Chapter 19
An Evolutionary System for Better Automatic Software Repair

Yuan Yuan and Wolfgang Banzhaf

19.1 Introduction

Automatic software repair [13, 39, 49] aims to fix bugs in software automatically, generally relying on a specification. When a test suite is considered as the specification, the paradigm is called *test-suite based repair* [39]. The test suite should contain at least one negative (i.e., initially failing) test that triggers the bug to be fixed and a number of positive (i.e., initially passing) tests that define the expected program behavior. In terms of test-suite based repair, a bug is regarded to be *fixed* or *repaired*, if a created patch makes the entire test suite pass. Such a patch is referred to as a *test-adequate patch* [33] or a *plausible patch* [44].

Evolutionary repair approaches [49] are a popular category of techniques for test-suite based repair. These approaches determine a search space potentially containing correct patches, then use evolutionary computation (EC) techniques, particularly genetic programming (GP) [2, 4, 21], to explore that search space. A major characteristic of evolutionary repair approaches is that they have high potential to fix multi-location bugs, since GP can manipulate multiple likely faulty locations at a time. However, GenProg [12, 25, 27, 51], the most well-known approach of this kind, does not fulfill the potential in multi-location bug fixing according to large-scale empirical studies [33, 44], partly due to the search ability of its underlying GP [42,44,57]. To tackle this issue, our previous work introduced ARJA [57], which uses a novel multi-objective GP approach with better search ability to explore the search space. Although ARJA has achieved much improved performance and also

Yuan Yuan
BEACON Center for the Study of Evolution in Action and Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA e-mail: yyuan@cse.msu.edu

Wolfgang Banzhaf
BEACON Center for the Study of Evolution in Action and Department of Computer Science and and Engineering, Michigan State University, East Lansing, MI, USA e-mail: banzhaf@msu.edu

demonstrated its strength in multi-location repair, major challenges [26] still remain for evolutionary software repair.

The first challenge is how to construct a reasonable search space that is more likely to contain correct patches. In this respect, GenProg and ARJA exploit the statement-level *redundancy assumption* [36] (also called *plastic surgery hypothesis* [3]). That is, they only conduct statement-level changes and use existing statements in the buggy program for replacement or insertion. The problem here is that fix statements randomly excerpted from somewhere in the current buggy program may have little pertinence to the likely-buggy statement to be manipulated. Due to this problem, GenProg usually generates patches overfitting the test suite or even fails to fix a bug. To relieve the issue, Kim *et al.* [20] proposed PAR, which exploits *repair templates* to produce program variants. Each template specifies one type of program transformation and is derived from common fix patterns (e.g., adding a null-pointer checker for an object reference) manually learned from human-written patches. Compared to GenProg, PAR usually works in a more promising search space, since the program transformations performed by PAR are more targeted. Nevertheless, as can be inferred from the results in [57], the redundancy-based approaches can really fix some bugs that cannot be fixed by typical template-based approaches (e.g., PAR and ELIXIR [46]) which implies that combining the redundancy assumption and repair templates to generate fix statements could further improve repair effectiveness.

The second challenge is how to design a search algorithm that can navigate the search space more effectively. The combination of the statement-level redundancy assumption and repair templates will lead to a much larger search space, thereby making this challenge more serious. Recent studies [42, 57] have indicated that compared to using GenProg's patch representation, using a lower-granularity patch representation that decouples the partial information of an edit can significantly improve the search ability of GP in bug repair. However such representations are specially designed for statement-level edits and cannot be directly used for template-based edits (usually occurring at the expression level). Besides the patch representation, the fitness function is another important factor that influences the search ability of GP. In existing evolutionary repair approaches, the fitness function is generally defined based on how many test cases a patched program passes. However this kind of fitness function can only provide a binary signal (i.e, passed or failed) for a test case and cannot measure how close a modified program is to pass a test case. In consequence, there may be a large number of plateaus in the search space [11, 26, 44], thereby trapping GP.

The third challenge is how to alleviate patch overfitting [47]. Evolutionary repair approaches can usually find a number of plausible patches within a computing budget. But most of these patches may be incorrect in general, by just overfitting the given test suite. To pick correct patches more easily, it is necessary to include a post-processing step for these approaches, which can filter out incorrect patches (i.e., *overfit detection*) or rank the plausible patches found (i.e., *patch ranking*). However, almost all existing evolutionary repair systems, including GenProg, PAR, and ARJA, do not implement such a step.

