3-2016

History driven program repair

Xuan-Bach D. LE
Singapore Management University, dxb.le.2013@phdis.smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Claire LE GOUES

2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

History Driven Program Repair

Xuan-Bach D. Le, David Lo
School of Information Systems
Singapore Management University
{dxb.le.2013,davidlo}@smu.edu.sg

Claire Le Goues
School of Computer Science
Carnegie Mellon University
clegoues@cs.cmu.edu

***Abstract*—Effective automated program repair techniques have great potential to reduce the costs of debugging and maintenance. Previously proposed automated program repair (APR) techniques often follow a generate-and-validate and test-case-driven procedure: They first randomly generate a large pool of fix candidates and then exhaustively validate the quality of the candidates by testing them against existing or provided test suites. Unfortunately, many real-world bugs cannot be repaired by existing techniques even after more than 12 hours of computation in a multi-core cloud environment. More work is needed to advance the capabilities of modern APR techniques.**

We propose a new technique that utilizes the wealth of bug fixes across projects in their development history to effectively guide and drive a program repair process. Our main insight is that recurring bug fixes are common in real-world applications, and that previously-appearing fix patterns can provide useful guidance to an automated repair technique. Based on this insight, our technique first automatically mines bug fix patterns from the history of many projects. We then employ existing mutation operators to generate fix candidates for a given buggy program. Candidates that match frequently occurring historical bug fixes are considered more likely to be relevant, and we thus give them priority in the random search process. Finally, candidates that pass all the previously failed test cases are recommended as likely fixes. We compare our technique against existing generate-and-validate and test-driven APR approaches using 90 bugs from 5 Java programs. The experiment results show that our technique can produce good-quality fixes for many more bugs as compared to the baselines, while being reasonably computationally efficient: it takes less than 20 minutes, on average, to correctly fix a bug.

***Keywords*—Automated Program Repair, Mutation Testing, Graph Mining**

I. Introduction

Bugs are prevalent in software development. Mature commercial software systems regularly ship with both known and unknown defects [31], despite the support of multiple developers and testers typically dedicated to such projects [3]. To maintain software quality, bug fixing is thus inevitable and crucial. Yet, bug fixing is notoriously a difficult, time-consuming, and labor-intensive process, dominating developer time [51] and the cost of software maintainance. Many defects, including security-critical defects, remain unaddressed for extensive periods [19], and the resulting impact on the global economy is measured in the billions of dollars annually [48], [10]. There is a dire need to develop automated techniques to ease the difficulty and cost of bug fixing in practice.

To address the above-mentioned need, substantial recent work proposes techniques for Automated Program Repair (APR). These techniques seek to automatically fix bugs by producing source-level patches. For example, GenProg [28] uses a Genetic Programming [26] heuristic to conduct a search for a patch that causes the input program to pass all given test cases (including at least one that initially failed, exposing the defect to be addressed). Subsequently, Kim *et al.* extend the GP approach in Pattern-based Automatic program Repair (PAR), which uses bug fix templates manually learned from existing human-written patches [25] to guide the creation of the potential patches. These techniques are instructive examples of generate-and-validate and test-case-driven approaches to defect repair: They generate many candidate patches, and validate them against a set of test cases. The process is repeated many times, with a *fitness score* computed for each candidate patch based on the number of test cases that the associated modified program passes or fails. This score guides subsequent iterations, and thus the way the techniques traverse the search space of candidate repairs.

Despite the promise of existing APR techniques, current approaches are limited in several key ways [43]. To truly improve the quality of real-world software as well as the experience of modern software developers, an ideal technique must be both *effective* (i.e, able to fix many real bugs) as well as *efficient* (i.e., able to do so in a short amount of time). Even merely *plausible* patches—those that lead the buggy program to pass the provide test cases, but that are not necessarily globally correct as judged by an informed programmer—may take more than 10 hours to generate, and the resulting patches may still be incorrect [25], [43]. Although the risk of low quality patches can be mitigated by using more comprehensive test suites to guide the search process, even with full-coverage test suites, existing test-guided techniques may be susceptible to *overfitting* [47]. That is, produced patches may fail to functionally generalize beyond the test suite used to produce them. Although the current APR state-of-research is still in its infancy, it is important to work towards both effectiveness and efficiency to allow APR to be ultimately adopted.

In this paper, we propose a novel technique for *history-based program repair*. Like several previous methods, our technique makes use of a stochastic search process to generate and then validate large numbers of patches, seeking one that causes previously-failing tests to pass. The most important feature differentiating our new technique from the previous work is that it evaluates the fitness or suitability of a candidate patch by assessing the degree to which it is similar to prior bug-fixing patches, taken from a large repository of real patches. This is in contrast with previous search-based approaches, which by and large use input test cases to assess intermediate patch suitability. Our intuition is that bug fixes are often similar in

978-1-5090-1855-0/16 $31.00 © 2016 European Union
DOI 10.1109/SANER.2016.76

nature and past fixes can be a good guide for future fixes. This has at least partially informed a number of previous studies and approaches [7], [17], [25], [33]. The important novelty in our technique is that, instead of simply using previous fixes to inform the *construction* of candidate patches, we use fix history to help assess their potential *quality*, or fitness. We expect that the history-driven approach mitigates the risk of overfitting to the test suite, because it does not directly use the test suite score to guide individual selection for later iterations. This increases the probability that the resulting patches generalize to the desired program specification. Moreover, using the history to guide the repair search can also imbue the APR process with history-informed "common sense" to identify plausible but clearly—to humans—nonsensical patches.

To illustrate, consider the buggy code snippet in Figure 1, taken from Math version 85 in Defect4J benchmark [22]. This buggy snippet throws a `ConvergenceException` when one of the test cases is executed. One low-quality way to "fix" the problem that eliminates the symptom, and causes the test case to trivially pass, simply deletes the `throw` statement. However, this would be a nonsensical solution, and is not consistent with the patch the human developer committed for the same defect. Unfortunately, prior generate-and-validate and test-case-driven APR techniques cannot identify such a solution as nonsensical. In our history based approach, on the other hand, the fact that such edits very rarely appear in the historical bug fix data means that it receives a very low score in the search process. In this way, our technique is more likely to avoid plausible but nonsensical patches.

```
//Human fix: fa * fb > 0.0
if (fa * fb >= 0.0 ) {
  throw new ConvergenceException("...")
}
```

Fig. 1: Bug in Math version 85

Our history-based APR technique works in three phases: (1) *bug fix history extraction* (2) *bug fix history mining* and (3) *bug fix generation*. The first two phases are conducted in advance of any particular bug-fixing effort. In the first phase, our technique mines historical bug fixes from revision control systems of hundreds of projects in GitHub. In the second phase, our technique identifies a clean set of data, seeking to find frequently appearing or common bug fixes, and infering a common representation to capture many similar such bug fixes. Bug fixes are represented as change graphs, which have the benefit of being generic and able to capture various kinds of changes along with their contexts. These change graphs, along with their frequencies, are used as a knowledge base for the third phase. In the third phase, our technique iteratively generates candidate patches, ranks them based on the frequency with which their constituent edits appear in the knowledge base inferred in the second phase, and returns a ranked list of plausible patches that pass all previously failed test cases as recommendations to developers.

