on average). For example, altering the probability distribution used to select the mutation type, changing the crossover algorithm, and changing the fault and fix space weightings allowed GenProg to repair 5 new bugs when compared against a default baseline (Le Goues et al. 2012c). Similarly, running the repair algorithm for longer (von Laszewski et al. 2010) causes GenProg to repair at least another 6 of the 105 scenarios, when compared to the 12-h scenario.

We have investigated several of the important parameter, operator, and representation choices (see especially Le Goues et al. 2012c), including two representations and four versions of the crossover operator. We also investigated the mutation operators and their selection probability as well as fault and fix space modifications and probability distributions. These investigations leave open the possibility of additional parameter sweeps in future work. Our results suggest additional avenues of future inquiry. For example, the patch representation (Sect. 2.3) appears to be more effective than the original abstract syntax tree/weighted path representation (Le Goues et al. 2012c), but the mechanism behind this remains unknown. While the two representation choices encode the same types of changes, we hypothesize that differences in the way are applied to the AST result in slightly different search space traversals in each case. Regardless, the GP algorithm that we use in GenProg is quite different from that typically used in many GP applications, a fact that motivates careful consideration of the various operator and parameter choices that underlie its implementation.

Overall, we view the successful and efficient repair of at least 8 different defect types in 16 programs and half of 105 systematically identified defects from programs totaling 5.1 million lines of code as a strong result, indicating the potential for generic automated repair algorithms such as GenProg.

3.3 Scalability and success

GenProg has repaired bugs in large programs with large test suites, as shown both in Tables 1 and 2. We have found that the time to execute test cases dominates repair time (comprising 64 % of the time to repair the benchmarks in Table 1, for example), which motivated our efforts to find ways to reduce the time necessary for fitness evaluation (Fast et al. 2010).

We are still working to characterize the conditions that influence GenProg's success rate and time to repair. However, we have investigated a number of potential relationships. We consistently find a weak but statistically significant power law relationship between fault localization size and both time to repair and probability of success (Le Goues et al. 2012a, b; Forrest et al. 2009). As fault space size increases, the probability of repair success decreases, and the number of fitness evaluations required to find a repair (an algorithmic measure of search time) increases. We have also found a negative correlation between the fix space size and repair time. We speculate that larger fix spaces include more candidate repair options, thus reducing the time to find any given one.

We have also analyzed the relationship between repair success and external metrics such as human repair time and size, and defect severity. The only significant correlation we have identified using such metrics is between the number of files touched by a human-generated

patch and repair success: The more files the humans changed to address the defect, the less likely GenProg was to find a repair. We have found no significant correlation between "bug report severity" and "GenProg's ability to repair," which we consider encouraging.

3.4 Example patch

In this subsection, we describe one patch produced by GenProg for a `php` bug from Table 2 and compare it to the one produced by humans for the same defect. We adapted this description from Le Goues et al. (2012b) for the purposes of illustrating the types of patches that GenProg can produce.

The `php` interpreter uses reference counting to determine when dynamic objects should be freed. User programs written in `php` may overload internal accessor functions to specify behavior when undefined class fields are accessed. Version 5.2.17 of `php` had a bug related to a combination of these features. At a high level, the "read property" function, which handles accessors, always calls a deep reference count decrement on one of its arguments, potentially freeing both that reference and the memory it points to. This is the correct behavior *unless* that argument points to `$this` when `$this` references a global variable—a situation that arises if the user program overrides the internal accessor to return `$this`. In such circumstances, the global variable has its reference count decremented to zero and its memory is mistakenly freed while it is still globally reachable.

The human-written patch replaces a line that always calls the deep decrement with a simple if-then-else: in the normal case (i.e., the argument is not a class object), calling the deep decrement as before, otherwise calling a separate shallow decrement function. The shallow decrement function will free the pointer, but not the object to which it points.

The GenProg patch adapts code from a nearby "unset property" function. The deep decrement is unchanged, but additional code is inserted to check for the abnormal case. In the abnormal case, the reference count is deeply incremented (through machinations involving a new variable), and then, the same shallow decrement is called.

Thus, at a very high level, the human patch changes the call to `deep_Decr()` to:

```
1 if (normal) deep_Decr(); else shallow_Decr();
```

while the GP-generated patch changes it to:

```
1 deep_Decr();
2 if (abnormal) { deep_Incr(); shallow_Decr(); }
```

The logical effect is the same but the command ordering is not, and both patches are of comparable length. The human patch is perhaps more natural: It avoids the deep decrement rather than performing it and then undoing it.

4 Related work

Automatic program repair and related problems have received considerable attention in recent years, including work on debugging and debugging assistance; error preemption, recovery, and repair; and evolutionary search, GP, and search-based software engineering.

Debugging Work on debugging and debugging assistance focuses on identifying defects or narrowing the cause of a defect to a small number of lines of code. Recent debugging

advances include replay debugging (Albertsson and Magnusson 2000), cooperative statistical bug isolation (Liblit et al. 2003), and statically debugging fully specified programs (He and Gupta 2004). Other techniques mine program history and related artifacts to suggest bug repairs or otherwise provide debugging support (Jeffrey et al. 2009; Ashok et al. 2009). Trace localization (Ball et al. 2003), minimization (Groce and Kroening 2005), and explanation (Chaki et al. 2004) projects also aim to elucidate faults in the context of static defect detection.