In this chapter, we describe ARJA-e, a new evolutionary repair system for Java programs that aims to address the above three challenges. To determine a search space that is more likely to contain correct patches, ARJA-e combines two sources of fix ingredients (i.e., the statement-level redundancy assumption and repair templates) with contextual analysis based search space reduction, thereby leveraging their complementary strengths. To encode patches in GP more properly, ARJA-e unifies the edits at different granularities into statement-level edits, and then uses a new lower-granularity patch representation that is characterized by the decoupling of statements for replacement and statements for insertion. Furthermore, ARJA-e uses a finer-grained fitness function that can make full use of semantic information contained in the test suite, which is expected to better guide the search of GP. To alleviate patch overfitting, ARJA-e includes a post-processing tool that can serve the purposes of overfit detection and patch ranking.

19.2 Background and Motivation

19.2.1 Related Work

Our system belongs to the class of evolutionary repair approaches which explore a repair search space using evolutionary algorithms. GenProg [25, 27], PAR [20], GenProg with anti-patterns [48] and ARJA [57] all fall into this category. Their basic ideas have been described in Section 19.1. ARJA-e organically combines the characteristic components of all these approaches, making it distinctly different from any of them. Several approaches employ other kinds of search algorithms, instead of EAs, to traverse GenProg's search space (e.g., RSRepair [43] uses random search and AE [50] uses an adaptive search strategy).

Inspired by the idea of using templates [20], some repair approaches (e.g., SPR [31] and ELIXIR [46]) employ a set of richer templates (or code transformations) that are defined manually. Genesis [30] aims to automatically infer such code transformations from successful patches. Cardumen [35] mines repair templates from the program under repair. Similar to these approaches, ARJA-e uses templates extended and enhanced from those in PAR.

Beyond the current buggy program and its associated test suite, some approaches exploit other information to help the repair process. HDRepair [24] uses mined historical bug fixes to guide its random search. ACS [55] uses the information of javadoc comments to rank variables. SearchRepair [19] and ssFix [53] both use existing code from an external code database to find potential repairs.

A number of existing approaches infer semantic specifications from the test cases and then use program synthesis to generate a repair that satisfies the inferred specifications. These are usually categorized as semantics-based approaches. SemFix [41] is a pioneer in this category. Other typical approaches of this kind include DirectFix [37], QLOSE [8], Angelix [38], Nopol [56], JFix [22] and S3 [23]. Recently, machine learning techniques have been used in software repair. Prophet [32] uses a probabilistic model to rank the candidate patches over the search space of SPR. DeepFix [14] uses deep learning to fix common programming errors.

19.2.2 Motivating Examples

In this subsection, we take real bugs as examples to illustrate the key insights motivating the design of ARJA-e.

Fig. 19.1 shows the human-written patch for bug Math85 from the Defects4J [18] dataset. To correctly fix this bug, only a slight modification is required (i.e., change `>=` to `>`), as shown in Fig. 19.1. However, redundancy-based approaches (e.g., GenProg [25, 27], RSRepair [43] and AE [50]) usually cannot find a correct patch for this bug since the fix statement used for replacement (i.e., `if (fa * fb > 0){...}`) or semantically equivalent ones do not happen to appear elsewhere in the buggy program. In contrast, some template-based approaches (e.g., jMutRepair [10, 34] and ELIXIR [46]) are very likely to fix the bug correctly since changing of infix boolean operators is a specified repair action in such approaches. In addition, GenProg can easily overfit the given test suite [44] by deleting the whole buggy `if` statement: `if (fa * fb >= 0){...}`), leading to a plausible but incorrect patch.

```
public static double[] bracket (...) { ...
-    if (fa * fb >= 0.0) {
+    if (fa * fb > 0.0) {
       throw new ConvergenceException (...); }   ... }
```

Fig. 19.1: The human-written patch for bug Math85.

Fig. 19.2 shows the human-written patch for bug Math39 from Defects4J. To correctly repair the bug, an `if` statement with relatively complex control logic should be inserted before the buggy code, as shown in Fig. 19.2. However, for approaches only based on repair templates, the bug is hard to fix correctly, because this fix generally does not belong to a common fix pattern and is difficult to be encoded with templates. In contrast, approaches that exploit the redundancy assumption can potentially find a correct patch for the bug, because the following `if` statement

```
if ((forward && (stepStart + stepSize > t)) || ((!forward) && (stepStart + stepSize <
    t))) { stepSize = t - stepStart; }
```

happens to be in the buggy program elsewhere, which is semantically equivalent to the one inserted by human developers.