We have evaluated our solution on 90 real bugs from 5 Java programs. We compare our technique against GenProg and PAR. GenProg is a popular generate-and-validate and test-case-driven APR technique that with a publicly available Java implementation.[1] Similarly, PAR is a generate-and-validate, test-case-driven technique developed for Java programs that explicitly makes use of edit templates manually synthesized from edit histories. Both are generic approaches that can, in theory, produce multi-line patches for bugs in programs. Our experiments show that our approach can correctly fix 23 bugs out of the 90 programs, while PAR and GenProg are only able to correctly fix 4 and 1 bugs, respectively. Moreover, our approach on average only needs 20 minutes to fix the 23 bugs. The results demonstrate the effectiveness and efficiency of our proposed approach.

The contributions of our work are as follows:

1) We propose a *generic* and *efficient history-based* automatic program repair technique that uses information stored in revision control systems of hundreds of software systems to generate plausible and correct patches. Our approach is generic since it can deal with bugs whose fixes involve multi-line changes. It is efficient since it can complete on average within less than 20 minutes.
2) We demonstrate that our approach is effective in fixing 23 bugs correctly, dramatically outperforming the performance of the baseline solutions.
3) Our approach supports Java instead of C. Java is the most popular programming language and its influence is growing over time.[2] Prior generate-and-validate and test-case-driven APR techniques mostly work on C programs with a few exceptions (e.g., PAR). Unfortunately, the implementation of PAR is not made publicly available. To facilitate reproducible research, we made the implementation of our approach available at https://github.com/xuanbachle/bugfixes

The structure of the remainder of this paper is as follows. In Section II, we elaborate the three steps of our proposed approach. In Section III, we present our experiment results which answer four research questions. We discuss related work in Section IV and conclude in Section V.

II. Proposed Approach

The overall goal of our approach is two-fold: to generate correct, high-quality bug fixes; and to quickly present such fixes to developers. To achieve this goal, we divide our framework into three main phases: (1) *bug fix history extraction*, (2) *bug fix history mining* and (3) *bug fix generation*. The first phase extracts a dataset of bug fixes made by human in the history from GitHub. This dataset is input to the second phase, which converts the bug fixes to a graph-based representation from which it automatically mines bug fix patterns. The mined bug fix patterns are input to the last phase.

In the last phase, we use a modified stochastic search technique [16] to evolve patches to a given buggy program, until we find a desired number of solutions. To reduce the risks of either overly constraining the search space or overfitting to the test suite, we use 12 existing mutation operators previously proposed in the mutation testing literature and used by prior repair techniques [38], [29], [25]. The *fitness* of the generated fix candidates is determined by the *frequency* with which the

[1] https://libraries.io/github/SpoonLabs/astor
[2] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

changes included in a given patch occur in the mined bug fix patterns produced by the second phase. Better fix candidates are thus those that frequently occur in the past fix patterns, and are thus more likely to be chosen to be validated against failed test cases, i.e., the test cases that reveal the bug in the original buggy program. Such selected patches are also more likely to be further developed and evolved in subsequent iterations. This fix candidate generation process is repeated until a desired number of candidates that pass all the failed test cases is identified. At the end of this phase, these candidates are presented to the developer as possible fixes for the bug, ranked by the frequency with which their edits appear in the bug fix history. The developer can then investigate the suggested fixes to find an actually correct fix.

In the next subsections, we describe each of the three phases of our framework in more detail (Sections II-A–II-C); Section II-D describes our mutation operators.

A. Bug Fix History Extraction

In this phase, we collect human-made bug fixes from many open source projects on Github. The primary purpose of this phase is to collect and collate commits that are solely related to bug fix actions, excluding feature requests, refactoring, and other non-repair types of edits.

To collect bug fix history data from GitHub, we follow the procedure described by Ray *et al.* [44] to gather large, popular, and active open source Java projects. In particular, we use *Github Archive* [1], a database that frequently records public activities from GitHub, e.g., new commits, fork events, etc, to select only projects with the above characteristics. The popularity of a project is indicated by the number of *stars* associated with its repository, which corresponds to the number of GitHub users that have expressed interest in that project. In the interest of identifying only large, popular projects, we filter out those with fewer than five stars and exclude projects with repositories smaller than 100 Megabytes. Finally, we retain projects that are active as of September, 2014. This still leaves us with thousands of projects.

For each of the retained projects, we iterate through its source control history, seeking to collect commits that exclusively concern bug repair. This is a difficult problem in repository mining [8]. We therefore seek a complete set of bug-fixing commits using heuristics, though acknowledge that our approach is best-effort. We deem a commit a bug fix if it simultaneously satisfies the following conditions:

1) Its commit log contains the keywords such as *fix, bug fix*, while not containing keywords such as *fix typo, fix build* or *non-fix*.
2) It includes the submission of at least one test case in that commit. Although the submitted test case does not necessarily mean the one that induces the bug, the inclusion of test case in the commit further increases the likelihood that the commit is a bug fix.
3) It involves changes on no more than two source code lines. The changed lines are counted, excluding code comments.

This last requirement warrants additional explanation. Commits that satisfy the first condition but involve many changed lines typically include changes beyond the bug fix, addressing feature addition, refactoring, etc [18], [23]. Thus, we filter out commits involving more than two changed lines. Ultimately, this leaves us with 3000 bug fixes across 700+ large, popular and active open source Java projects from GitHub.[3]

B. Bug Fix History Mining

In the second phase, we mine frequent bug fix patterns from the 3000 bug fixes, that appear in more than 700 projects, collected by previous phase. We first convert the collected bug fixes into a graph-based representation. We then apply an existing graph mining technique to the dataset to mine closed frequent patterns from the converted graphs.

Graph-based representation of bug fixes. Our goal in representing bug fixes is to succinctly abstract similar bug fixes into a common, abstract representation amenable to mining, which is especially challenging in the face of naming differences. Different bug fixes may vary in terms of the naming scheme in the underlying code, containing modifications to different variable names, method names, etc. For example, Figure 2 shows two bug fixes that both involve the change of method call parameter. Although there are differences in the expressions (variables) that invoke the method calls, the method call names and parameter names, conceptually, these bug fixes can easily be classified as the same kind of bug fix, i.e., "method call parameter replacement."