Such work is best viewed as complementary to automated repair: A defect found or localized automatically could also be explained and repaired automatically. However, a common underlying assumption of such work is that unannotated programs must be repaired manually, albeit with additional information or flexibility presented to the developer. We propose several ways that these approaches might be extended or improved to improve automated repair and related challenges, in Sect. 5.1

Automated error preemption and defect repair One class of approaches to automatic error handling uses source code instrumentation and run-time monitoring to prevent harmful effects from certain types of errors. Programs with monitoring instrumentation can detect data structure inconsistency or memory over- or under-flows. Various strategies are then used to enable continued execution (Elkarablieh and Khurshid 2008; Demsky et al. 2006), generate trace logs, attack signatures and candidate patches for the system administrators (Smirnov and Chiueh 2005; Smirnov et al. 2006), or dispatch to custom error handlers (Sidiroglou et al. 2005; Sidiroglou and Keromytis 2005). Jolt (Carbin et al. 2011) assists in the dynamic detection of and recovery from infinite loops.

Other research efforts (including our own) focus directly on patch generation. At the binary level, ClearView (Perkins et al. 2009) uses monitors and instrumentation to flag erroneous executions and generate candidate patches. ARMOR (Carzaniga et al. 2013) replaces library calls with equivalent statements, using multiple implementations to support recovery from erroneous run-time behavior. AutoFix-E (Wei et al. 2010) uses contracts present in Eiffel code to propose semantically sound fixes. SemFix uses symbolic execution and program constraints to build repairs from relevant variables (Nguyen et al. 2013). Gopinath et al. also use formal specifications to transform buggy programs (Gopinath et al. 2011). Axis (Liu and Zhang 2012) and AFix (Jin et al. 2011) soundly patch single-variable atomicity violations, while Bradbury et al. propose to use GP to address concurrency errors (Bradbury and Jalbert (2010), PACHIKA (Dallmeier et al. 2009) infers object behavior models to propose candidate fixes.

Many of these techniques are designed for particular types of defects, making use of pre-enumerated repair strategies or templates. Buffer overruns are particularly well handled in the previous work, but overall generality remains a dominant research concern. Additionally, concurrency bugs remain a significant challenge. AFix and Axis are two of the only techniques to address them explicitly, although several of the other techniques can repair deterministic bugs in multi-threaded programs. However, non-deterministic bugs remain very difficult to test, and addressing that challenge is largely independent of integrating any solution into an automated repair framework that depends on testing. Some techniques, such as AutoFix-E, require specifications or annotations. While this enables semantic soundness guarantees, formal specifications are rare in practice (Palshikar 2001). We discuss several potentially fruitful research directions suggested by the current state of the art in program repair in the next section.

One major challenge in comparing these techniques for expressive power, generality, or generality utility is the relative dearth of agreed upon benchmark defects and experimental frameworks. Many researchers, ourselves included, identify bugs on which to test by

combining through bug database histories, borrowing from other previous work, or following word-of-mouth. It can be difficult to reproduce the datasets from other work for direct comparison. We have begun to propose a process for identifying candidate benchmark defects (as described in Sect. 3.1 and Le Goues et al. 2012b), but community consensus would go a long way toward enabling comprehensive comparative studies of both previously and newly proposed techniques.

SBSE and evolutionary computation Search-Based Software Engineering (SBSE) (Harman 2007) traditionally uses evolutionary and related methods for software testing, e.g., to develop test suites (Wappler and Wegener 1925; Michael et al. 2001). These techniques typically focus on automatically generating high-coverage test suites (Jia and Harman 2010), often with multiple competing search objectives (e.g., high-coverage tests, optimizing for a smaller test suite Lakhotia et al. 2007). SBSE also uses evolutionary methods to improve software project management and effort estimation (Barreto et al. 2008), find safety violations (Alba and Chicano 2007), and in some cases re-factor or re-engineer large software bases (Seng et al. 1909).

Arcuri and Yao (2008) proposed to use GP to coevolve defect repairs and unit test cases; the idea has since been significantly expanded (Arcuri 2011; Wilkerson et al. 2012). These approaches use competitive coevolution: The test cases evolve to find more bugs in the program, and the program evolves in response (Adamopoulos et al. 2004). Techniques along these lines tend to rely at least in part on formal program specifications to define program correctness, and thus the fitness function (Arcuri and Yao 2008; Wilkerson and Tauritz 2011). More work remains to increase the scalability, usability, and applicability of specification and verification-based approaches. We discuss potential extensions along these lines in the next section.

There has been considerable recent interest in and independent validations of the potential of GP (Bradbury and Jalbert (2010) or random mutation (Debroy and Wong 2010) for program repair. It has been applied to new languages (Orlov and Sipper 2011; Schulte et al. 2010; Ackling et al. 2011) and domains (Sitthi-Amorn et al. 2011; Langdon and Harman 2010), and has improved non-functional program properties, particularly execution time (White et al. 2011).