From the above examples, it can be seen that redundancy- and template-based approaches potentially have complementary strengths in bug fixing. We aim to com-

```
public void integrate (...) throws ... { ...
+ if (forward) {
+    if ( stepStart + stepSize >= t) { stepSize = t - stepStart ; }
+ } else {
+    if ( stepStart + stepSize <= t) { stepSize = t - stepStart ; } }
 ... }
```

Fig. 19.2: The human-written patch for bug Math39.

bine both statement-level redundancy assumption and repair templates, to generate potential fix ingredients. Such a combination will lead to a much larger search space, posing a great challenge to the search algorithm. So we will also introduce several strategies to properly reduce the search space and enhance the search algorithm with a new lower-granularity patch representation.

19.3 Overview of ARJA-e

The input of ARJA-e is a buggy program associated with a JUnit test suite. ARJA-e basically aims to make all these test cases pass by modifying the buggy program. First, we use the fault localization technique called Ochiai [5] to select n likely-buggy statements (LBSs) with the highest suspiciousness. For the j-th LBS, we determine three sets denoted by R_j, I_j and O_j. R_j is the set of statements that can be used to replace the LBS, I_j is the set of statements that can be used for insertion before the LBS, and O_j is a subset of three operation types: “delete”, “replace” and “insert”. To find simpler patches, we uses a multi-objective GP to explore the determined search space, with the guidance of a finer-grained fitness function. Through evolutionary search, ARJA-e can usually find a number of plausible patches. However, many of these patches may overfit the test suite and would thereby be not correct. To alleviate the patch overfitting issue, we develop a post-processing tool which can identify overfitting patches or rank the plausible patches found by ARJA-e.

In the following sections, we will detail how to shape the search space (i.e., determine R_j, I_j and O_j, see Section 19.4), how to conduct multi-objective search (see Section 19.5) and how to alleviate patch overfitting (see Section 19.6).

19.4 Shaping the Search Space

19.4.1 Exploiting the Statement-Level Redundancy Assumption

For each LBS selected, we first collect the statements within the package where the LBS resides, and then ignore those statements that are not in-scope at the destination of the LBS or violates the complier constraints. For each of the remaining statements (denoted by s), we further check the program context. Our insight is that if replacing the LBS with s is a promising manipulation, s should generally exhibit a certain *similarity* to the LBS; and if it is potentially useful to insert s before the LBS, s should generally have a certain *relevance* to the context surrounding the LBS. In the following, we describe how to quantify such similarity and relevance.

Suppose V_s and V_{LBS} are the sets of variables (including local variables and fields) used by s and the LBS respectively. We define the similarity between s and the LBS as the Jaccard similarity coefficient between sets V_s and V_{LBS}:

$$sim(s,\text{LBS}) = \frac{|V_{\text{LBS}} \cap V_s|}{|V_{\text{LBS}} \cup V_s|} \tag{19.1}$$

Note that when collecting fields used by a statement, we also consider the fields accessed by invoking the methods in the current class.

In the method where the LBS resides, suppose V_{bef} and V_{aft} are the sets of variables used by k statements before and after the LBS, respectively, where k is set to 5 by default. We define the relevance of s to the context of LBS as follows:

$$rel(s,\text{LBS}) = \frac{1}{2}\Big(\frac{|V_s \cap V_{\text{bef}}|}{|V_s|} + \frac{|V_s \cap V_{\text{aft}}|}{|V_s|}\Big) \tag{19.2}$$

Eq. (19.2) indeed averages the percentages of the variables in V_s that are covered by V_{bef} and V_{aft}.

If $|V_{\text{LBS}} \cup V_s| = 0$, $sim(s,\text{LBS})$ is set to 1, and if $|V_s| = 0$, $rel(s,\text{LBS})$ is set to 0. So $sim(s,\text{LBS}) \in [0,1]$ and $rel(s,\text{LBS}) \in [0,1]$. Only when $sim(s,\text{LBS}) > \beta_{\text{sim}}$ can s be put into R_j (i.e., the set of candidate statements for replacement), and only when $rel(s,\text{LBS}) > \beta_{\text{rel}}$ can s be put into I_j (i.e., the set of candidate statements for insertion), where β_{sim} and β_{rel} are predetermined threshold parameters.

19.4.2 Exploiting Repair Templates

In ARJA-e, we also use 7 repair templates to manipulate the LBS, which are mainly extended from templates used in PAR. These templates are described in Table 19.1.