Our first step in storing a bug fixing change is to capture its effects at the Abstract Syntax Tree (AST) level, which abstracts away many incidental syntactic differences (e.g., whitespace, bracket placement) that obscure a patch's semantic effect. To this end, we use GumTree,[4] an off-the-shelf, state-of-the-art tree differencing tool that computes AST-level program modifications [15]. GumTree represents differences between two ASTs via a series of actions including additions, deletions, updates or moves of individual tree nodes to transform one AST to another. To do this, given a bug fix, we first identify the file modified by the bug fix, and then retrieve the versions of the file before and after the modifications were made. Both versions of the modified file are then parsed to ASTs, denoted as the "buggy AST" and "fixed AST," respectively. We then use GumTree to compute the actions needed to transform the buggy AST into the fixed AST. For example, GumTree gives us the action needed to represent the *bug fix 1* in the Figure 2 as *update* from *x1* to *x2*.

However, this raw information provided by GumTree is insufficiently abstract on its own, since it is still specific to the variable names *x1* and *x2*. Additionally, the semantic context surrounding the action is unclear, that is, whether the action applies to a method call, an assignment, etc. To remedy this issue, we convert the series of actions produced by GumTree into a labelled directed graph that further abstracts over the edit actions, while being able to capture surrounding semantics. In this directed graph representation, an edge from a parent vertex to a child vertex is labelled by the kind of the action made to the child vertex. The context of the action is then captured by the parent vertex. To illustrate by example, Figure 3 depicts the

[3]Dataset available: https://github.com/xuanbachle/bugfixes

[4] https://github.com/GumTreeDiff/gumtree

graph that represents the *bug fix 1* in Figure 2. Similarly, this graph also represents the change made in *bug fix 2*. Thus, by using this graph representation, we can represent bug fixes in a common abstraction and capture contexts of the bug fixes. This graph-based representation will then help us in using graph mining techniques to mine frequent bug fix patterns.

```
//Bug fix 1: x1 replaced by x2, others remain
    the same
       - obj1.doX(x, x1)
       + obj1.doX(x, x2)

//Bug fix 2: y1 replaced by y2, others remain
    the same
       - obj2.doY(y, y1)
       + obj2.doY(y, y2)
```

Fig. 2: Example of two bug fixes involving method call parameter replacement.

Mining closed frequent bug fix patterns. Given the full set of bug fixes, represented as graphs, we mine *closed frequent patterns* from the graphs. A pattern is *frequent* if it often occurs in the database; we heuristically set this count to at least two. A frequent pattern g is *closed* if there exists no proper supergraph of g that has the same number of supergraphs, i.e., *support*, as g. Thus, by definition, closed frequent patterns are the largest possible patterns that frequently occur in the database. In our domain, our goal is to mine the largest possible bug fix patterns to precisely capture behaviours of the changes. We therefore employ an extension of gSpan,[5] an implementation of a state-of-the-art frequent graph miner [52] for this task. We consider a pattern is frequent if it has support greater than or equal to two. We store information about patterns, including each pattern's vertices, edges, and supergraphs that contain the pattern. The number of supergraphs of a pattern constitutes the frequency of the pattern.

C. Bug Fix Generation

Overview. In this phase, we use a stochastic search approach loosely inspired by genetic programming [26] to evolve a patch for a given buggy program. The search objective is a patch that, when applied to the input program, addresses the defect, as identified by a set of failing test cases. GP is the application of genetic algorithms (GA) to problems involving tree-based solutions (programs, typically; in our application, these are small edit programs applied to the original buggy program). A GA is a population-based, iterative stochastic search method inspired by biological evolution. Given a tree-based *representation* of candidate solutions, GP uses computational analogues of biological mutation and crossover to generate new candidate solutions, and evaluates solutions using a domain-specific objective, or *fitness* function. Potential solutions with high fitness scores are more likely to be randomly retained into future iterations both alone, modified slightly (via *mutation*), or, in some applications, in combination with other solutions (via *crossover*).

[5] https://www.cs.ucsb.edu/ xyan/software/gSpan.htm

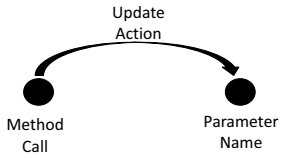

Fig. 3: Graph-based representation of bug fixes in Figure 2

In our approach, we represent a single candidate solution as a patch consisting of a sequence of edits to be made to the buggy program; this representation has been shown both efficient and effective in search-based program improvement [30]. Given a population of candidate solutions, we then use a selection process to create new candidates through mutation, and then to select mutated candidates to subsequent generations for additional evolution. The selection phase applies to the mutation step, in which a new edit is pseudo-randomly constructed and then added to an existing (possibly-empty) candidate patch. This selection is informed by the bug fix history database constructed as discussed in Section II-B. Note that our algorithm does not perform crossover, using only mutation to create new individuals; we leave the development of a suitable crossover operator in our context to future work.

The details of this phase are further described in Algorithm 1. The primary inputs to the algorithm are the buggy program, where the bug is indicated by one or more failing test cases; a set of faulty locations, weighted by a preexisting fault localization procedure; a distribution of edit frequencies mined as discussed in the previous section, and a set of possible mutation operators. We presently assume that the faulty methods are known in advance, as file- and method-level localization represent an orthogonal problem; we then compute the faulty lines in each prospective faulty method using existing statistical fault localization techniques [2]. The stochastic algorithm includes several tunable parameters, described in context.

Given those inputs, the algorithm works in multiple iterations. The first iteration constructs an initial generation of *PopSize* candidate solutions by repeatedly constructing single-edit patches for the program (lines 16–18). Subsequent generations are created by adding new mutations to retained solutions in the current population. We describe mutation as it is used to create the initial population of single-edit patches; its application in subsequent iterations follows naturally.

Mutation. The mutation procedure adds an edit to a (possibly-empty) candidate patch to create a new patch candidate. It is described from line 4 to line 12 in Algorithm 1. At a high level, the mutation step creates a large number of candidate edits, from which a single edit is ultimately propagated into the candidate patch. First, the algorithm randomly selects a subset of L fault locations to which mutations may be applied (line 5), weighted by the score given by the statistical fault localization. We heuristically set L to 10 in our experiments, leaving a full parameter sweep to future work. Next, lines 7 to 10 generates a set of possible edits to select in this mutation step. This involves first identifying which mutation operators can be applied to each of the prospective faulty locations. For