5 Open research challenges

The research results summarized in the previous sections, both our own and those of others, suggest that the prospects for automated software repair are promising. Transforming research results into practicality, however, raises important challenges, which we believe should guide future work. In this section, we summarize the near-term challenges that we believe are most important to automated program repair. At a high level, these challenges fall into two broad categories:

- *Real-world practicality* How can we transform automatic software repair research into widely used real-world software maintenance techniques? Challenges include scalability (how quickly they find repairs, how many lines of code they can handle) and generality (what sorts of programs and bugs they can address). An additional challenge is establishing the credibility of automated repairs in terms of programmers' confidence in them and their understandability.
- *Theory of unsound repair methods* How and why do current unsound repair techniques work? The success of existing approaches, particularly those that are unsound or

stochastic (e.g., ClearView Perkins et al. 2009 or GenProg Le Goues et al. 2012b), has been demonstrated empirically, but we lack a theoretical underpinning. The empirical results raise questions about the nature of extant software, its robustness in the face of random modifications, and how robustness and evolvability can be leveraged and enhanced.

The following subsections outline a set of specific challenges, each of which falls into one of these two general categories.

5.1 Adapting fault localization for automated repair applications

There exists considerable research on the problem of *fault localization* which automatically identifies (potentially) faulty code regions. In the context of automatic repair, these techniques identify likely locations for making a code modification, i.e., where the code is likely broken. We have consistently observed that well-localized bugs are more likely to be repaired and take less time to repair than poorly localized bugs. As a result, there is a concern that data-only bugs such as SQL injection vulnerabilities will be less suitable candidates for automated repair. Because the fault localization methods we have used to date are quite simple, there is more sophisticated methods (e.g., Chen et al. 2002; Abreu et al. 2006) could be incorporated to allow automated program repair of data-only bugs.

Contrary to our initial intuition, statements that are executed exclusively by the bug-inducing input may not be the best locations for a repair. This suggests the need to revisit our ideas about fault localization, because GenProg and related techniques may not benefit from increasingly precise identification of buggy statements. Rather, we need fault localization techniques that identify good candidates for code change to affect the buggy execution in a positive way, without breaking non-buggy execution. This reconception of fault localization is a subtle but potentially important shift from its traditional purposes. It may be necessary to modify, extend, or reimagine existing fault localization techniques, designed for human consumption to help developers identify regions of code associated with a bug, to this new task of identifying regions of code that are good locations to change for repairing a particular bug.

Other extensions of the repair technique may require entirely novel innovations in fault localization. For example, Schulte et al. extended the GenProg approach to repair programs at the assembly level (Schulte et al. 2013). They proposed a novel stochastic, sampled fault localization technique to smoothly identify good candidate mutation locations along the assembly-level execution path. Applying the algorithm at other, different levels of abstraction (at the component or software architectural level, for example) will almost certainly demand similar types of innovation.

5.2 Automatically finding or generating code that is likely to repair software defects

Beyond the challenge of identifying good repair locations, it is also desirable to understand, formalize, and automatically predict how best to make a repair, addressing the *fix localization* problem. Progress in this area could increase the applicability and scalability of automated repair techniques, and it might improve our understanding of bug repair in general.

For example, GenProg currently restricts inserted code to statements that appear elsewhere in the same program. New programming paradigms or APIs, or novel bugs or vulnerability types, could stymie this paradigm. Considerable research attention has been

devoted to mining information from source code or repositories (Lanza et al. 2012); this work could suggest augmented mutation operators including repair templates, even for novel bugs. We envision a technique that suggest candidate repairs by mining other related projects or modules, perhaps adapting methods from specification mining (Robillard et al. 2012) or guided by software quality metrics (Buse and Weimer 2008; McCabe 1976). Mined templates could also potentially introduce expression- or declaration-level modifications in a domain intelligent way, without necessarily expanding the size of the search space prohibitively. These types of modifications may enable the repair of more complex defects than GenProg, and related techniques have addressed to date.

Even within the same project, identifying particularly promising code to serve as sources for automated repair remains a largely open problem. We took an initial step in this direction by restricting the genetic operators, such that they do not move variables out of scope. Going beyond this general approach will likely involve special knowledge about the nature of both the bug and the program under repair. For example, in the security domain, specialized transformations have been proposed to repair or preclude buffer overruns (Smirnov et al. 2006; Sidiroglou et al. 2005; Barrantes et al. 2003). If GenProg had reason to believe that the bug in question is a buffer overrun (because of the results of a pre-processing static analysis step, for example), it could apply such transformations with higher probability than it might otherwise.

The fix localization challenge affects all automated repair techniques, not just EC- or GP-based stochastic search methods, including those that use pre-specified templates and those that make semantically guaranteed mutations. Which repair template is best for a given defect type? This challenge is motivating other researchers in the area, and we expect it will continue to do so (Kim et al. 2013). By accurately refining the search space for repairs and identifying more semantically expressive fixes, fix localization could enable more efficient searches for repairs and the repair of more complex bugs.

5.3 Formalizing patch quality and maintainability

Patch quality is an important impediment to the practical adoption of automated repair techniques. Human developers, who may never fully "leave the loop," must be confident that a given patch correctly repairs a defect without violating other system requirements. This problem arises in all techniques that do not provide soundness guarantees. In the absence of fully automated programming, it is desirable for repair techniques, whether sound or unsound, to produce patches that humans can understand and later maintain. Formal models of repair quality would allow tools to present only the best repairs to humans or provide confidence estimates of the repair's quality. They could provide useful insight about patches from any source, both human- and tool-generated.