Template ER replaces an abstract syntax tree (AST) node element in a LBS with another compatible one. Table 19.2 lists the elements that can be replaced and also shows alternative replacers for each kind of elements. This template generalizes

Table 19.1: The Description of Repair Templates Used in this Study

No.	Template Name	Description
1	Null Pointer Checker (NPC)	Add an `if` statement before a LBS to check whether any object reference in this LBS is `null`
2	Range Checker (RC)	Add an `if` statement before a LBS to check whether any array or list element access in this LBS exceeds the upper or lower bound.
3	Cast Checker (CC)	Add an `if` statement before a LBS to assure that the variable or expression to be converted in this LBS is an instance of casting type.
4	Divide-by-Zero Checker (DC)	Add an `if` statement before a LBS to check whether any divisor in this LBS is 0.
5	Method Parameter Adjuster (MPA)	Add, remove or reorder the method parameters in a LBS if this method has overloaded methods.
6	Boolean Expression Adder or Remover (BEAR)	For a condition branch (e.g., `if`), add a term to its predicate (with `&&` or `||`), or remove a term from its predicate
7	Element Replacer (ER)	Replace an AST node element (e.g., variable or method name) in a LBS with another one with compatible type

the templates "Parameter Replacer" and "Method Replacer" used in PAR. Several replacement rules are inspired by recent template-based approaches (e.g., replacing a primitive type with widened type follows ELIXIR [46] and replacing `x` with `f(x)` follows the transformation schema in REFAZER [45]).

Table 19.2: List of Replacement Rules for Different Elements

Element	Format	Replacer
Variable	x	(i) The visible fields or local variables with compatible type (ii) A compatible method invocation in the form of f() or f(x)
Field access	e.g., this.a	The same as above
Qualified name	a.b	The same as above
Method name	f (...)	The name of another visible method with compatible parameter and return types
Primitive type	e.g., int	A widened type, e.g., float to double
Boolean literal	true or false	The opposite boolean value
Number literal	e.g., 1 or 0.5	Another number literal located in the same method
Infix operators	e.g., + or >	A compatible infix operator, e.g., + to −, > to >=
Prefix/Postfix operators	e.g., ++	The opposite prefix/postfix operator, e.g., ++ to −−
Assignment operators	e.g., +=	The opposite assignment operator, e.g., += to −=
Conditional expression	a ? b : c	b or c

Unlike PAR which applies templates on-the-fly (i.e., during the evolutionary process), ARJA-e executes the above 7 repair templates in an offline manner. Specifically, we perform all the possible transformations defined by the templates for each LBS before searching for patches. Then each LBS can derive a number of new statements, each of which can either replace the LBS or be inserted before it. So various template-based edits (usually at the expression-level) are abstracted into two types of statement-level edits (i.e., replacement and insertion). These statements for replacement and insertion are added into R_j and I_j respectively. For the LBS `a.callX()`, Fig. 19.3 illustrates the way to exploit the templates in ARJA-e. Note that we do not consider similarity and relevance as in Section 19.4.1 since the statements generated by templates are highly targeted. Moreover, we only apply a template to a single AST node at a time to avoid combinatorial explosion. For example, we do not simultaneously modify `a` and `callX` in `a.callX()` using the template ER.

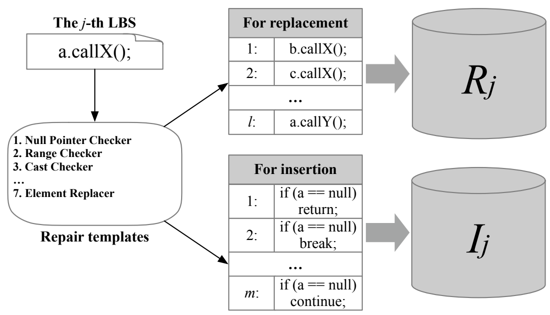


Fig. 19.3: Illustration of the offline execution of templates.

19.4.3 Initialization of Operation Types

The deletion operation should be executed carefully because it can easily lead to the following two problems: It can (i) cause a compiler error of the modified code; or (ii) generate overfitting patches [44]. To address the first problem, we use the two rules defined in [57], that is, if a LBS is a variable declaration statement or a `return`/`throw` statement which is the last statement of a method not declared `void`, we disable the deletion operation for this LBS. To address the second problem, we use the 5 anti-delete patterns defined in [48]. If a LBS follows any of these patterns, we ignore the deletion operation. For example, according to one of the anti-delete patterns, if a LBS is a control statement (e.g., `if` statement or loops), deletion of the LBS is disallowed.

19.5 Multi-Objective Evolution of Patches

19.5.1 Patch Representation

To encode a patch as a genome in GP, we first number the LBSs and the elements in R_j, I_j and O_j respectively, starting from 1, where $j \in \{1,2,\ldots,n\}$. All the IDs are fixed throughout the search.