Algorithm 1: Bug Fix Generation. The `select` procedure returns one or more individuals from a population, either uniformly or weighted by a provided function. `applies` and `instant` are described in text. The tunable parameters are *PopSize* (population size), *M* (desired solutions), *E* (number of seeded candidates to the initial population), and L (number of locations considered in the mutation step).

```
   Input : BugProg: Buggy program
           FaultLocs: Fault locations
           NegTests: initially failing test cases
           FixPar: mined edit frequency distribution
           ops: possible operators
           params: Tunable parameters PopSize, M, E, L
   Output: A ranked list of possible solutions
1  helper fun editFreq(cand)
2      let N ← |cand|
3      return (Σ_{i=0}^{N-1} FixPar(cand_i)) / N
4  helper fun mutate(cand)
5      let locs ← select(FaultLocs, L)
6      let pool ← ∅
7      foreach f ∈ locs do
8          let op_f ← ⋃_{op∈applies(ops,loc)} instant(op, loc)
9          let cand' ← cand + select(op_f, 1)
10         pool ← pool ∪{ cand' }
11     end
12     return select(pool, 1, editFreq)
13 fun main
14     let Solutions ← ∅
15     let Pop ← {E empty patches}
16     while | Pop | < PopSize do
17         Pop ← Pop ∪ mutate([ ])
18     end
19     repeat
20         foreach c ∈ Pop do
21             if c ∉ Solutions then
22                 if c passes NegTests then
23                     Solutions ← Solutions ∪ c
24                 else
25                     c ← mutate(c)
26                 end
27             end
28         end
29     until |Solutions| = M
30     return Solutions
```

example, we should not use *append* to add any statements after a *return* statement, because doing so results in a dead code. This check is performed by the function `applies` on line 8. We reuse existing mutation operators (see Section II-D for a complete list, and details about their application), that have been proposed in prior mutation testing and program repair studies, to provide a diverse set of bug fix edit candidates. There are often several ways to instantiate a given operator. For example, *append* requires the selection of fix code to append at a given location. The `instant` function returns all possible instantiates of a given operator to the provided location, also on line 8. We select one of these edits (line 9) and create a new candidate by adding it to the current candidate.

This results in an intermediate *pool* of new pseudo-random patch candidates (initialized on line 6, updated on line 10) from which a single candidate will be retained. This retained candidate is thus the single result of the mutation step; it is the result of adding a new random edit to the (possibly-empty) candidate patch under mutation. To pseudo-randomly select an edit from this pool, we weight each edit by the *frequency* with which it appears in the mined bug fix patterns. This computation is performed in helper function `editFreq`, used in selection on line 12. Note that since exact graph matching (isomorphism) is notoriously difficult and expensive [49], we relax the conditions of matching fix candidates against past fix patterns. We instead say a fix candidate matches a fix pattern (graph) if the graph representing the candidate has more than half of its labels of vertices and edges matched with the fix pattern's vertices and edges respectively.

The frequency formula at line 3 works as follows: Given a fix candidate consisting of N edit operations, each edit operation contributes equally to the candidate's frequency. That is, we break down the block of N edits into each constitute edit and then fuse the frequency of each small edit together. The intuition is that, due to the randomness of the mutation procedure, generated fix candidates may contain bug-fix irrelevant edit operations, e.g., field or variable declarations. Ideally, these irrelevant edits should not affect the score of fix candidates containing them, since such edits contribute nothing or very little to the fixing effort. If we count the frequency of the fix candidate consisting of these edit operations by the whole block of N combined edits, it would make the fix candidate very rare when comparing the candidates against the historical bug fix patterns, and reduce the likelihood that the fix candidate will persist for future evolution. Our use of mean edit frequency mitigates the effect of adding bug-fix irrelevant edit operations with respect to the viability of the overall patch.

At line 12, we pseudo-randomly select one edit from the pool to add to the current candidate solution. This selection is informed by the computed frequency of a candidate patch that includes each edit in turn (the higher the frequency score of the overall patch that includes it, the more likely it is that the potential edit is selected from the pool). We use stochastic universal sampling [6] for this task. This selected candidate is thus the return value of the mutation procedure.

Main algorithm. Mutated candidates are created and processed by the main algorithm, described from line 13 to line 30. Line 15 adds E number of empty candidate patches to the initial population as seeds. We heuristically set E to 3 in our algorithm. Lines 16–18 create an initial population with $PopSize$ candidate patches by repeatedly mutating the empty patch. We heuristically set $PopSize$ to 40 in our algorithm. Next, from line 20 to line 28, we validate each candidate in the current population against the failed test cases. If a candidate passes all the failed test cases, we add it to the set of possible solutions (line 23). Otherwise, we mutate the candidate and carry the mutated candidate over the next iteration (line 25).

The process continues until a given number of fix candidates that pass all the previously failed test cases is reached. This is indicated at line 29, where the solutions' size reaches M desired solutions. We heuristically set M to 10 in our algorithm. Ultimately, these candidates are presented to the developer as possible fixes to the buggy program, ranked by the frequency of the underlying edits. The developer is then responsible for assessing the correctness of the suggested fixes.

TABLE I: 12 mutation operators employed in our framework

Operator Action	Description
GenProg Mutation Operators	
Insert statement	Insert a statement before or after a buggy statement
Replace statement	Replace a statement with a buggy statement
Delete statement	Delete a buggy statement
Mutation Testing Operators	
Insert Type Cast	Cast an object to a compatible type
Delete Type Cast	Delete type cast used on an object
Change Type Cast	Change type cast to another compatible type
Change Infix Expression	Change primitive operator (arithmetic, relational, conditional, etc) in an infix expression
Boolean Negation	Negate a boolean expression.
PAR Mutation Operators	
Replace Method Call parameter	Replace a parameter in a method call by another parameter with compatible types.
Replace Method Call Name/Invoker	Replace the name of a method call, or a method-invoking expression, by another method name or expression with compatible types.
Remove Condition	Remove a boolean condition in an existing *if* condition
Add Condition	Add a boolean condition to an existing *if* condition

For example, the developer can pick any of the fixes appearing on top of the recommendation to validate the fixes by running them against previously passed test cases, and see if the fixes are actually semantically correct or not.

D. Mutation Operators

In this section, we describe the 12 mutation operators we employ to generate fix candidates in our framework; these operators are listed in Table I. These operators have been used previously in mutation testing and well-known repair techniques; we use them to simultaneously provide a broad array of potential edit types, while mitigating the risk of overfitting the operators used in our experiments to the underlying dataset.

GenProg Mutation Operators. We employ the three mutation operators from GenProg [28], [29]. The *delete* operator deletes a given potentially-buggy statement. The *insert* and *replace* operators work under the assumption that the repair is a piece of code that can be found from somewhere else in the same program. The insert operator inserts a randomly-selected statement before or after a given buggy statement.The *replace* operator replaces a potentially-buggy statement with another randomly-selected statement. For insert and replace, the original GenProg algorithm randomly chooses a statement from elsewhere in the same program, given certain semantic constraints (e.g., variable scoping). However, given a time limit, a large program can enormously reduce the possibility of selecting the correct statement.