Quantitative measures of quality could include existing techniques such as held-out test suites and black-box fuzzing or new techniques which have not yet been discovered. We anecdotally observe that GenProg is more likely to reduce program functionality in the face of inadequate test cases than it is to introduce a new malicious and exploitable vulnerability. Additional research in measuring, predicting, or ensuring functional patch quality might profitably begin by focusing on generating high-coverage tests to validate the behavior impacted by a patch, rather than on designing new operators to mitigate the probability that new vulnerabilities are introduced.

A formal model of quality, with related metrics, could also be used to analyze and improve existing techniques with repair quality in mind. Such a model would likely require human studies to understand and quantify what factors affect programmer understanding of

a patch. Many programmers appear to have an instinct about whether a not a patch is "good." The research challenge is in formalizing, quantifying, and ultimately predicting this judgment. We have previously published results that studied the impact of patches on certain measures of maintainability (Fry et al. 2012). In a recent human study, Kim et al. (Kim et al. 2013) evaluated machine-generated patches from GenProg and their PAR tool in terms of "acceptability," and found that their patches are comparable to human-written patches. Much more work remains to be done in this general area, both to develop additional measures, study other repair methods, and improve existing repair search techniques. We believe that a full model that predicts, measures, and controls for the effect of a patch on higher-level system concerns will likely incorporate both syntactic and deeper semantic features of a patch.

Extending such an approach so it can *predict* the impact of a given change may allow automated repair techniques to optimize for both functional and non-functional patch properties. Tools like GenProg and ClearView can often generate multiple patches for the same bug, and a notion of patch quality could be used to select from multiple options, or as an additional parameter in the fitness function.

Patch explanation or documentation techniques (Buse and Weimer 2010) could increase developer confidence in the output of automated tools, improving usability and adoption. Such work could also apply to human-generated patches. While existing change documentation techniques produces what may be referred to as "what" explanations, or explanations of the externally visible behavior the patch impacts and under what conditions, it may be possible to develop more helpful descriptions of what a patch is doing and why. For example, dataflow or slicing analyses may be able to describe the input and output variables affected by a proposed patch or provide an execution "frontier" beyond which a patch will no longer have an effect. Alternatively, a patch could come with an automatically generated "justification" for each change that explains what happens (which test cases fail, for example) if it is removed.

5.4 Automatic generation of full test cases: inputs and *oracles*

Repair techniques such as GenProg use testing to guide a search or measure acceptability. Test suites, found much more commonly in practice than formal specifications, serve as a proxy for complete specifications and are used by many program analysis techniques (e.g., Jones and Harrold 2005). However, despite their prevalence in industrial practice, few programs are comprehensively tested, constraining the applicability of automated repair techniques that depend on test suites.

Although automated test case generation is a popular topic, most techniques intended for large programs produce test inputs, not full test cases (e.g., Godefroid et al. 2005; Sen 2007). These methods typically generate successive inputs that help maximize code coverage, for example, but are not full test cases in the sense of the oracle-comparator model (Binder 1999). Unless the program crashes, it is often difficult to determine whether the program has "passed" when presented with the input. There is promising recent work in this arena (Fraser and Zeller 2012; Fraser and Zeller 2011), but we think there is much more to be done, both for automated testing and automatic program repair.

Another natural way to lift the test case assumption is to integrate GenProg with existing bug-finding tools, particularly those that identify faulty inputs or execution paths associated with buggy behavior (Cadar et al. 2006). Such approaches could mitigate the need for humans to identify the failing test cases that serve as observers for the bug in question. Reconstructing inputs or even execution traces from static bug-finding tools is a

known challenge (Jhala and Majumdar 2005). The problem of creating high-quality test oracles applies here as well. Previous work has suggested that program-specific temporal safety specifications are more useful for identifying bugs than library- or API-level specifications (such as "lock should always eventually be followed by unlock") (Weimer and Necula 2005). This suggests that advances in specification mining for extant software may prove useful in this domain (Robillard et al. 2012; Le and Weimer 2012).

5.5 Combining unsound repair algorithms and formal methods

Today, few production programs come with a complete formal specification, and most commercial software is validated through test suites. Formal methods, however, have made enormous advances in the past decade, and there are several intriguing possibilities for integrating the two approaches. In GenProg, for example, the fitness function is sensitive only to whether or not a test case was passed, a binary signal. Measuring correctness only through test cases is a crude and possibly misleading approach. For example, consider a bug that requires the addition of the synchronization primitives `lock` and `unlock` around the use of a shared variable. A candidate patch that inserts only a `lock` will almost certainly lead to code that performs worse than inserting nothing at all, since it will likely lead to deadlock.

It is therefore desirable to have a finer-grained fitness function that can report how close a program is to passing a test case, especially the negative test cases. This might be accomplished by incorporating program analysis information, such as learned invariants (Ernst et al. 2007; Nguyen et al. 2012; Gabel and Su 2012), into the fitness function. In other work, we performed an initial study of such an approach (Fast et al. 2010). We found that, on a particular case study, it was possible to use behavior as measured by such invariants to construct a fitness function that is more accurate than test cases alone (Jones and Forrest 1995). However, the model we constructed was fairly opaque, and in subsequent efforts, we were unable to generalize it to different bugs or programs. This is not to say that such a general model of intermediate patch correctness as profiled by dynamic invariants is impossible. Rather, using observed behavior on dynamic program predicates to characterize intermediate program behavior en route to a repair shows promise as an approach, but it will require additional insight before it can achieve general utility.