A solution (i.e., a patch) to the program repair problem is encoded as $\mathbf{x} = (\mathbf{b},\mathbf{u},\mathbf{p},\mathbf{q})$, which contains four different parts each being a vector of size n. In the solution $\mathbf{x}$, $b_j \in \{0,1\}$ indicates whether the j-th LBS is to be edited or not; $u_j \in \{1,2,\ldots,|O_j|\}$ indicates the u_j-th operation type in O_j is used for the j-th LBS; $p_j \in \{1,2,\ldots,|R_j|\}$ means that if replace operation is used, the p_j-th statement in R_j will be selected to replace the j-th LBS; and $q_j \in \{1,2,\ldots,|I_j|\}$ means that if insert operation is used, the q_j-th statement in I_j will be inserted before the j-th LBS. Fig. 19.4 illustrates the new lower-granularity patch representation. Suppose the j-th LBS is `a.callX();` in this figure, then the edit on the j-th LBS is: replace `a.callX();` with `b.callX();`.

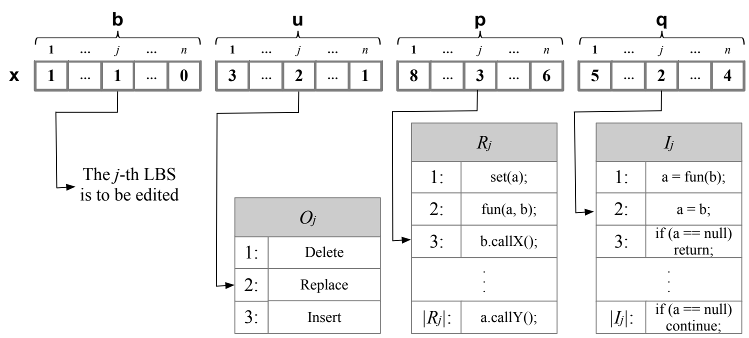


Fig. 19.4: Illustration of the new lower-granularity patch representation.

19.5.2 Finer-Grained Fitness Function

To evaluate the fitness of an individual $\mathbf{x}$, we still use a bi-objective function as in the original ARJA [57]. The first objective (i.e., $f_1(\mathbf{x})$) is the patch size, which is exactly the same as that in ARJA. The second objective (i.e., $f_2(\mathbf{x})$) is the weighted failure rate. Different from that in ARJA, we compute $f_2(\mathbf{x})$ through finer-grained analysis of test execution in this study, in order to provide smoother gradient for the genetic search to navigate the search space. Since our repair system targets Java, our implementation is based on the JUnit [7] framework. Specifically, we define a metric to measure the degree of violation for each assertion, which we call *assertion distance*. Suppose an assertion (denoted by e) asserting x and y are equal to within a positive δ: `assertEquals(`x, y, δ`)`, then the assertion distance $d(e)$ is computed as:

$$d(e) = \begin{cases} \nu(|x-y|-\delta), & |x-y| \geq \delta \\ 0, & |x-y| < \delta \end{cases} \tag{19.3}$$

Here, $\nu(x)$ is a normalizing function in $[0,1]$ and we use the one suggested in [1]: $\nu(x) = x/(x+1)$.

After executing a program variant $\mathbf{x}$ over a test case t, we can compute a metric $h(\mathbf{x},t) \in [0,1]$ to indicate how badly $\mathbf{x}$ fails the test case t by using the collected assertion distances. This metric is defined as follows:

$$h(\mathbf{x},t) = \frac{\sum_{e \in E(\mathbf{x},t)} d(e)}{|E(\mathbf{x},t)|} \tag{19.4}$$

where $E(\mathbf{x},t)$ is the set of executed assertions by $\mathbf{x}$ over t, and $d(e)$ is the assertion distance for the assertion e. Based on $h(\mathbf{x},t)$, $f_2(\mathbf{x})$ can be formulated as follows:

$$f_2(\mathbf{x}) = \frac{\sum_{t \in T_{pos}} h(\mathbf{x},t)}{|T_{pos}|} + w \times \left(\frac{\sum_{t \in T_{neg}} h(\mathbf{x},t)}{|T_{neg}|}\right) \tag{19.5}$$

where $w \in (0,1]$ is a parameter that can introduce a bias toward negative test cases.

19.5.3 Genetic Operators

Genetic operators, including crossover and mutation, are used to produce the offspring individuals in GP. Crossover is applied to each part of the patch representation separately, in order to inherit good genetic materials from parents. For all four parts, we employ the half uniform crossover (HUX) operator.

We apply a guided mutation to the information of a single selected LBS. To be specific, we first use roulette wheel selection to choose a LBS, where the j-th LBS is chosen with a probability of $susp_j / \sum_{j=1}^{n} susp_j$; suppose that the j-th LBS is finally selected, then we apply bit flip mutation to b_j and uniform mutation to u_j, p_j and q_j respectively. Fig. 19.5 illustrates the crossover and mutation operations, where only a single offspring is shown for brevity.

