The Development of Automated Program Repair

Andrew Hill
North Carolina State University
ahill6@ncsu.edu

ABSTRACT

High quality software is increasingly important in a world which now depends on applications for everything from regulating utilities to transportation to human interaction. Unfortunately, the process of developing this software is sufficiently complicated that users often find new bugs for the entire life of the application. The cost of this constant cycle of bug repair is a major expenditure both of employee time and of public trust. The field of Automated Program Repair (APR) seeks to improve software quality and provide developers the means to keep pace with new bug discoveries by developing general-use automated tools to automatically repair software defects. This process can both reduce cost and ensure high software quality for more users.

This paper provides an overview of the development of the field of APR from 2009 to 2016. It traces the maturation of the field in both its main historical focus (Generate-and-Validate techniques) and newer approaches such as semantic algorithms and search-based repair, as well as analyzing the underlying concepts upon which each strategy is built and suggesting directions for future work.

Keywords

Automation, Testing, Debugging, Validation, Software Engineering, Program Repair, Semantics-Based Repair, Genetic Programming, Genetic Algorithm

1. INTRODUCTION

High quality software is increasingly important in a world which now depends on applications for everything from regulating utilities to transportation to human interaction. Unfortunately, the process of developing this software is sufficiently complicated that users often find new bugs for the entire life of the application. The cost of this constant cycle of bug repair is a major expenditure both of employee time and of public trust. The field of Automated Program Repair (APR) seeks to improve software quality and provide developers the means to keep pace with new bug discoveries by developing general-use automated tools to automatically repair software defects. This process can both reduce cost and ensure high software quality for more users.

This paper provides an overview of the development of the field of APR from 2009 to 2016. It traces the maturation of the field in both its main historical focus (Generate-and-Validate techniques) and newer approaches such as semantic algorithms and search-based repair, as well as analyzing the underlying concepts upon which each strategy is built and suggesting directions for future work.

The remainder of the paper is structured as follows: Section 2 provides background information for readers unfamiliar with the process and requirements of Automated Program Repair. Section 3 traces the development of the field from the understanding of a need and the creation of fault localization tools which allowed work on defect repair. After a review of the major methods and developments in APR, Section 4 discusses issues on which there is broad agreement and future work that is needed.

DOI: 10.1145/1235

2. BACKGROUND

All methods in Automated Program Repair have basic similarities - comparable to the way in which most Genetic Algorithms (GAs) differ only in their fitness function. Table 1 shows these basic steps. Of these steps, approaches to APR can be characterized by differences in how they Encode Desired Behavior, Generate Patches, and Rank Patches. It should be noted however that patch generation is generally a question of efficiency and almost perfectly determined by the way in which desired behavior is encoded. Differences in ranking patches are usually minor, but can be compared to secondary sorts in GA fitness functions. A familiarity with these basic steps is necessary to understand the progress of the field generally, as well as to understand the importance of the distinctions between methods which are often quite similar.

2.1 Fault Localization

In order to fix a bug, a program must first be able to find it. Fault localization methods return a list of the most likely locations for bugs. In APR, this is generally done after a test has failed, and uses the path of the failing test(s) to help localize the bug. While some authors have created their own fault localization methods due to the vital nature of this step in successful repair (e.g. [3]), most present research in APR assumes that standard tools like Tarantula or Ochiai will be able to find the fault location with acceptable accuracy (e.g. [8, 7, 24]).

Table 1: Steps in Automated Program Repair

1	Localize Fault
2	Encode Desired and Buggy Behavior
3	Create Prospective Patches
4	Rank Patches
5	Select and Apply Patch
6	Validate Patch
7	Evaluate Technique

2.2 Encoding Desired Behavior

2.2.1 Extracting Knowledge From Bug

Once the fault location is localized, how can the developer access knowledge about what is causing the program to fail? Initially, formal specifications were required to encode this behavior (e.g. [30], or else work was restricted to languages for which such formal requirements were a standard part of program construction[30]. There is still some variation in the means of attaining failure data and desired functionality, but most methods today use unit tests. This is in large part because current development, testing, and operations models mean that unit tests are widely available for a broad cross-section of application types. All current major algorithms use unit tests to encode desired and failing program behavior, as well as to evaluate the success of potential patches.

2.2.2 Problem Representation

After a fault location has been identified and faulty/desired behavior is available, this information needs to be encoded into some type of model to be evaluated or optimized.

The choice of representation is one of the clearest distinguishing features in APR methods. For example, GA-based techniques strive to encode these requirements in such a way as to stochastically optimize, usually representing source code as an Abstract Syntax Tree (AST) - a tree representation of source code for a given programming language, often used for semantic analysis in compilers. Semantic methods prefer to describe the problem in terms of a Satisfiability Modulo Theory (SMT) problem. An SMT problem is a mapping from non-binary inputs to True/False - one common example being inequalities.[1] Nearly all approaches to APR are split between these two means of representing the problem, and the representation chosen has considerable influence in how the problem is solved.

It is interesting to note that one way in which APR differs from standard Search-Based Software Engineering (SBSE) is that objective values are not usually a function of decision variables. In APR, decision variables often determine the source code to be used. Once this mapping from decision variables to source code is complete, there is an intermediate-space of source code, which is then mapped to the true objective values such as number of unit tests passing. While it is not unusual for model-based approaches to SBSE to have multiple evaluation steps in a somewhat analogous way(see [12]), this does result in certain differences in the field, which are sadly beyond the scope of this paper.

2.3 Create Prospective Patches

[1] $4x + 3y - 2z > 1$ can have any values for x,y,z; but the inequality is either true or false

The next step in the process is the generation of potential patches to correct the failing behavior. This is the most obvious way in which methods differ, but in large part the creation of prospective patches is nearly identical among methods with the same problem representation and encoding mechanisms. Patch creation for GP-type methods is accomplished through the standard means of cross-over and mutation (in some rare cases by random search or complete enumeration of the search space). Because semantic techniques tend to represent the problem as an SMT, patch creation is realized by translating the solution provided by an SMT solver into the source code space, a process known as patch synthesis.