Other possibilities for integrating sound program analysis techniques into unsound approaches for program repair include automatically generating proofs for repairs in the context of proof carrying code (Necula 1997), or automatically mutating code in a semantics-preserving way with a goal of simplifying automatic verification (because equisatisfiable verification conditions are not always equally easy to discharge) (Yin et al. 2009). These potential applications could lead to higher-quality repairs as well as repairs that are easier for humans to validate and trust.

5.6 Robustness and evolvability as first-order concerns

GenProg and other evolutionary search techniques modify code using guided random mutations. Techniques such as ClearView use template-based repair systems that modify the code to avoid faulty behavior, without necessarily knowing *a priori* what the specified correct behavior should be (beyond universal "don't crash" policies). Given that these techniques often succeed, in the sense that the software passes all tests (Le Goues et al. 2012b), is just as maintainable by humans (Fry et al. 2012), and defeats red teams (Perkins et al. 2009), they raise important questions about software robustness in the face of

external perturbations (random or not). In this context, software is considered *robust* if it continues to function correctly in the face of perturbations to its environment, operating conditions, or program code. In our own experience, software appears surprisingly robust to random statement-level changes, at both the abstract syntax tree and the assembly language levels, regardless of the coverage of its test suite (Schulte et al. 2012). Why is it that seemingly random mutations can improve software with respect to a set of test cases?

The success these unsound techniques challenges common definitions of *acceptability*: Continued execution of an adapted or evolved system may often be a better outcome than complete failure. This is the general approach taken by Rinard et al. in their failure-oblivious computing models, where programs are dynamically modified to continue executing in the face of a defect such as a buffer overrun or infinite loop (Perkins et al. 2009; Carbin et al. 2011; Rinard et al. 2004). Additional exploration of such ideas is warranted, both to characterize why current repair techniques can succeed at all and to gain better insight into the nature of software.

These observations apply to program repair as well as to software systems in general. Software today is deployed in highly dynamic environments. Nearly, every aspect of a computational system is likely to change during its normal life cycle (Le Goues et al. 2010). New users interact with software in novel ways, often finding new uses for old code; the specification of what the system is supposed to do changes over time; the owner and maintainers of the system are replaced by new developers; the system software and libraries on which the computation depends are upgraded or replaced; and the hardware on which the system is deployed is continuously modified.

A formal notion of software change and evolvability would open new avenues for understanding, prescribing, and predicting software behavior. It also suggests extensions to more traditional lines of software engineering research. One example might be new methods for developing and maintaining evolving test suites that continue to encode correct behavior for a given program as it evolves naturally throughout the software engineering process.

5.7 Fully automated software development

Beyond the goal of automated software repair lies the challenge of fully automating the entire software development process, including synthesizing complex programs from scratch using unsound techniques such as genetic programming. GenProg starts with a program that is almost correct and has a single identified defect. What would it take for automated repair to start with a partially debugged program and iteratively improve the program until all the defects were eliminated, perhaps in the style of extreme programming (Beck 2000)? Or more ambitiously, what technical impediments are there to using a technique like GenProg to synthesize full-length programs, perhaps beginning with a small set of functional test cases and then coevolving the test suite with the program? We do not know the answers to these questions, and we consider this a long-term goal for automated software engineering rather than a near-term challenge. We note, however, that maintaining large systems often consists of iteratively debugging small sections of code with a goal of ever-increasing program functionality. It is not beyond the realm of possibility to imagine a future GenProg, or another unsound method, developing software from scratch using a "programming as iterated debugging" development paradigm.

6 Conclusions

Software quality is an expensive problem, and the need for automated techniques for defect repair is pressing. Existing work in the area is promising, receiving attention from DARPA red teams, and other external measures of success. In this article, we have described the current state of *GenProg*, a technique that uses genetic programming to repair unannotated legacy programs. It does so by evolving a set of changes that causes a given buggy program to pass a given set of test cases. In recent evaluations, we have established that GenProg is expressive and scalable, evaluated against a varied set of bugs as well as a large, systematically generated set of defects.

As a result of our initial experience with automated program repair, we have identified a number of design decisions that are critical to scalability and repair success, e.g., evolving patches rather than programs, focusing the search on commonly visited code areas. In addition, we have identified a number of challenges that we believe lie at the heart of future progress in automated program repair: locating possible fixes, evaluating the quality of repairs, operating without full test suites or formal specifications, understanding change, and ultimately taking new steps toward automated software development.

We have released the source code for GenProg as well as our benchmark programs and defects at http://genprog.cs.virginia.edu. We hope other researchers focusing on bugs in legacy software will find utility in a relatively large set of objects to study, and that encourage others to download, extend, and compare against our tool. Overall, we hope that this work will motivate researchers to tackle this important and promising area, which is still in its infancy.

References

Abreu, R., Zoeteweij, P., & van Gemund, A. J. C. (2006). An evaluation of similarity coefficients for software fault localization. In Pacific rim international symposium on dependable computing. IEEE Computer Society, 39–46.

Ackling, T., Alexander, B., & Grunert, I. (2011). Evolving patches for software repair. In Genetic and evolutionary computation, 1427–1434.