We mitigate this problem in several ways. First, we reduce the scope of source statement selection to the same file with the target buggy statement. Previous studies have shown that this is adequate for many automated program repair problems [7]. Second, we heuristically prioritize in-scope statements. We view the problem of finding the source statement as two stages: First, we find the *clones* of the piece of code (method) surrounding the target buggy statement. Second, each of statements in the clones that have higher similarity is given higher probability to be a source statement. For statements that are not in any clones, we give them a default probability which is less than the probabilities of any statements found in clones. To find clones, we employ tree-based clone detection technique described by Jiang et al. in [20].

Mutation Testing Operators. We employ five mutation operators proposed in mutation testing research [38], [39]. The first three concern type casting: *delete type cast*, *insert type cast*, and *change type cast*. The latter two focus on inserting or changing casts only to compatible types. The *change infix expression operator* changes the operator used in a given infix expression. For example, an infix expression like $a \geq b$ involves an arithmetic operator that can be randomly changed, such as to $a > b$, $a < b$ or $a \leq b$. An infix expression $a \neq b$ that involves relational operator can be changed to $a == b$. An infix expression $a \;\&\&\; b$ that involves conditional operator can be changed to $a \;||\; b$ and vise versa. The *boolean negation* operator tries to negate a boolean expression. For example, $true$ can be negated to $false$, and $isNegative(a)$ can be negated to $!isNegative(a)$.

PAR Mutation Operators. We employ four out of ten mutation operators proposed by Kim et. al. [25], leaving the employment of the remaining six operators as future work. These operators are applied to either method call or *if* condition. The first operator replaces a method call parameter, while the second operator replaces method call name, or the expression that invokes the method call. The last two operators deal with condition expression of `if` statement. An `if` condition expression containing more than two conditions can apply the *remove condition expression*. For example, `if(a || b){...}` can be changed to `if(a){...}` by removing condition b, which is randomly chosen from the condition. The *add condition expression* tries to add a condition to an `if` condition. The condition to be added is chosen from a pool of conditions collected from the same file with the faulty `if` statement. However, this pool of collected conditions can be inappropriate to fix a given bug, since they may reference out of scope variables.

To address this, our framework further cultivates the search space by inventing new conditions that have not appeared elsewhere in the same file. The idea is that the missing condition may very likely involve one of the variables used in the current `if` condition. Toward this end, we collect all variables used in the `if` condition. We then collect all boolean usages that involve the types of the collected variables from the same file. We then apply the usages with the collected variable names, and add these usages to the pool possible conditions that can be added to the current `if` condition.

III. Experiments and Analysis

In this section, we first describe our dataset (Section III-A), followed by our experimental settings (Section III-B), research questions (Section III-C), and results (Section III-D). We conclude with a discussion of threats to validity (Section III-E).

A. Dataset

We apply our approach to repair a subset of bugs from Defects4J [22], a large collection of defects in Java program intended to support research in fault localization and software quality. Defects4J has also been used in previous study of several automated program repair (APR) tools [14]. The

TABLE II: Dataset Description. "#Bugs" denotes the total number of bugs in the Defects4J dataset. "#Bugs Exp" denotes the number of filtered bugs we used in our experiments.

Program	#Bugs	#Bugs Exp
JFreeChart	26	5
Closure Compiler	133	29
Commons Math	106	36
Joda-Time	27	2
Commons Lang	65	18
Total	357	90

dataset contains 357 real and reproducible bugs from 5 real-world open source Java programs. In our experiments, we use 90 bugs from Defects4J.[6] Table II depicts the number of bugs from each program in Defects4J and the number of bugs from each program that are used in our experiments. We use only these 90 bugs out of 357 bugs in Defects4J since we filtered out bugs that are too difficult for current state-of-the-art repair techniques to fix. That is, we first filter out bugs that involve more than six changed lines since they are typically too difficult for current automated program repair techniques to fix [42]. Second, we also filter out too difficult bugs considering the semantics of the bugs, even though they involve changes that are syntactically fewer than six lines. For example, one kind of difficult bugs could be adding a field in a class and use that field for fixing bugs in methods. We hypothesize that an effective and usable APR technique should be able to fix classes of bugs that are easier to fix first before it can handle very difficult bugs. We thus prefer this dataset, filtered according to rules suggested in previous empirical studies to a completely manually-constructed dataset to mitigate to some degree the threat over overfitting our technique to the bugs under repair [35]. We use the fix template database constructed as described in Sections II-A–II-B.

B. Experiment Settings

We compare our approach against PAR [25] and GenProg [29]. Since PAR is not publicly available, we re-implemented a prototype of PAR for this experiment based on our framework. We also note that the original version of GenProg works on C programs and thus we used a publicly available implementation of GenProg[7] that works on Java program provided by Monperrus *et al.* [14].

We assign one trial for each approach to run on each bug. Specifically, each trial of our approach is assigned one 2.4 GHz Intel Core i5-2435M CPU and 8GBs of memory. Each trial is terminated either after 90 minutes or 10 generations or if 10 possible solutions were found. The size for each population is set to 40 for consistency with previous work [29], [25]. Since we consider current automated program repair techniques as only recommendation systems (since they cannot fix most of the bugs yet), an automated program repair technique needs to be efficient enough (c.f., [27]). We thus set the timeout for our experiment as 90 minutes for each trial. We note that since our approach, PAR and GenProg are all stochastic, multiple trials are needed to properly assess their performances [4]. We discuss this in threats to validity.

[6]The bugs are made available here: https://github.com/xuanbachle/bugfixes

[7]https://libraries.io/github/SpoonLabs/astor

C. Research Questions

In our experiments, we seek to answer the following research questions:

RQ1 How many bugs can our technique fix, correctly, as compared to the baselines?

We compare the effectiveness of our approach against PAR and GenProg in terms of number of bugs that each approach can correctly fix. To do this, the first author of the paper manually inspected generated patches to verify their correctness with respect to the corresponding bugs. A patch is deemed a correct fix if it satisfies the following conditions: (1) It results in a program that passes all test cases (both passing and initially failing). (2) it follows the behavior of the corresponding human-made fix. Checking the first condition is not difficult. However, the second condition involves an intrinsic qualitative judgement and a deep understanding of the program in question. Thus, for the second condition, we only consider fixes that are as close as possible to the human-made fixes. We leave a comprehensive human study on bug fixes quality to future work.

RQ2 Which bugs can the approaches fix in common? Which bugs can only be repaired by one of the approaches?

To gain insight into the process and limitations of the different approaches, we identify the defects for which our approach, PAR and GenProg all generate correct fixes. We describe case studies that illustrate potential reasons why some bugs can be fixed by one approach but not others.

RQ3 How long does it take to produce correct fixes?