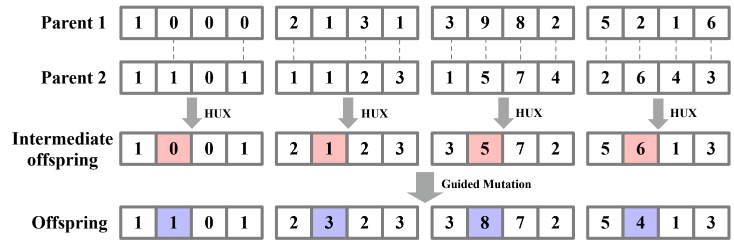


Fig. 19.5: Illustration of the crossover and mutation.

19.5.4 Multi-Objective Search

With the patch representation, fitness function and genetic operators designed above, any multi-objective evolutionary algorithm can serve the purpose of searching for patches. In this work, we basically employ NSGA-II [9] as the multi-objective search framework. To initialize the population, we combine the fault localization result and randomness: for the first part (i.e., $\mathbf{b}$), b_j is initialized to 1 with a probability of $susp_j \times \mu$, where $\mu \in (0,1)$ is a predefined parameter; and the remaining three parts (i.e., $\mathbf{u}$, $\mathbf{p}$, $\mathbf{q}$) are just initialized randomly. After population initialization, the search algorithm iterates over generations until the stopping criterion is satisfied.

19.6 Alleviating Patch Overfitting

19.6.1 Overfit Detection

For overfit detection, we take a buggy program, a set of positive test cases and a plausible patch as input, and determine whether or not this plausible patch is an overfitting patch. Our approach is based on the assumption that the buggy program will perform correctly on the test inputs encoded in the positive test cases.

Fig. 19.6 shows the overall process of this approach. First, given a plausible patch and a buggy program, we can localize the methods where the statements will be modified by the patch. Then we instrument the bytecode of these methods in the buggy program. For each method, the instrumentation is conducted at its entry point and all its possible exit points. At the entry point, we inject new bytecode to save the *input* of the method, including all the method parameters and the current object `this` (i.e., the object whose method is being called), into a file. At each exit point, we inject new bytecode to save the *output* of the method, including the return value, the current object `this` and the reference-type method parameters, into another file. Note that if a method to be instrumented is a static method, we just ignore the current object. To save the objects, we leverage the Java serialization technique.

This technique can convert object state to a byte stream that can be reverted back into a copy of the object.

With the instrumented buggy program, we run the positive test cases so that we can capture a number of input-output pairs for the localized methods. Suppose that there are K such pairs, denoted by a set $PA = \{(In_1, Out_1), (In_2, Out_2), \ldots, (In_K, Out_K)\}$. According to our assumption, all these input-output pairs should reflect the correct program behavior. In order to judge patch correctness, we will feed these inputs $In_1, In_2, \ldots, In_K$ into the corresponding methods in the patched program so as to see whether the correct outputs can be obtained. Specifically, for each input-out pair $(In_i, Out_i) \in PA$ collected previously, we deserialize In_i from the file and use the Java reflection technique to invoke the corresponding method in the instrumented patched program with the deserialized input In_i, so that we can collect the method output Out_i'. Lastly, we compare every Out_i' with the corresponding Out_i, and if there exists any difference, we identify the plausible patch as an overfitting patch that is incorrect.

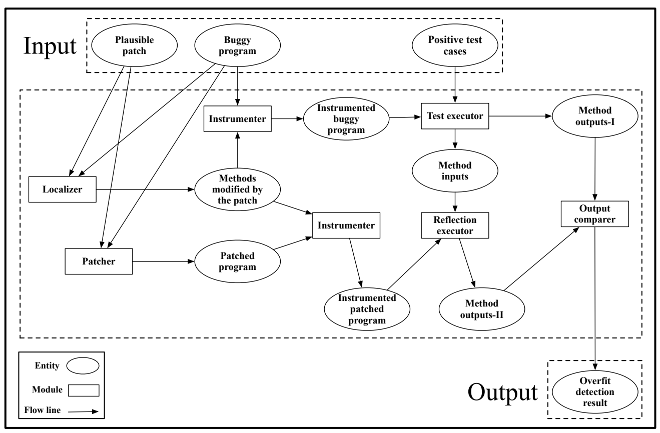


Fig. 19.6: The overview of our overfit detection approach.