Adamopoulos, K., Harman, M., & Hierons, R. M. (2004). How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In Genetic and evolutionary computation conference, 1338–1349.

Alba, E., & Chicano, F. (2007). Finding safety errors with ACO. In Genetic and evolutionary computation conference, 1066–1073.

Albertsson, L., & Magnusson, P. S. (2000). Using complete system simulation for temporal debugging of general purpose operating systems and workload. In International symposium on modeling, analysis and simulation of computer and telecommunication systems, 191.

Al-Ekram, R., Adma, A., & Baysal, O. (2005). diffX: An algorithm to detect changes in multi-version XML documents. In Conference of the centre for advanced studies on collaborative research. IBM Press, 1–11.

Anvik, J., Hiew, L., & Murphy, G. C. (2006). Who should fix this bug? In International conference on software engineering, 361–370.

Arcuri, A. (2011). Evolutionary repair of faulty software. *Applied Soft Computing, 11*(4), 3494–3514.

Arcuri, A., & Yao, X. (2008). A novel co-evolutionary approach to automatic software bug fixing. In Congress on evolutionary computation, 162–168.

Ashok, B., Joy, J., Liang, H., Rajamani, S. K., Srinivasa, G., & Vangala, V. (2009) DebugAdvisor: A recommender system for debugging. In Foundations of software engineering, 373–382.

Ball, T., Naik, M., & Rajamani, S. K. (2003). From symptom to cause: Localizing errors in counterexample traces. *SIGPLAN Notices, 38*(1), 97–105.

Barrantes, E. G., Ackley, D. H., Palmer, T. S., Stefanovic, D., & Zovi, D. D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. In Computer and communications security, 281–289.

Barreto, A., de Barros, O. M., & Werner, C. M. (2008). Staffing a software project: A constraint satisfaction and optimization-based approach. *Computers and Operations Research, 35*(10), 3073–3089.
BBC News. (2008). Microsoft Zune affected by 'bug'. http://news.bbc.co.uk/2/hi/technology/7806683.stm.
Beck, K. (2000). *Extreme programming explained: Embrace change*. Reading: Addison-Wesley.
Binder, R. V. (1999). *Testing object-oriented systems: Models, patterns, and tools*. Reading: Addison-Wesley Longman Publishing Co., Inc.
Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., et al. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. In Object-oriented programming, systems, languages, and applications, 169–190.
Bradbury, J. S., & Jalbert, K. (2010). Automatic repair of concurrency bugs. In: International symposium on search based software engineering—fast abstracts, 1–2.
Buse, R. P. L., & Weimer, W. (2008). A metric for software readability. In International symposium on software testing and analysis, 121–130.
Buse, R. P. L., & Weimer, W. (2010). Automatically documenting program changes. In Automated software engineering, 33–42.
Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., & Engler, D. R. (2006) EXE: Automatically generating inputs of death. In Computer and communications security, 322–335.
Carbin, M., Misailovic, S., Kling, M., & Rinard, M. C. (2011) Detecting and escaping infinite loops with Jolt. In European conference on object oriented programming.
Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., Pezzè, M. (2013). Automatic recovery from runtime failures. In International conference on sofware engineering.
Chaki, S., Groce, A., & Strichman, O. (2004). Explaining abstract counterexamples. In Foundations of software engineering, 73–82.
Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., & Brewer, E. (2002). Pinpoint: Problem determination in large, dynamic Internet services. In International conference on dependable systems and networks, 595–604.
Dallmeier, V., Zeller, A., & Meyer, B. (2009). Generating fixes from object behavior anomalies. In Automated software engineering, 550–554.
Debroy, V., & Wong, W. E. (2010). Using mutation to automatically suggest fixes for faulty programs. In International conference on software testing, verification, and validation, 65–74.
Demsky, B., Ernst, M. D., Guo, P. J., McCamant, S., Perkins, J. H., & Rinard, M. C. (2006) Inference and enforcement of data structure consistency specifications. In International symposium on software testing and analysis.
Elkarablieh, B., & Khurshid, S. (2008). Juzi: A tool for repairing complex data structures. In International conference on software engineering, 855–858.
Engler, D. R., Chen, D. Y., & Chou, A. (2001). Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In Symposium on operating systems principles.
Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., & Xiao, C. (2007). The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming, 69*(1–3), 35–45.
Fast, E., Le Goues, C., Forrest, S., & Weimer, W. (2010). Designing better fitness functions for automated program repair. In Genetic and evolutionary computation conference, 965–972.
Forrest, S. (1993). Genetic algorithms: Principles of natural selection applied to computation. *Science, 261*, 872–878.
Forrest, S., Weimer, W., Nguyen, T., & Le Goues, C. (2009). A genetic programming approach to automated software repair. In Genetic and evolutionary computation conference, 947–954.
Fraser, G., & Zeller, A. (2012). Mutation-driven generation of unit tests and oracles. *Transactions on Software Engineering, 38*(2), 278–292.
Fraser, G., & Zeller, A. (2011). Generating parameterized unit tests. In International symposium on software testing and analysis, 364–374.
Fry, Z. P., Landau, B., & Weimer, W. (2012). A human study of patch maintainability. In M. P. E. Heimdahl, & Su, Z., (Eds.), International symposium on software testing and analysis, 177–187.
Gabel, M., & Su, Z. (2012). Testing mined specifications. In Foundations of software engineering, ACM, 1–11.
Godefroid, P., Klarlund, N., & Sen, K. (2005). Dart: Directed automated random testing. In Programming language design and implementation, 213–223.
Gopinath, D., Malik, M. Z., & Khurshid, S. (2011). Specification-based program repair using sat. In Tools and algorithms for the construction and analysis of systems. Volume 6605 of lecture notes in computer science. Springer, 173–188.
Groce, A., & Kroening, D. (2005). Making the most of BMC counterexamples. *Electronic Notes in Theoretical Computer Science, 119*(2), 67–81.
Harman, M. (2010). Automated patching techniques: The fix is in (technical perspective). *Communications of the ACM, 53*(5), 108.