In this research question, we investigate the average amount of time for each approach to run on the bugs that they can correctly fix. An approach is deemed efficient if it needs a reasonable computation time to find correct fixes. We consider current automated program repair techniques as recommendation systems, and a recommender that takes several hours to produce recommendations is ineffective.

RQ4 What are the rankings of the correct fixes among the solutions that our approach presents to the developer?

Our approach generates a ranked list of possible solutions to a given bug. The higher a correct fix is ranked, the better, requiring less effort from the developer to try the solutions one by one from the top to the bottom. Thus, in this research question, we investigate the ranking of the correct fixes among the possible solutions that our approach presents to the developer.

We report on two types of ranking. First, we present the ranking in the order that fixes are generated temporally. If effective, this ranking is helpful in case the developer is rushing to clear the bug, since he or she can just try whatever suggestions appear earlier instead of waiting for the whole process to complete. Second, we assess a ranking based on the frequency with which fix edits appear in the historical data.

TABLE III: Effectiveness of our approach, PAR and GenProg in terms of number of defects repaired from each program.

Program	Our Approach	PAR	GenProg
JFreeChart	2/5	-/5	-/5
Closure Compiler	7/29	1/29	-/29
Commons Math	6/36	2/36	-/36
Joda-Time	1/2	-/2	-/2
Commons Lang	7/18	1/18	1/18
Total	23/90	4/90	1/90

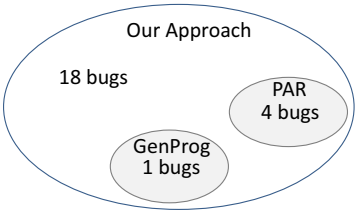

Fig. 4: Common Bugs Fixed by Program Repair Techniques

D. Results

RQ1: Number of Bugs Correctly Fixed. Table III depicts the number of bugs for which each approach can generate correct fixes. In total, out of the 90 bugs, our approach generates correct fixes for 23 bugs; PAR can only correctly fix 4 bugs; GenProg generates only one correct fix. For the 23 bugs fixed by our approach, 11 (out of 12) mutation operators help fix these bugs. Each of these 11 operators helps fix no more than 5 bugs in the 23 bugs. Thus, it is not the case that the use of only a few operators can help fix all of the 23 bugs fixed by our approach. This supports our belief that while our approach is more effective than the baselines, its effectiveness is less likely to be biased by the experimental dataset.

Although the results for the previous techniques are somewhat worse than expected, we note that our timeout is set at 90 minutes and that we run only one trial on each bug. In previous experiments, GenProg and PAR set time outs at 12 hours, and run 10 trials in parallel for each bug [29], [25]. We can expect greater success if we increase the number of trials. However, our results are consistent with a recent study, demonstrating that GenProg produces correct patches for 4 of 357 bugs in Defects4J with a 2 hour timeout and single trial per bug [14]. In sum, the results show that our approach substantially outperforms PAR and GenProg in terms of number of bugs correctly fixed.

RQ2: Case Studies. Figure 4 shows that the bugs that PAR and GenProg correctly fix are a subset of those that our approach correctly fixes. There are 18 bugs that our approach can fix that PAR and GenProg do not. We present observations on this in the form of illustrative case studies.

Lack of Mutation Operators. In many cases, PAR's mutation operators/templates are inadequate for fixing these bugs in the same way that the developer did. For example, consider the human-produced fix for Commons Math Version 5:

```
if(real == 0.0 && imaginary == 0.0){
- return NaN;
+ return INF;
}
```

Here, the human replaced one `return` statement with another. PAR has no mutation operator for this, while our approach has the *replace statement* operator adopted from GenProg, which helped generate this fix. Note that GenProg timed out on this bug, and thus did not fix the bug in our experiments.

Timeout. Even when the previous techniques possess the necessary mutation operators to potentially fix the bugs in the same way the developers did, in several cases they timed out before finding the fixes. For example, consider the developer-produced fix for Closure Compiler version 14:

```
for(Node finallyNode :
    cfa.finallyMap.get(parent)){
-    cfa.createEdge(fromNode, Branch.UNCOND,
    finallyNode)
+    cfa.createEdge(fromNode, Branch.ON_EX,
    finallyNode);
}
```

The developer replaced the method call parameter `Branch.UNCOND` with another parameter, `Branch.ON_EX`. PAR includes potentially appropriate templates, such as change method call name or replace parameter for method call. There are thus many possibilities for PAR to generate fix candidates for this buggy statement. However, even if PAR can generate the correct fix candidate among the pool of possible fix candidates, the correct fix candidate was not evaluated, as PAR timed out while evaluating other, incorrect candidates. We leave a more extensive study with longer timeouts and more random trials to future work.

Plausible vs Correct Fixes. Automated program repair techniques can generate both plausible and correct patches. A plausible patch leads the patched program to pass all test cases, but does not necessarily correspond to a true fix, consistent with the underlying specification and developer intent. A correct fix, on the other hand, is the one that correctly fixes the semantics of the buggy program. For example, consider the following code, including a plausible patch generated by GenProg for Math version 85:

```
//Fix by human and our approach: change
    condition to fa * fb > 0.0
  if (fa * fb >= 0.0) {
  //Plausible fix by GenProg
-  throw new ConvergenceException("...")
  }
```

GenProg's plausible patch simple deletes the `throw` statement. This fix makes the program pass all the given test cases, at least in part because the test cases do not truly check the underlying behavior. However, as compared to the human fix for the same bug, this fix is unlikely to correspond to developer intent or the underlying program specification. Additionally, the deletion of `throw` statements rarely happens in historical practice. A more correct fix for this bug changes the arithmetic

operator so that the exception is thrown in a correct manner that indeed satisfies the desired behaviour of the program; this is shown in the comment in the snippet, above the if condition.

In our approach, the *delete statement* mutation operator adopted from GenProg and the *change infix (arithmetic) expression* operator adopted from mutation testing both lead to the generation of a plausible patch: one similar to GenProg's, and the other similar to the human fix. However, partially due to the guidance provided by historical bug fixes, we avoid the plausible but incorrect patch and correctly choose the correct patch since the historical bug fix patterns suggest that changing an arithmetic happens more frequently in bug fixing practice. We also note that PAR does not generate any patch for this bug. Although PAR has the *expression replacer* operator which replace an *if* condition with another condition collected from the same scope, this operator does not help PAR generate patches for this bug since there is no correct condition appearing elsewhere in the same scope (same file).

Unfixed bugs in common. We observe that a common reason for why our approach, PAR and GenProg cannot fix bugs is a lack of ingredients that help synthesize the fix. For example, consider the human repair for Closure Compiler version 42:

```
+ if(loopNode.isForEach()){
+    errorReporter.error("unsupported...",
    sourceName, loopNode.getLineno(),"", 0);
+    return newNode(...);
}
```

The developer added an entire `if` statement to fix the bug. At first sight, the bug may be fixable by the program repair techniques in the same way as the developer did, if the same `if` statement appears elsewhere in the search space. However, it is indeed not the case. Thus, the three approaches failed to generate fixes for this bug.