2.4 Rank Patches

The "Create Patches" step commonly creates hundreds of patches (GA next generation or enumeration of space for semantic methods). Ranking patches is often accomplished by the Selection operator of a GA. However, more modern algorithms have achieved impressive improvements in speed by altering how patches are evaluated (e.g. trying test cases likely to fail first and skipping all remaining or evaluating patches most likely to succeed first). Patches are ranked on a variety of criteria, with the most common provided below.

2.4.1 Passing All Test Cases

That the patched program should not have the same bugs as the original program is the *sine qua non* of APR. How to describe a successfully-patched program is an area of some dispute (see Encoding Desired Behavior above), but it is standard practice to rank patches based on the number of previously failing unit tests which now pass (with some penalty factor for previously passing unit tests which now fail).

2.4.2 Minimal Patch

The early attempts at APR generated patches via GA which often had many times more changes than were required due to the evolutionary process of finding patches. Such patches nearly always led to undesirable side effects both in functional properties (unexpected new bugs because of unnecessary code) and non-functional properties (increased storage and runtime requirements). As far back as 1999, the need for minimal-change patches was understood and techniques for debugging minimally were developed.[35] These developments would merit an entire paper in themselves, but by the advent of APR these strategies were well-developed. As a result, APR often treats the existence of such techniques as axiomatic. Today, techniques for constructing minimal patches are widely known and widely accepted as necessary. The goal of minimizing changes in a patch is

well-established in the field, and nearly all APR techniques rank patches with few changes higher than larger patches with the same functionality; or else minimize changes in a post-processing step.

2.4.3 Maintainability/Readability

Because human-readability is a rather difficult concept to automate, there are many different ways by which algorithms try to achieve it. The assumption that human-readable patches are qualitatively better and improve maintainability (or likelihood of APR adoption in industry) was accepted axiomatically until recently, but even today the vast majority of the field would agree that human-similar patches are desirable (that is, patches which resemble those created by a human developer).

The evolution of thought on this issue will be covered in more detail below. While there is disagreement as to how to fit readability into a ranking system (or if it belongs instead in the Evaluate Technique step), there is general agreement that it is an important factor to be considered.

2.5 Select Patch and Apply

The means of selecting the "best" patch of those generated varies greatly from method to method. Early techniques selected at random and literally wrote code to be compiled and executed. More recently most techniques select the top-ranked patch and use some form of symbolic execution; or at minimum create a sandboxed virtual environment so that untested, stochastically-generated code is not running on a physical machine. For symbolic execution, the availability of tools such as KLEE have greatly simplified implementation.

2.6 Validate Patch

Whether using a GA, enumeration, or random search, all methods need a means of determining whether the current best patch does in fact repair the bug. Borrowing terminology from Machine Learning, let unit tests be divided into training and testing groups. The training group (or some subset of it in the case of large test suites) will be used to create the patch, with other tests held out for patch validation. A patch which passes all tests in the testing group is called a *plausible* or *validated* patch, while one which further passes all tests in the test group is a *correct* patch. One problem with current methods is that many techniques (even among the most popular methods) must generate as much as ten plausible patches to find *one* that is correct.

Furthermore, the split into training and test groups is not completely standard in the community, and due to a lack of understanding on these issues terms such as *validated* can be used to denote patches which have passed all tests in the training set, a subset of the training set, or both training and testing sets depending on author. As an example of the problem, the first paper to raise the issue of doing testing on the training group in APR was published in 2015[28] and the newest techniques are still not consistent in their testing protocols.[20] Additionally, lack of familiarity with these concepts often means that verification of patch correctness is done by manual inspection.

2.7 Evaluate Method

This paper considers the history of the field 2009-2016. Until 2015, evaluating methods was done primarily using the same measures as patch ranking. That is to say that the only standard measures used between papers were the number of patches generated (the distinction between plausible and correct patches had not yet been made explicit in the literature) and runtime of the method. Individual authors would sometimes offer other candidates, such as number of candidate patches (NCP) generated before a valid patch is found, but the standardization of metrics to evaluate methods against one another is an area of future work for the field.

3. DEVELOPMENT OF AUTOMATED PROGRAM REPAIR

3.1 Automated Program Repair Before GenProg

3.1.1 Need

The cost of finding and repairing bugs has been acknowledged as an issue since moths were found in relays, but with the increasingly critical nature of computing, the need for a rapid and cost-effective means of addressing general bugs became more urgent. Unfortunately, searching for the fault was originally a manual, time-intensive procedure. By 2003 this issue was being researched heavily, and new techniques were developed based on set, graph, and tree representations of programs.[6] As of 2005, the four dominant methods were derived from set union, set intersection, nearest-neighbor, and cause transitions; with the set methods often giving dependable results only if all unit tests for the entire suite were run - a time-consuming and impractical task for large software projects.