Harman, M. (2007). The current state and future of search based software engineering. In International conference on software engineering, 342–357.
He, H., & Gupta, N. (2004). Automated debugging using path-based weakest preconditions. In Fundamental approaches to software engineering, 267–280.
Hutchins, M., Foster, H., Goradia, T., & Ostrand, T. (1994). Experiments of the effectiveness of dataflow-and control flow-based test adequacy criteria. In International conference on software engineering 191–200.
Jeffrey, D., Feng, M., Gupta, N., & Gupta, R. (2009). BugFix: A learning-based tool to assist developers in fixing bugs. In International conference on program comprehension.
Jhala, R., & Majumdar, R. (2005). Path slicing. In Programming language design and implementation. New York, NY: ACM Press, 38–47.
Jia, Y., & Harman, M. (2010). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering, 99* (PrePrints).
Jin, G., Song, L., Zhang, W., Lu, S., & Liblit, B. (2011). Automated atomicity-violation fixing. In Programming language design and implementation.
Jones, T., & Forrest, S. (1995). Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In International conference on genetic algorithms, 184–192.
Jones, J. A., & Harrold, M. J. (2005). Empirical evaluation of the Tarantula automatic fault-localization technique. In Automated software engineering, 273–282.
Kim, D., Nam, J., Song, J., & Kim, S. (2013). Automatic patch generation learned from human-written patches. In International conference on sofware engineering.
Koza, J.R. (1922). *Genetic programming: On the programming of computers by means of natural selection.* Cambridge: MIT Press.
Koza, J. R. (2009). Awards for human-competitive results produced by genetic and evolutionary computation. http://www.genetic-programming.org/hc2009/cfe2009.html.
Lakhotia, K., Harman, M., & McMinn, P. (2007). A multi-objective approach to search-based test data generation. In Genetic and evolutionary computation conference, 1098–1105.
Langdon, W. B., & Harman, M. (2010). Evolving a CUDA kernel from an nVidia template. In Congress on evolutionary computation, 1–8.
Lanza, M., Penta, M. D., Xi, T., (Eds). (2012). IEEE working conference o mining software repositories. MSR, IEEE.
Le Goues, C., & Weimer, W. (2012). Measuring code quality to improve specification mining. *IEEE Transactions on Software Engineering, 38*(1), 175–190.
Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. (2012a). GenProg: A generic method for automated software repair. *Transactions on Software Engineering, 38*(1), 54–72.
Le Goues, C., Forrest, S., & Weimer, W. (2010). The case for software evolution. In Workshop on the future of software engineering research, 205–210.
Le Goues, C., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012b). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In International conference on software engineering, 3–13.
Le Goues, C., Forrest, S., & Weimer, W. (2012c). Representations and operators for improving evolutionary software repair. In Genetic and evolutionary computation conference, 959–966.
Liblit, B., Aiken, A., Zheng, A. X., & Jordan, M. I. (2003). Bug isolation via remote program sampling. In Programming language design and implementation, 141–154.
Liblit, B., Naik, M., Zheng, A. X., Aiken, A., & Jordan, M. I. (2005). Scalable statistical bug isolation. In Programming language design and implementation, 15–26.
Liu, P., & Zhang, C. (2012). Axis: Automatically fixing atomicity violations through solving control constraints. In: International conference on software engineering, 299–309.
McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308–320.
Michael, C. C., McGraw, G., & Schatz, M. A. (2001). Generating software test data by evolution. *IEEE Transactions on Software Engineering, 27*(12), 1085–1110.
Miller, B. L., & Goldberg, D. E. (1996). Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary Computing, 4*(2), 113–131.
Necula, G. C. (1997). Proof-carrying code. In Principles of programming languages. New York, NY: ACM, 106–119.
Nguyen, T., Kapur, D., Weimer, W., & Forrest, S. (2012) Using dynamic analysis to discover polynomial and array invariants. In International conference on software engineering, 683–693.
Nguyen, H. D. T., Qi, D., Roychoudhury, A., & Chandra, S. (2013). SemFix: Program repair via semantic analysis. In International conference on sofware engineering, 772–781.
Nullhttpd. (2002). Bug: http://nvd.nist.gov/nvd.cfm?cvename=CVE-2002-1496. Exploit: http://www.mail-archive.com/bugtraq@securityfocus.com/msg09178.html.