RQ3: Average Amount of Time to Correct Fixes. In this research question, we report the average amount of time that our approach, GenProg, and PAR need in order to generate the correct fixes. GenProg requires less than 10 minutes to produce the fix for the one bug that it can correctly fix. PAR requires on average 10 minutes to generate correct fixes for the 4 bugs that it successfully fixes. Our approach needs on average 20 minutes to generate correct fixes for each of the 23 bugs.

This indicates that PAR and GenProg are still efficient and effective for a certain class of bugs. For example, bugs that have a small search space to be traversed to find correct fixes could be quickly fixed by PAR or GenProg. Our approach, on the other hand, is resilient to many classes of bugs with the help of both the mutation operators and the guidance of historical bug fix data. Note, however, that although our technique takes longer than the baselines, 20 minutes is still well within the range of a suitably efficient technique. Also, the average time is computed over the time needed by our approach to fix the more difficult bugs that cannot be fixed by PAR and GenProg even within 90 minutes (timeout cases). The key to good efficiency of our approach is that we generate a diverse set of possible fix candidates, and then use historical data to help pick the likely good fix candidates and test them against only the failed test cases, which originally make the buggy program fail. Thus, we do not waste too much time on evaluating nonsensical candidates. However, we do depend on the developer to assess the final patches for suitability with respect to the initially passing test cases.

RQ4: Rankings of Correct Fixes among Recommended Solutions. In this research question, we report the rankings of the correct fixes among the possible solutions that our approach presents to the developer. Recall that for each bug, we attempt to generate 10 possible solutions. We investigate two criteria for ranking possible solutions: *time* in which the fixes are produced, and *edit frequency* in the historical database.

Using time, the correct fixes are ranked number one for 13 out of the 23 bugs that we can produce correct fixes. We note that there are 6 bugs that we can only generate one solution for each bug and this solution is indeed the correct fix of the bug. For the remaining 10 bugs, each bug has correct fix ranked from 3 to 7 among the 10 possible solutions presented to the developer. Using frequency, there are 11 bugs that have correct fixes ranked number one. The remaining 12 bugs have correct fixes ranked from 2 to 10, among the 10 possible solutions presented to the developer.

These results suggest that ranking the correct fixes among possible solutions by either time or frequency is acceptable.

E. Threats to Validity

We consider three types of threats to validity: internal, external and construct validity:

Threats to Internal Validity. Threats to internal validity relate to errors in our implementation and experiments. We use a publicly available implementation of GenProg for Java programs. That implementation of GenProg is not written by the authors of GenProg, and thus it is possible that the implementation does not match all details specified in the original paper. Similarly, we reimplement PAR following the details provided in its paper. There could be bugs in the implementation that we are not aware of. To mitigate these threats, we have rechecked our implementation and experiments, fixed errors that we have found, and released our prototype for assessment by and comment from the community.

Threats to External Validity. Threats to external validity correspond to the generalizability of our findings. We perform our experiment on a dataset of 90 bugs from five Java projects; although we filter heuristically, this dataset is independently created and curated, mitigating the risk that our technique overfits to it. Still, they may not fully represent all real-world bugs. We plan to experiment on a larger dataset in the future.

Threats to Construct Validity. Threats to construct validity correspond to the suitability of our evaluation metrics. We consider a patch is correct if it passes all the test cases and qualitatively semantically matches human-made fix. To assess the second criterion, the first author manually checks the patches generated by the APR techniques. The APR techniques often generate patches that are syntactically the same as the human-made fix, but not for all cases. The first author has exercised caution when checking the patches, but it is possible that there are mistakes that we are not aware of. We plan to do a human study on the patches that the three APR techniques produce to better assess the quality of the patches.

IV. Related Work

Automated program repair has been the subject of considerable recent attention in the software engineering research community, and a large number of projects concern some form of repair. In the interest of brevity, we focus on indicative and closely-related techniques.

APR targeting general bugs: GenProg [28], [29] uses a Genetic Programming technique to evolve patches to a buggy input program, searching for candidates that cause the input program to pass all given test cases (both initially passing and initially failing). Other techniques using randomized search for patch generation have also been proposed [5], [9], [41], [50]. Debroy and Wong's efforts notably also reuse mutation operators to create patches, but do not reference edit history in evaluating candidate solutions [13]. Our work is importantly different because it uses the program history to inform fitness computation and candidate selection, with a long-term goal of creating more natural repairs and repairs that are of higher quality because of their resemblance to previous repairs. Pattern-based Automatic program Repair (PAR), which uses bug fix templates manually learned from existing human-written patches [25], is closely conceptually related to our own. PAR uses a similar randomized technique to apply these templates to a buggy program. The above mentioned approaches, including ours, make use of an important hypothesis that new code (e.g., bug fixes) can often be reconstructed from fragments of code that already exist in the code base. Barr et al. [7] empirically validate this hypothesis, showing that changes are 43% graftable from the exact version of program being changed. Another closely related work, that is developed in parallel with our work, is the work by Long and Rinard, named Prophet [32]. In that paper, the authors also leverage history information to fix bugs. Our work is different from theirs in several aspects: (1) Prophet can only fix bugs that involve one line bugs, while our approach can solve bugs that require multi-line changes, (2) We use bug-fix information from hundreds of open source projects, while prophet only uses eight, (3) We use a graph based representation which is a generic representation as compared to the ad-hoc set of features that are used by Prophet, (4) Prophet works on C program, our approach works on Java program, (5) Prophet successfully fixes 14 bugs, our approach successfully fixes 23 bugs, and (6) Our approach is more efficient than Prophet; only 4 of the bugs can be fixed by Prophet in 20 minutes or less (which is the average time needed by our approach to fix bugs). Unfortunately, an empirical comparison between our approach and Prophet cannot be made since we support different programming languages.

By contrast with search-based heuristic approaches, semantic approaches borrow ideas from program synthesis to construct bug-fixing patches. Techniques in this class include SemFix [36], a repair tool using semantic analysis such as symbolic execution, constraint solving and program synthesis. SemFix leverages test cases as implicit program specification to guide the patch synthesis process. More recently, DirectFix [34] extends this approach by targeting simplicity of generated patches, using MaxSAT constraint solving and component-based program synthesis. SearchRepair [24] uses SMT-solver-informed semantic code search and lightweight analyses to construct high-quality patches at a higher granularity level than previous techniques.