19.6.2 Patch Ranking

ARJA-e can sometimes output more than one plausible patch (with the same patch size) for a bug. As a post-processing step, we design a heuristic procedure to rank these patches. For this ranking purpose, we first define three metrics for a patch. The first metric, denoted by $Susp$, represents the summation of the suspiciousness for the LBSs modified by the patch. The second metric, denoted by $Dist$, is based on our overfit detection approach. Recall that for the purpose of overfit detection, we only need to know whether there is a difference between Out_i and Out_i', where $i = 1, 2, \ldots, K$. Here we want to quantify such a difference. To do this, we deserialize Out_i and Out_i' and extract all primitive data and string data contained in the two outputs in a recursive way. Similar to the computing of assertion distance, we can easily compute the distance for each corresponding primitive/string data and normalize it to $[0, 1]$. Then *Dist* is calculated as the average of these normalized distances for all outputs. Before defining the third metric, we determine a preference relation of operation types in our system. We prefer the operation type that is generally less likely to bring in side effects, and the preference relation is: NPC/RC/CC/DC $\prec$ MPA $\prec$ ER $\prec$ BEAR $\prec$ SR/SI $\prec$ SD. Here SR and SI mean statement replacement and insertion based on the redundancy assumption respectively, and SI means statement deletion. The others are all template-based operations that can be referred to in Section 19.4.2. We assign a preference score for each operation type: SD is scored 1, SR and SI are scored 2, BEAR is scored 3 and so on. With these scores, the second metric for a patch, denoted by $Pref$, is defined as the sum of scores of operation types contained in the patch. For *Susp* and $Pref$, larger is better; whereas for *Dist*, smaller is better.

When comparing two patches in the ranking, *Susp*, *Dist* and $Pref$ are considered in sequence until the two patches can be distinguished. If all the three metrics cannot distinguish the two patches, the patch found earlier is ranked higher.

19.7 Experimental Design

19.7.1 Research Questions

We intend to answer the following research questions:

RQ1: How effective is ARJA-e compared to state-of-the-art repair systems on real bugs?

RQ2: Can ARJA-e fix bugs in a novel way compared to a human developer?

RQ3: How good is our overfit detection approach?

19.7.2 Dataset of Bugs

We perform the empirical evaluation on a database of real bugs, called Defects4J [18], which has been extensively used for evaluating Java repair systems [6, 33, 46, 53, 55, 57]. We consider four projects in Defects4J, namely Chart, Time, Lang and Math. Table 19.3 shows the descriptive statistics of the four projects. In total, there are 224 real bugs: 26 from Chart (C1–C26), 27 from Time (T1–T27), 65 from Lang (L1–L65) and 106 from Math (M1–M106).

Table 19.3: The descriptive statistics of Defects4J dataset

Project	ID	#Bugs	#JUnit Tests	Source KLoC	Test KLoC
Chart	C	26	2,205	96	50
Time	T	27	4,043	28	53
Lang	L	65	2,295	22	6
Math	M	106	5,246	85	19
Total		224	13,789	231	128

19.7.3 Parameter Setting

Table 19.4 shows the parameter setting for ARJA-e in the experiments. Note that crossover and mutation operators presented in Section 19.5.3 are always executed, so the probability (i.e., 1) is omitted in this table. Given the stochastic nature of ARJA-e, we execute 5 random trials in parallel for each bug. Each trial of ARJA-e is terminated after it reaches the maximum number of generations (i.e., 50) or its execution time exceeds one hour.

Table 19.4: The parameter setting for ARJA-e

Parameter	Description	Value
N	Population size	40
$\gamma_{\min}$	Threshold for the suspiciousness	0.1
$n_{\max}$	Maximum number of LBSs considered	60
β_{sim}	Threshold for similarity	0.3
β_{rel}	Threshold for relevance	0.3
w	Refer to Section 19.5.2	0.5

19.8 Results and Discussions

19.8.1 Performance Evaluation (RQ1)

To show the superiority of ARJA-e over the state of the art, we compare ARJA-e with 13 existing repair approaches in terms of the number of bugs fixed and correctly fixed. The 13 approaches are jGenProg [33, 34] (an implementation of GenProg for Java), xPAR (a reimplementation of PAR by Le *et al.* [24]), Nopol [56], HDRepair [24], ACS [55], ssFix [53], JAID [6], ELIXIR [46], ARJA [57], SimFix [17], CAPGEN [52], SOFIX [29] and SKETCHFIX [16], which include almost all published approaches that have ever been tested on Defects4J. Note that here we use a strict criterion for judging whether a bug is correctly fixed by ARJA-e, that is, a bug is regarded as being correctly fixed only when the plausible patch ranked first (using the procedure in Section 19.6.2) is correct.