Furthermore, it is important to note that as late as 2008, the goal of Automated Program Repair in many minds was to evolve complete programs from scratch - so-called Automatic or Genetic Programming (GP).[4] It was not until later that same year that the suggestion of Genetic Improvement (GI) was raised.[5] This approach focused on using existing material to repair bugs in preexisting software rather than grow a whole program or method from scratch. It is likely not a coincidence that within one year the first general-purpose APR method (GenProg) was developed.[33]

3.1.2 Tarantula - 2005

In 2005, new fault-localization software called Tarantula was released which had a deceptively simple approach: the likelihood a line of code is "suspicious", given in Figure 1.[14] Suspiciousness is simply the proportion of failing tests executing that line divided by the sum of the passing and failing proportions executing that line. Tarantula is still in widespread use today, and while fault-localization is an open problem, for the purposes of APR it is commonly treated as solved due to the predominance of Tarantula as the fault localization mechanism for many major GA-based techniques (e.g. [24, 22]. Other products such as Ochiai have been found to be superior in some instances, but lack the widespread adoption of Tarantula.[1]

3.1.3 ClearView - 2009

There were some in the early days who saw APR as a sub-field of Security. This is reasonable, as the most successful real-world applications of APR techniques were used to secure networks from attack. ClearView was developed

$$suspiciousness(e) = 1 - hue(e) = \\ = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$

Figure 1: Formula for identifying suspicious lines of code in Tarantula fault localizer

in 2009 in collaboration with DARPA for the purpose of automatically identify malicious code and patching systems during operation. An early version of an Anomaly-based Intrusion Protection System, the algorithm learned invariants of normal behavior and failure behavior. It then generated patches that enforced the normal invariants and iterated to ensure continued functioning of the system. The technique was tested by a DARPA Red-Team exercise in which a security company developed 10 binaries for exploits. ClearView successfully detected all attacks, prevented any malicious code from being executed on the target systems, and automatically deployed patches which repaired the underlying vulnerability for 8 of the 10 exploits. In addition, the Red Team attempted to trick ClearView into applying extraneous or damaging patches to systems not under attack (essentially trying to induce false positives), but were unsuccessful.

Notwithstanding ClearView's very impressive successes, it was a very tailored technique which only worked against certain types of attacks (only tested against memory management, heap buffer/stack overflow, out of bounds array index, and unchecked JS type attacks), and did not extend to general programming situations. Nonetheless, it was an early and powerful example of what was possible.

3.1.4 GenProg - 2009

GenProg, the eventual standard-bearer of APR (detailed in section 3.2) was released to much fanfare in 2009[33, 9, 31], winning best paper at ICSE 2009, GECCO 2009, and SBST 2009 in addition to Gold at the Humies Awards for human-competitive results by Evolutionary or Genetic Algorithms. It used unit tests rather than full formal specifications or niche applications to encode desired behavior, and its fault localization calculation was used by nearly all Generate and Validate (G&V) type methods 2012-2014. Nonetheless, the technique at the time was considered part of a niche community that worked on toy problems, and for the next three years research was focused in other directions.

3.1.5 AutoFix-E - 2010

Due to the incompleteness of unit tests (not encoding all desired functionality or all ways for the program to fail), other avenues were explored to encode the program requirements. AutoFix-E skirted the entire debate by using the Eiffel programming language, which requires contract-based specifications for all classes.[30] Contracts are made of pre- and post-conditions, checks for the class, and class invariants. Thus classes in this language already have a much more thorough specification than is possible from mining unit tests.

With this information, AutoFix-E creates a Finite State Machine to describe correct behavior and guide patch creation. Patches which satisfy all requirements are ranked, with fewest changes being preferred. Out of 42 faults, AutoFix-E succeeded in creating valid patches for 38%, with 30% being manually verified as correct fixes by developers. Again, the results were not generalizable to more widely-used languages without contracts, and 30% is not an ideal success rate. Even so, being able to automatically repair 30% of errors in code without human interaction was a result that excited many.

3.1.6 AFix - 2011

Another example of an early success story is AFix, which was able to successfully patch 75% of bugs presented to it, with the added benefit of never introducing a patch that caused undesired behavior.[13] Unfortunately, there were several shortcomings. In addition to only being tested on 8 bugs, this technique only worked on a single subtype of bugs: single-variable atomicity violation concurrency bugs. As with previous techniques with good results, the method was not generalizable.

3.1.7 Status Quo 2011 and Changes Coming

Despite the many successes outlined above, there was feeling that progress was not being made quickly enough. In a 2011 paper, Arcuri suggests that the progress being made was limited to a few defect classes - those which can be corrected even with the severe restrictions on allowed changes placed by the above algorithms.[3] He suggests that the only way to find a solution in the incredibly large space of possible repairs is to formulate it as a search problem. While this is an important suggestion, it is also confirmation of the fractured nature of the field, as the author shows no knowledge of GenProg (released two years earlier), and for data compares random search, Simulated Annealing, and a standard GA.

3.1.8 Angelic Debugging - 2011

In order to address concerns with the size of the search space, angelic debugging was developed.[7] Angelic debugging executes the program concretely to a point near fault, then represents the faulty region as a symbolic execution tree such as shown in Figure 2. By making extensive use of symbolic execution to increase flexibility, angelic debugging checks whether there is any possible value of scoped variables which would allow all unit tests to pass. If creation of a correct patch in the current situation is impossible, it is a waste of resources to continue generating potential solutions in this branch. The authors suggest that this technique prunes the search space efficiently enough that it could be a replacement for the whole GA in GenProg.

3.1.9 Emphasis on Non-Functional MOEAs - 2011

With the difficulty of evolving correct patches for existing programs and the seeming impossibility of evolving whole programs from scratch, White, Arcuri, and Clark considered using APR methods to evolve optimizations for already-functioning code.[34] They found that they could reduce program instruction count (used as a proxy for runtime) by 50% on average, but that the results were very program-dependent (27-98% improvements depending on program). In the ASE 2012 keynote, Mark Harman recommended such approaches[11] and later called for methods which could describe the Pareto front of trade-offs between such non-functional properties, allowing developers and business users