Orlov, M., & Sipper, M. (2011). Flight of the FINCH through the Java wilderness. *Transactions on Evolutionary Computation, 15*(2), 166–192.
Orlov, M., & Sipper, M. (2009) Genetic programming in the wild: Evolving unrestricted bytecode. In Genetic and evolutionary computation conference, 1043–1050.
Palshikar, G. (2001). Applying formal specifications to real-world software development. *IEEE Software, 18*(5), 89–97.
Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., et al. (2009). Automatically patching errors in deployed software. In Symposium on operating systems principles.
Rinard, M. C., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., & Beebee, W. S. (2004). Enhancing server availability and security through failure-oblivious computing. In Operating systems design and implementation, 303–316.
Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M., & Ratchford, T. (2012). Automated API property inference techniques. *Transactions on Software Engineering, 99* (PP).
Rowe, J. E., & McPhree, N. F. (2001) The effects of crossover and mutation operators on variable length linear structures. In Genetic and evolutionary computation conference, 535–542.
Saha, D., Nanda, M. G., Dhoolia, P., Nandivada, V. K., Sinha, V., & Chandra, S. (2011). Fault localization for data-centric programs. In Foundations of software engineering.
Schulte, E., Forrest, S., & Weimer, W. (2010). Automatic program repair through the evolution of assembly code. In Automated software engineering, 33–36.
Schulte, E., Fry, Z. P., Fast, E., Forrest, S., & Weimer, W. (2012). Software mutational robustness: Bridging the gap between mutation testing and evolutionary biology. CoRR **abs/1204.4224**.
Schulte, E., DiLorenzo, J., Forrest, S., & Weimer, W. (2013). Automated repair of binary and assembly programs for cooperating embedded devices. In Architectural support for programming languages and operating systems.
Seacord, R. C., Plakosh, D., & Lewis, G. A. (2003). *Modernizing legacy systems: software technologies, engineering process and business practices*. Reading: Addison-Wesley Longman Publishing Co. Inc.
Sen, K. (2007). Concolic testing. In Automated software engineering, 571–572.
Seng, O., Stammel, J., & Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In Genetic and evolutionary computation conference, 1909–1916.
Sidiroglou, S., & Keromytis, A. D. (2005). Countering network worms through automatic patch generation. *IEEE Security and Privacy, 3*(6), 41–49.
Sidiroglou, S., Giovanidis, G., & Keromytis, A. D. (2005). A dynamic mechanism for recovering from buffer overflow attacks. In Information security, 1–15.
Sitthi-Amorn, P., Modly, N., Weimer, W., & Lawrence, J. (2011). Genetic programming for shader simplification. *ACM Transactions on Graphics, 30*(5).
Smirnov, A., & Chiueh, T. C. (2005). Dira: Automatic detection, identification and repair of control-hijacking attacks. In Network and distributed system security symposium.
Smirnov, A., Lin, R., & Chiueh, T. C. (2006). PASAN: Automatic patch and signature generation for buffer overflow attacks. In Systems and information security, 165–170.
von Laszewski, G., Fox, G., Wang, F., Younge, A., Kulshrestha, A., Pike, G., et al. (2010). Design of the futuregrid experiment management framework. In Gateway computing environments workshop, 1–10.
Wappler, S., & Wegener, J. (2006). Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In Genetic and evolutionary computation conference, 1925–1932.
Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., & Zeller, A. (2010). Automated fixing of programs with contracts. In International symposium on software testing and analysis, 61–72.
Weimer, W. (2006). Patches as better bug reports. In Generative programming and component engineering, 181–190.
Weimer, W., & Necula, G. C. (2005). Mining temporal specifications for error detection. In Tools and algorithms for the construction and analysis of systems, 461–476.
Weimer, W., Nguyen, T., Le Goues, C., & Forrest, S. (2009). Automatically finding patches using genetic programming. In International conference on software engineering, 364–367.
White, D. R., Arcuri, A., & Clark, J. A. (2011). Evolutionary improvement of programs. *Transactions on Evolutionary Computation, 15*(4), 515–538.
Wilkerson, J. L., & Tauritz, D. R. (2011). A guide for fitness function design. In Genetic and evolutionary computation conference, 123–124.
Wilkerson, J. L., Tauritz, D. R., & Bridges, J. M. (2012). Multi-objective coevolutionary automated software correction. In Genetic and evolutionary computation conference, 1229–1236.
Yin, X., Knight, J. C., & Weimer, W. (2009). Exploiting refactoring in formal verification. In International conference on dependable systems and networks, 53–62.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., & Bairavasundaram, L. N. (2011). How do fixes become bugs? In: Foundations of software engineering, 26–36.
Zeller, A. (1999). Yesterday, my program worked. Today, it does not. Why? In Foundations of software engineering.

Author Biographies

Claire Le Goues received the BA degree in computer science from Harvard University and the MS degree from the University of Virginia, where she is currently a graduate student. Her main research interests lie in combining static and dynamic analyses to prevent, locate, and repair errors in programs.

Stephanie Forrest received the BA degree from St. John's College and the MS and PhD degrees from the University Michigan. She is currently a Professor of Computer Science at the University of New Mexico and a member of the External Faculty of the Santa Fe Institute. Her research studies complex adaptive systems, including immunology, evolutionary computation, biological modeling, and computer security.

Westley Weimer received the BA degree in computer science and mathematics from Cornell University and the MS and PhD degrees from the University of California, Berkeley. He is currently an associate professor at the University of Virginia. His main research interests include static and dynamic analyses to improve software quality and fix defects.