APR targeting specific bugs: Our approach targets general defect repair in Java. By contrast, a number of techniques target particular defect types or classes. For example, Perkins *et al.* propose ClearView, targeting security errors in binary programs [40]. Jin *et al.* present AFix, which uses static analysis to automatically repair single-variable atomicity violations [21]. Carbin *et al.* detect and fix infinite loop errors [11]; Smirnov *et al.* target fixing buffer overflow related errors [46]; Sidiroglou *et. al* propose an architecture to repair flaws that are exploited by zero-day worms [45]; Novark *et al.* present Exterminator - a system that automatically fix memory errors, including buffer overflows and dangling pointers [37]; Coker and Hafiz target integer vulnerabilities in C [12]. We leave an assessment of the particular defect types to which our approach most naturally applies to future work.

V. Conclusion and Future Work

Bug fixing is a difficult task that often takes much time and resources. To help developers fix bugs, researchers have proposed automated program repair (APR) techniques. Unfortunately, existing techniques are often not effective or efficient enough. They often unsuccessfully return correct patches despite running for a long period of time (e.g., more than 10 hours). In this work, we propose a *generic* and *efficient* APR technique that leverages information from *historical bug fixes*. Our solution takes as input a large set of repositories of software projects to create a knowledge base which is then leveraged to generate a ranked list of plausible bug fix patches given a buggy program and a set of test cases. It works on three phases: bug fix history extraction, bug fix history mining, and bug fix generation. We have evaluated the effectiveness of our proposed approach on a dataset of 90 bugs from five Java programs, and compared its effectiveness against two other generic generate-and-validate and test-case-driven APR techniques that work on Java programs. Our experiment results highlight that our approach can fix 23 bugs correctly, which are many more than the bugs that can be fixed by GenProg and PAR. On average, our solution can fix the 23 bugs within 20 minutes. These highlight the superior performance of our proposed approach in terms of effectiveness and efficiency as compared to existing generic APR solutions that can fix multi-line bugs in Java programs.

In the future, we plan to improve the effectiveness and efficiency of our solution further. We plan to do so by designing better ways to traverse the search space of potential patches. We also plan to incorporate data from not only 3,000 bug fixes but even a larger number taken from even many more programs. Moreover, we plan to design an adaptive APR strategy that can vary the way it generates patches depending on characteristics of a bug.

Acknowledgment

We would like to thank Just *et al.* for releasing the Defect4J dataset [22] and Monperrus *et al.* for releasing their implementation of GenProg[8] which works for Java programs [14].

[8]https://libraries.io/github/SpoonLabs/astor

REFERENCES

[1] Github archive. [Online]. Available: https://githubarchive.org/

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007, pp. 89–98.

[3] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Workshop on Eclipse Technology eXchange*, San Diego, California, 2005, pp. 35–39.

[4] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, Honolulu, HI, USA, 2011, pp. 1–10.

[5] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Congress on Evolutionary Computation*, 2008, pp. 162–168.

[6] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," in *Proceedings of the second international conference on genetic algorithms*, 1987, pp. 14–21.

[7] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 306–317. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635898

[8] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Amsterdam, The Netherlands, August 2009, pp. 121–130.

[9] J. S. Bradbury and K. Jalbert, "Automatic repair of concurrency bugs," in *International Symposium on Search Based Software Engineering (SSBSE) fast abstract*, M. Di Penta, S. Poulding, L. Briand, and J. Clark, Eds., Benevento, Italy, Sep. 2010.

[10] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," University of Cambridge, Judge Business School, Tech. Rep., 2013.

[11] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with jolt," in *ECOOP 2011–Object-Oriented Programming*. Springer, 2011, pp. 609–633.

[12] Z. Coker and M. Hafiz, "Program transformations to fix C integers," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, 2013, pp. 792–801.

[13] V. Debroy and W. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *International Conference on Software Testing, Verification, and Validation*, Paris, France, 2010, pp. 65–74.

[14] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, "Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset," *ArXiv e-prints*, May 2015.

[15] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[16] M. Harman, "The current state and future of search based software engineering," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2007, pp. 342–357.

[17] ——, "Automated patching techniques: the fix is in: technical perspective," *Communications of the ACM*, vol. 53, no. 5, pp. 108–108, 2010.

[18] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 2013, pp. 121–130.

[19] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007, pp. 34–43.

[20] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *IN ICSE*, 2007, pp. 96–105.

[21] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 389–400. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993544

[22] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 2014, pp. 437–440.

[23] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 351–360.

[24] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Proceedings of the 30th IEEE/ACM International Conference On Automated Software Engineering (ASE)*, Lincoln, NE, USA, November 2015.

[25] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 802–811.

[26] J. R. Koza, "Genetic programming: On the programming of computers by means of natural selection, 1992," *See http://miriad. lip6. fr/microbes Modeling Adaptive Multi-Agent Systems Inspired by Developmental Biology*, vol. 229, 1992.

[27] X.-B. D. Le, T.-D. B. Le, and D. Lo, "Should fixing these failures be delegated to automated program repair?" in *26th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

[28] 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 *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 3–13.

[29] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[30] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Genetic and Evolutionary Computation Conference*, T. Soule and J. H. Moore, Eds. ACM, 2012, pp. 959–966.

[31] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, USA, 2003, pp. 141–154.

[32] F. Long and M. Rinard, "Prophet: Automatic patch generation via learning from successful patches," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016 (to appear)*, 2016.

[33] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 492–495.

[34] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 448–458. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818811

[35] 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*. ACM, 2014, pp. 234–242.

[36] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 772–781.

[37] G. Novark, E. D. Berger, and B. G. Zorn, "Exterminator: Automatically correcting memory errors with high probability," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250736

[38] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[39] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "The class-level mutants of mujava," in *Proceedings of the 2006 International Workshop on Automation of Software Test*, ser. AST '06. New York, NY, USA: ACM, 2006, pp. 78–84. [Online]. Available: http://doi.acm.org/10.1145/1138929.1138945

[40] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 87–102. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629585

[41] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair." in *ICSE*, 2014, pp. 254–265.

[42] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," 2015.

[43] Z. Qi, F. Long, S. Achour, and M. C. 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, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, 2015, pp. 24–36.

[44] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.

[45] S. Sidiroglou and A. D. Keromytis, "Countering network worms through automatic patch generation," 2003.

[46] A. Smirnov and T.-c. Chiueh, "Dira: Automatic detection, identification and repair of control-hijacking attacks." in *NDSS*. Citeseer, 2005.

[47] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, Sep. 2015.

[48] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 011, 2002.

[49] J. Torán, "On the hardness of graph isomorphism," *SIAM J. Comput.*, vol. 33, no. 5, pp. 1093–1108, May 2004. [Online]. Available: http://dx.doi.org/10.1137/S009753970241096X

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

[51] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *International Workshop on Mining Software Repositories*, Minneapolis, MN, USA, 2007.

[52] X. Yan and J. Han, "Closegraph: mining closed frequent graph patterns," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 286–295.