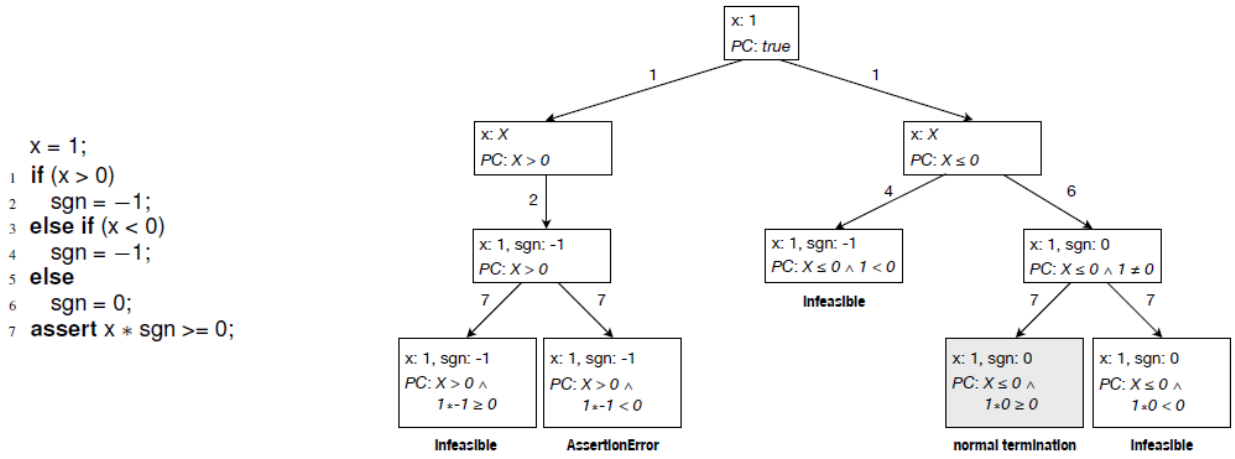

Figure 2: Example of Symbolic Execution for Simple Program

to pick not just product trade-offs, but efficiency preferences as well.[17] While these efforts have continued slowly, they would be derailed by the 2012 release of GenProg's results on real-world program repair.

3.2 GenProg - 2009, 2012

As stated above, GenProg was released to much fanfare in 2009.[33] However, it was after a 2012 update and the publication of results for its work on millions of lines of real-world C code that it became the benchmark by which all subsequent methods were based.[18] GenProg was the first completely automated method to successfully produce large numbers of patches on real-world code without any additional specifications or annotations beyond unit tests - which are already created by standard business practices in most cases. A 2012 paper by the original authors further enhanced the algorithm, allowing scalability up to 1.2M Lines of Code (LoC), improved proofs of repair quality, explicit discussion of fixes spanning eight defect classes, and an improvement of average success to 77%.[19] Finally, in response to criticism about potential bias in their dataset, the GenProg researchers constructed the GenProg Benchmark Set, a collection of bugs with good coverage of various defect classes, in order to facilitate accurate comparisons between methods. GenProg is the true beginning of APR, and of its many revolutionary innovations only the four most important will be discussed here.

3.2.1 *Representation*

GenProg was among the first methods to represent program source code as an Abstract Syntax Tree (AST), a tree representation of source code for a given programming language, often used for semantic analysis in compilers. The authors would later adapt this approach to store only patches of changes to a base AST in order to enable scalability.[19] This (together with the simplified fault localization and embrace of the Genetic Improvement approach discussed below) enabled GenProg to scale to millions of lines of code, an unprecedented success in the field.

3.2.2 *Fault Localization and Path Constraints*

Prior to GenProg, fault localization was treated as a semi-solved problem. There were many good fault localization techniques (e.g. see Tarantula above), but many researchers still created their own fault localization when implementing APR. GenProg's fault localization algorithm was so simple and effective that it drove the field to either copy GenProg (RSRepair, PAR), or use another standard library such as Tarantula (SemFix, Angelix). Note that the this breakdown is usually along party lines (with G&V methods based on GenProg and semantic techniques using Tarantula).

In order to calculate fault location, a suspiciousness value of 0 is given to any statement never visited by a failing test case, a value of 1 is given to statements visited only by failing test cases, and a value of 0.1 is given to statements visited by both passing and failing test cases. This effectively encodes human intuition that code is more likely to be the problem if every time it is used bad behavior results, without ruling out the possibility that small bugs could sometimes still yield the correct answer.

GenProg uses this elegantly simple formula to limit the locations in which mutations can take place. Rather than open up all functioning parts of the source code to random changes, GenProg changes are only made in areas which are likely to be buggy. This approach is of the type of genius that is instantly recognized, but difficult to create, and was immediately adopted by nearly all other methods.

3.2.3 *Use of Existing Code (Genetic Improvement)*

Initial attempts at Genetic Programming most often tried to evolve solutions from scratch, [4] or limited search space by design to predetermined fixes.[32, 13, 25, 30] GenProg uses the AST representation to improve the granularity with which the method can operate, but this results in explosive growth of the search space. In order to counter this tendency, the authors chose to only use code from the program itself for changes. That is, they assume that for large programs, any buggy section has similar functionality implemented correctly elsewhere. This allows them to search only existing code rather than all possible combinations of

characters. In addition, it improves code quality, because code written by humans and transplanted should resemble human-written code. This use of existing application code to improve the application (as opposed to evolving whole programs from scratch) became known as Genetic Improvement (as opposed to Genetic Programming), and popularized and expanded the ideas of [5] from the year before.

3.2.4 Minimal Patching

Once a patch has been found which passes all unit tests under consideration, it is likely that evolution has created a large amount of junk code. In order to minimize unnecessary lines of code, and also to reduce the chance of inadvertent code interactions, GenProg uses the structural differencing[2] and delta debugging[35] to obtain a minimal patch. Minimal patch here means that it retains all patch functionality, but that if any single line were removed, functionality would be lost. The concepts of structural differencing and delta debugging were already well-known in the field, but GenProg's effective use of them brought them even more widespread acceptance.

3.3 Generate and Validate Techniques Through 2014

3.3.1 TrpAutoRepair - 2013

Optimizations to the new star of APR were the source of many research projects, and the most immediate was TrpAutoRepair.[26] This method built a prioritization scheme on top of GenProg in order to improve runtimes. Qi et al. recognized that running test cases consumed the majority of resources for GenProg, and devised a scheme to run test cases that were most likely to fail first. Since the search continues until a patch is found which passes all test cases, being able to exit the unit tests early if it is discovered that this patch fails is very valuable. By further ordering the unit tests such that the first tests applied are the most likely to be failed, TrpAutoRepair significantly improves runtime while finding a comparable number of patches to GenProg. The quality of the results is not surprising, since TrpAutoRepair is simply an optimized version of GenProg.

3.3.2 AE - 2013

Interestingly enough, one of GenProg's major competitors was provided by two members of the team that created GenProg. The Adaptive Equivalence (AE) method also seeks to limit the number of evaluations needed, but seeks to do this by finding equivalence classes and uses a deterministic algorithm.[32] AE builds on the success of TrpAutoRepair in prioritizing test cases that are most likely to fail and thus saving time that otherwise would be spent evaluating candidates. The innovation of AE is that candidate repairs are ordered so that repairs most likely to pass all tests are tried first, with this calculation based on a model of observations which can be updated during execution. AE finds marginally fewer fixes (55 vs 57 in original article above), but finds them an order of magnitude faster. This is due entirely to such optimizations, since AE focuses exclusively on patches which perform only a single edit, then exhaustively enumerates them for search.

3.3.3 PAR - 2013

While AE took a step back from randomness with its exhaustive enumeration of a constrained search space, the first real attack on the inherent randomness of GAs after GenProg came from Kim et al. in the creation of PAR.[16] They pointed out that because of the randomness inherent in mutation, the patches generated by GenProg were often incomprehensible to humans, even when they do work. Suggesting that this would pose serious problems for maintainability, the authors proposed a system which would attempt to write patches that looked like human-generated patches. This was accomplished by conducting a study of how human developers repair bugs. The authors found that human fixes fell into discernible patterns, listed in Figure 3. With these human-inspired templates to guide the patches, bug fixes should more closely resemble human-generated code.

For 119 Java bugs, PAR repaired 27 to GenProg's 16. However, these were nearly disjoint sets, with only 5 overlapping. A group of 17 Software Engineering graduate students and 68 professional developers were asked to rank the quality of patches for those five bugs. The patches from PAR, GenProg, and ones written by human developers were randomized and presented to the reviewers pairwise for comparison. Patches from PAR and human-developers were statistically indistinguishable in rankings, while GenProg patches were significantly worse, suggesting that the goal of achieving human-similar patches was achieved.

Martin Monperrus immediately released a response to the publication of PAR,[23] describing his concerns with their methodology, and wider concerns that the whole field of APR should be discussing which defect classes techniques worked on rather than simply comparing numbers of bugs with a patch or (equivalently) number of test suite unit tests passed. Additionally, he pointed out the the original tests did not give enough data for statistically significant statements, did not address composition or construction of the dataset used, and challenged the assumption that human readability was a desirable trait in automatically generated patches. PAR has become a major player in the field, but this paper had a praiseworthy influence on the clarity of thought in future papers. In particular, his challenges to the assumptions of how methods were compared would prove prescient.

3.3.4 RSRepair - 2014

The last of the initial reactions to the success of GenProg was RSRepair in 2014. Qi et al. (the same researchers who created TrpAutoRepair) presented RSRepair as an update to TrpAutoRepair.[27] Like their previous work, RSRepair uses a version of GenProg with the now-standard optimizations of prioritizing test case. However, Random Search Repair (RSRepair) also replaces the GA in GenProg with random search. The experimental results show that RSRepair finds solutions competitive with GenProg, but much faster. While the authors do not give numbers for comparison, their graphics suggest that much of the time the median number of fitness function evaluations for RSRepair is at least as fast as the fastest 10% of GenProg trials. The authors suggest that their result casts doubt on the utility of the crossover and mutation operators, and suggests that all future work be baselined against random search.

Despite numerous potential threats to validity and incredibly poor organization when presenting their results, the fact that this method is at present considered essentially just another G&V method, and essentially indistinguishable from

Template Name	Description
Parameter Replacer	For a method call, this template seeks variables or expressions whose type is compatible with a method parameter within the same scope. Then, it replaces the selected parameter by a compatible variable or expression.
Method Replacer	For a method call, this template replaces it to another method with compatible parameters and return type.
Parameter Adder and Remover	For a method call, this template adds or removes parameters if the method has overloaded methods. When it adds a parameter, this template search for compatible variables and expressions in the same scope. Then, it adds one of them to the place of the new parameter.
Expression Replacer	For a conditional branch such as `if()` or ternary operator, this template replaces its predicate by another expression collected in the same scope.
Expression Adder and Remover	For a conditional branch, this template inserts or removes a term of its predicate. When adding a term, the template collects predicates from the same scope.
Null Pointer Checker	For a statement in a program, this template adds `if()` statements checking whether an object is `null` only if the statement has any object reference.
Object Initializer	For a variable in a method call, this template inserts an initialization statement before the call. The statement uses the basic constructor which has no parameter.
Range Checker	For a statement with array references, this template adds `if()` statements that check whether an array index variable exceeds upper and lower bounds before executing statements that access the array.
Collection Size Checker	For a collection type variable, this template adds `if()` statements that check whether an index variable exceeds the size of a given collection object.
Class Cast Checker	For a class-casting statement, this template inserts an `if()` statement checking that the castee is an object of the casting type (using `instanceof` operator).

Figure 3: PAR Fix Patterns

the GA techniques seems inconceivable. Finally, the claim that random search gives results that are equivalent to GA results seems to have been ignored, though it was a portent of things to come.

3.4 Troubled Times: Generate and Validate Since 2014

While GenProg more or less created the field of APR, by 2014 the repeated attacks of PAR's results in readability and RSRepair undermining the value of GAs as a means of generating potential fixes brought the underlying ideas of GenProg under closer scrutiny. G&V methods were found to have serious problems with their assumptions (following GenProg), and two new approaches to APR were created as alternatives. A 2015 study found that developers for GenProg and TrpAutoRepair had not divided the data into training and testing groups. Patches passed all unit tests due to overfitting, and when the experiments were done again, often could not pass unit tests from the test group even half the time.[28] Furthermore, the criteria used to rank candidates (namely, number of unit tests passed) were the same as the considerations of whether their methods were working. Techniques such as Inter-Generational Distance, Spread, Hypervolume (or their analogues for a somewhat specialized solution space) were not considered, and all comparisons between methods were reducible to number of unit tests passed or number of patches created. In particular, Smith found that when applied to programs with few bugs, GenProg and TrpAutoRepair were more likely to break existing functionality than repair buggy sections.

This recognition of problems from overfitting has greatly accelerated the search for alternate means of attacking the problems of APR - both from within the G&V camp and without. Defection from Generate-and-Validate as a strategy had already begun in 2013 with SemFix, and these alternate approaches (discussed below) do seem to do better addressing overfitting concerns, although they come with their own challenges.

3.4.1 Kali - 2015

As researchers began to analyze GenProg's early results more closely in light of recent findings, Qi et al. found another troubling result in the vein that had led to RSRepair and Smith's result: the majority of patches generated from GenProg, AE, and RSRepair were not correct, but only plausible.[28] This distinction means that they passed all unit tests in the training group, but not in the testing group. Worse still, the authors found that even of patches which passed all unit tests (training and testing), the majority were equivalent to deleting poorly-tested functionality.

Hoping that the problem was with insufficient test diversity in the training group, the authors expanded the experiment to encompass stronger test suites, but found that this merely removed the previously plausible patches. GenProg succeeded in making few or no patches on stronger test suites. The authors then suggest (it is suspected tongue-in-cheek) Kali, an APR tool named for the Hindu goddess of death, which is only capable of deleting functionality. The authors find that this tool's edits are comparable to the quality of GenProg and the other G&V techniques, but suggest that while Kali's patches are incorrect, they do offer valuable information about failing behavior.

3.4.2 SPR - 2015

Staged Program Repair with Condition Synthesis (SPR) is one of the most recent G&V type techniques. In this method, repairs are represented as transformation schemas (most previous work represented repairs as patches to the program AST). SPR also begins by finding constraints that would allow test cases to be passed, then using those values to constrain potential branches in the search for repairs. Finally, SPR does not depend solely on existing code from the program, a repository, or template. SPR adds a small pool of allowed innovations which permit patches to be generated with functionality not initially present in the program.[21]

SPR correctly repairs 19 of 69 defects in the GenProg dataset, and furthermore generated the correct repair as the first repair to validate in 11 of the 19 occasions. In terms of plausible repairs (ones which pass all test cases tried) SPR generated 38 plausible repairs, as compared to 16 for GenProg and 25 for AE. A limitation of this method (specifically mentioned in [21]) is that the authors see no way to generalize the mechanisms which make SPR so effective beyond a few additional defect classes.

3.4.3 Prophet - 2015

Prophet is one of the newest methods to join the G&V family, and takes a radically different approach. In an attempt to both constrain the search space and create human-similar patches, Prophet analyzes candidate patches over a

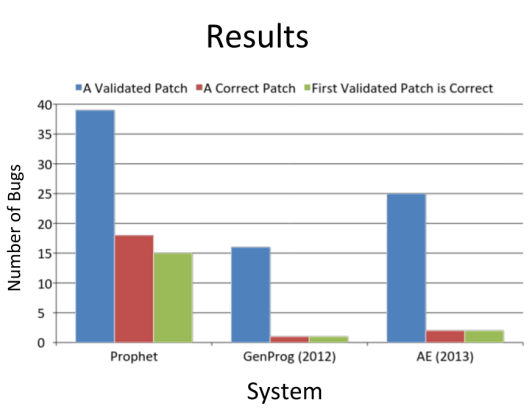

Figure 4: Prevalence of Correct Patches

large patch database prior to taking any input. It uses this information to develop a statistical model[2] of what correct patches look like. Once this model is learned, Prophet uses the model to guide search. The hope is that in addition to increased human-similarity (because model learned from human-generated patches), this model will help overcome weak test suites. For the GenProg benchmark set described above for SPR, Prophet found 14 correct patches which were also the first patch to validate.[20] Figure 4[3] compares Prophet, AE, and GenProg in how many validated patches are evaluated before finding a correct one.[4]

3.5 Semantic Approaches

A second means of avoiding the problems with GenProg is to approach the problem from an entirely different direction. While G&V techniques describe APR as a search problem over a repair space (often encoded as patches over an AST), semantic approaches treat it as a SMT to be solved, often using symbolic or hybrid execution techniques in order to facilitate this approach. The reason these methods are called semantic - and indeed, the reason they are represented differently - is because such approaches seek to describe code at the expression level or higher rather than syntactically. Briefly, G&V generates and validates changes in individual lines of code down to individual characters. Semantic approaches attempt to encode the structure of the program at the expression level or higher and use this structure to improve search.

3.5.1 SemFix - 2013

One of the major differences in semantic techniques is that they do not have selection operators to optimize, and differences tend to depend on how to organize the transformation of the problem into an SMT problem. Once a related SMT problem exists, all techniques simply feed it to a standard SMT solver such as Microsoft's Z3. The major steps of this process are described in SemFix algorithm, shown in Figure 5. Note in particular that this algorithm chains together existing products (Tarantula, KLEE, test suites). This is a common tendency of semantic methods, which generally focus on the boolean algebra of the problem setup and encoding into an SMT problem, but use as many pre-existing products as possible for other portions of the problem.

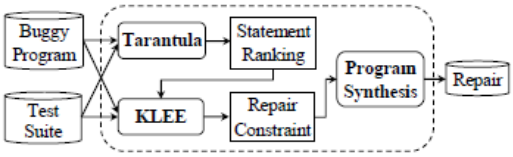

Figure 5: SemFix Algorithm

Because of the increased importance of correct fault localization in semantic methods and the cost of symbolic execution if the wrong area is being used, SemFix uses the Tarantula fault-localization algorithm (although the original paper also tries using Ochiai with negligible differences in performance).

As the first semantic technique, SemFix provided promising results. It succeeded in repairing 48 of the 90 bugs from the GenProg benchmark set (compared to 16 for GenProg), and did so roughly 35% faster than GenProg on average. Further, SemFix was 4x faster than enumerating repair options. This is worth mentioning because semantic methods generally work by enumerating all possible solutions to their constrained problem, somewhat in the spirit of AE. More efficient traversal of this enumeration is a major area of research. However, the fact that runtime vs complete enumeration is even mentioned should ring a warning bell that scalability is an issue with this approach.

3.5.2 Nopol - 2014

Nopol is an extension and modification of SemFix to add two additional defect types: buggy if conditions and missing preconditions. While following the same basic outline as SemFix, Nopol differes in a few details: 1) it uses a different fault localizer (homegrown, but based on the idea of GenProg's), and it includes the idea of angelic debugging.[7] One characteristic of semantic papers (alluded to above) which bears mentioning is that a large part of any paper is spent discussing the boolean algebra of the transformation to and from SMT form. This task is non-trivial and the transformation is often the most important distinguishing feature between methods (as mentioned above).

3.5.3 Angelix - 2016

As with SemFix, scalability problems are major issues for semantic methods. Semantic methods also struggle with multi-point repairs (as do G&V techniques). In 2016, Angelix made significant strides on both problems simultaneously. Angelix is the next step in the line of SemFix and Nopol (and a technique called DirectFix not covered here). It makes extensive use of angelic debugging and significantly extends it to impressively reduce the potential solution space.

The first two steps (placing guards and finding angelic paths) of the Angelix algorithm are shown in Figure 6. First, guards (if statements with temporary variables) are placed before each line. Then, for a chosen value of n, Tarantula

[2]A parameterized log-linear distribution over program modifications taken with error localization with parameters taken by maximum log-likelihood

[3]This graph comes from the Automated Program Repair class notes of Dr. Stolee, Fall 2016.

[4]Note that here "validated" means passing all unit tests (i.e. plausible)

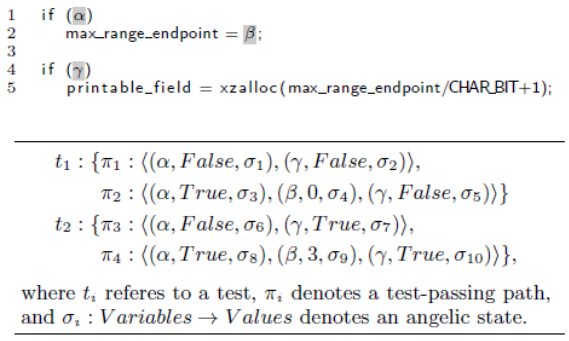

Figure 6: Example of Angelix Algorithm

is used to find the n most suspicious expressions. If an angelic path is found, symbolic execution generates the SMT problem in order to find the correct path. An angelic path is found if there is a means of mapping existing scoped variable values to values which would cause the unit test under consideration to pass. This check means that the expensive SMT encoding and solution is not done unless a solution is possible, i.e. invalid solutions are never evaluated, profoundly improving runtime.

More importantly, setting n > 1 in this algorithm means that Angelix can create multi-point patches. All other techniques to date were either completely incapable of producing patches which require multi-point interactions, or else produced repairs of very low quality.

Human-similarity in patches is achieved by improving the SMT solver. The partialMaxSMT solver used in Angelix allows for hard (mandatory) and soft (desired) constraints. Angelix encodes functionality as hard constraints and minimum change as soft constraints. Thus the SMT solver will automatically provide a solution which repairs the bug while also minimizing source code changes.

Angelix performs slightly worse than SPR in trials, with Angelix fixing 28 defects of the 32 in its defect classes (compared to 31 for SPR), with 10 (11) equivalent to developer fixes. However, considering the 2015 Smith concerns and Kali demonstrations, it is important to check these results for functionality-deleting fixes. 42% of SPR's repairs were functionality-deleting, while only 21% of Angelix fixes were. This suggests that while SPR finds more patches, Angelix produces better patches.

Following [3], the Angelix team does an excellent job of describing which defect classes Angelix can solve. They say that they have provided a method which gives scalable, multi-location fixes with high similarity to human repairs for "side-effect/funtion-call free expressions composed of boolean/arithmetic/relational operators, available variables, and constants."[22] Since function calls, variable creation, and complex operators make up a large part of programming, there is clearly work to be done yet. In addition, only time will tell if Angelix's shocking results have flaws of the type eventually found in GenProg. For now, Angelix is a giant leap forward for APR.

3.6 Database Search Approaches

A final approach to APR which has been suggested in only a single paper [15] claims that semantic methods do not go far enough in trying to mine program structure. If APR has accepted as axiomatic that buggy code is likely implemented correctly elsewhere, why not see if the entire module is implemented elsewhere? In particular, if a repository of common programming types were created, is it possible to turn finding the correct repair not into a search over the abstract space of solutions, but into a database query?

3.6.1 SearchRepair - 2015

SearchRepair is the only method to date to attempt this type of approach. The potential benefits are threefold. First, by replacing large blocks of code with human-written patches, there should by definition be higher human-similarity and thus improved maintainability/industry adoption. Second, it is suggested that human-written patches implementing the same functionality will satisfy unwritten requirements (or requirements not captured by existing unit tests), further increasing patch quality. Finally, by capturing the unwritten requirements, this method should reduce overfitting.

SearchRepair uses Tarantula to localize faults. The buggy region, as well as any surrounding predicate statements[5] is then transformed into a set of SMT input-ouput constraints (think of them like mini-unit tests). These input-output constraints are used to search a database of code samples which are indexed as SMT constraints. Matches are ranked by similarity to desired behavior and validated. A validated match is used to create a patch.

This method was compared to GenProg, RSRepair, and AE, and produced the fewest patches of any method (150, compared to 159, 287, 247 for AE, GenProg, and RSRepair). However, while RSRepair and AE found almost exclusively subsets of GenProg patches (as is expected since those algorithms are modifications to GenProg), 20 of the 150 patches found by SearchRepair were for bugs not patched by any other technique. Furthermore, when the patched program was tested against a second group of unit tests which had not been involved in patch creation or initial validation, over 97% of SearchRepair were correct, with none of the other three techniques achieving more than 72.1% passing on the new tests. This result also further validates the concerns raised by Smith[29] and Qi et al.[28].

The first and most obvious limitation of this method is that unlike G&V and Semantic Search, this method will only produce a repair if a code snippet with the desired functionality is already in the database. This likely contributes to the distinct group of repairs made by SearchRepair. Also, the testing was done on toy problems (very small student-written C programs from a programming course), and it is not clear if this technique will be able to scale. The technique is also not capable of the kind of multi-location fixes shown by Angelix or of fixes which involve simply interchanging parameters in a function call, such as is possible with PAR.

4. CONCLUSION

In this paper, we have reviewed the development of the field of Automated Program Repair from its inception in 2009 to present. From the origins of the field as a collection of specialized techniques most successful in network security, through the years of Generate-and-Validate rule riding on the coattails of GenProg's remarkable insights, to the current state of increasing diversity, the field has made extraordinary progress considering the field is barely seven

[5]i.e. the whole if-then-else if the bug is in such a block

years old.

Many of the issues which initially divided the field have now been resolved. For instance, nearly all methods now use unit tests or similar functionality to describe desired and buggy program behavior. Though fault localization is still an active area of research generally, it is no longer common for new methods in APR to develop their own fault localization techniques. Use of standard products such as Tarantula or copying of standard algorithms such as the localizer from GenProg is very common, and when new localization algorithms are developed, they are most often slight variations on one of the preceding options. The utility of terminology and concepts borrowed from Machine Learning, particularly the idea of overfitting and techniques to avoid it have become increasingly common since 2015, with little to no pushback on adoption. Finally, the goal of APR has shifted from finding algorithms which can create entire programs from scratch to finding algorithms which can repair some software defects with slight modifications to existing code. This shift has not been ideological so much as pragmatic, and while there is widespread acknowledgment of the need to focus on Genetic Improvement at present, Genetic Programming is still a desired goal for many in the field should it ever become feasible (c.f. [10]).

For the future, the short-term need is to continue to learn from colleagues in Machine Learning and Automated Software Engineering, lest 2020 find another paper published which undermines the foundations of current work by applying principles already well understood in related disciplines. The standardization of metrics with which to compare techniques to one another is another immediate need for the field, as the old metrics of tests passed and bugs patched have proven ineffective measures of quality. In light of the entirely laudable efforts of many in recent years to approach the problems of APR from new directions - leading to the development of semantic and search-based approaches in a mere 18 months - an agreed-upon set of metrics which are applicable to all repair strategies is vital.

Longer term, a primary goal should be gaining a more systematic understanding of defect classes, their relationships to one another, and the solubility of each. Despite repeated calls to this task, at present no one has stepped forward to undertake this critical (if thankless) task. Finally, while the short-term suggestions represent years of work for the field and will doubtless propel APR to a point where new and presently unforeseeable issues arise, it seems clear that at some point a deeper understanding of the syntactic, semantic, and structural interactions of code will be necessary in order to make meaningful progress from the current state of the art.

5. ACKNOWLEDGMENTS

Several of the ideas in this paper are indebted to the class notes for Katie Stolee's Automated Program Repair (Fall 2016), as well as Figure 3, which is taken directly from them. In addition, graphics were taken from the original papers they describe, excepting only the steps in APR table.

6. REFERENCES

[1] R. Abreu, P. Zoeteweij, et al. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[2] R. Al-Ekram, A. Adma, and O. Baysal. diffx: an algorithm to detect changes in multi-version xml documents. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.

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

[4] A. Arcuri and X. Yao. Coevolving programs and unit tests from their specification. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 397–400. ACM, 2007.

[5] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 162–168. IEEE, 2008.

[6] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *ACM SIGPLAN Notices*, volume 38, pages 97–105. ACM, 2003.

[7] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 121–130. IEEE, 2011.

[8] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 65–74. IEEE, 2010.

[9] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.

[10] M. Harman, Y. Jia, and W. B. Langdon. Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system. In *International Symposium on Search Based Software Engineering*, pages 247–252. Springer, 2014.

[11] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–14. ACM, 2012.

[12] A. Hill. Title forthcoming. *preprint*, 2016.

[13] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *ACM SIGPLAN Notices*, volume 46, pages 389–400. ACM, 2011.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[15] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (t). In *Automated Software Engineering (ASE), 2015 30th*

IEEE/ACM International Conference on, pages 295–306. IEEE, 2015.

[16] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[17] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, (99), 2013.

[18] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[19] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[20] F. Long and M. Rinard. Prophet: Automatic patch generation via learning from successful patches. 2015.

[21] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[22] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701. ACM, 2016.

[23] M. Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[24] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[25] 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. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[26] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 180–189. IEEE, 2013.

[27] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[28] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[29] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[30] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[31] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[32] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.

[33] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[34] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[35] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Software Engineering - ESEC/FSE 1999*, pages 253–267. Springer, 1999.