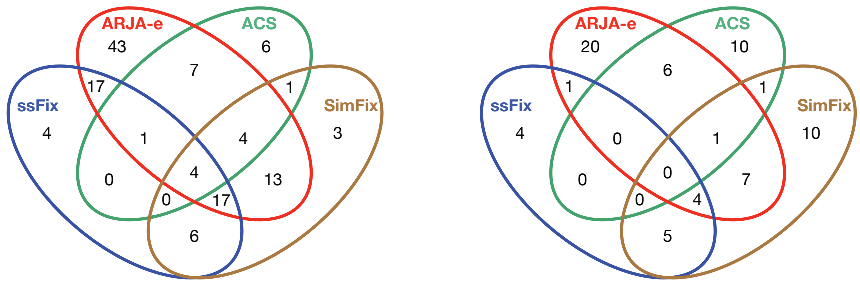
Table 19.5: Comparison with Existing Repair Tools in terms of the Number of Bugs Fixed and Correctly Fixed (Plausible/Correct). The Best Results are Shown in Bold

Project	ARJA-e	jGenProg	xPAR	Nopol	HDRepair[1]	ACS	ssFix
Chart	**18/7**	7/0	–/0	6/1	–/2	2/2	7/2
Lang	**28/9**	0/0	–/1	7/3	–/7	4/3	12/5
Math	**51/21**	18/5	–/2	21/1	–/6	16/12	26/7
Time	**9/2**	2/0	–/0	1/0	–/1	1/1	4/0
Total	**106/39**	27/5	–/3	35/5	–/16	23/18	49/14

Project	JAID	ELIXIR	ARJA[2]	SimFix	CAPGEN	SOFIX	SKETCHFIX
Chart	4/2	7/4	9/3	8/4	–/4	–/5	8/6
Lang	8/1	12/8	17/4	13/**9**	–/5	–/4	4/1
Math	8/1	19/12	29/10	26/14	–/12	–/13	8/7
Time	0/0	3/**2**	4/1	1/1	–/0	–/1	1/1
Total	20/4	41/26	59/18	48/28	–/21	–/23	21/15

"–" means the data is not available since it is not reported by the original authors.
[1] HDRepair generated correct patches for 16 bugs, but only 10 of them were ranked first.
[2] In ARJA, a bug is regarded as being fixed correctly if one of its plausible patches is identified as correct.

Table 19.5 shows the comparison results. From Table 19.5, we can see that ARJA-e outperforms all other approaches in terms of the number of fixed bugs and correctly fixed bugs. We further compare ARJA-e with ACS, ssFix and SimFix by analyzing the overlaps among their repair results. ACS, ssFix and SimFix are selected because they show prominent performance among the 13 compared approaches and the IDs of (correctly) fixed bugs are available for them [17,53,55]. Fig. 19.7 shows the intersection of fixed bugs (in Fig. 19.7(a)) and correctly fixed bugs (in Fig. 19.7(b)) between ARJA-e, ACS, ssFix and SimFix, using Venn diagrams. From Fig. 19.7(a), ARJA-e performs much better than the other three approaches in terms of test-adequate bug fixing, and most of the bugs fixed by ACS, ssFix and SimFix can also be fixed by ARJA-e. From Fig. 19.7(b), ARJA-e fixes the highest number of bugs correctly (i.e., 39), where 20 bugs cannot be fixed correctly by any of the other three approaches. So ARJA-e indeed complements to the three approaches very well. But it should be noted that the three approaches also show good complementarity to ARJA-e in terms of correct bug fixing. Specifically, ACS, ssFix and SimFix can correctly fix 11, 9 and 16 bugs that cannot be correctly fixed by ARJA-e, respectively. This may be the case because ACS and ssFix are quite different from ARJA-e in technique. ACS aims at performing precise condition synthesis while ssFix uses existing code from a code database. It seems possible to further enhance the performance of ARJA-e by borrowing ideas from ACS and ssFix. For example, we can use a method similar to ACS to generate more accurate conditions for instantiating the template BEAR, or we can reuse the existing code outside the buggy program like ssFix.



(a) Test-adequate bug fixing (b) Correct bug fixing

Fig. 19.7: Venn diagram of repaired bugs by ARJA-e, ACS, ssFix and SimFix.

In summary, ARJA-e outperforms 13 existing repair approaches by a considerable margin. Specifically, by comparison with the best results, ARJA-e can fix 79.7% more bugs than ARJA (from 59 to 106), and can correctly fix 39.3% more bugs than SimFix (from 28 to 39). Moreover, ARJA-e is an effective approach complementary to the state-of-the-art techniques.

19.8.2 Novelty in Generated Repairs (RQ2)

We found that ARJA-e can fix some bugs in a different way from the human developer. These patches are generally beyond a programmer's expectations, showing the surprising novelty [28]. In the following, we will present case studies to demonstrate this point.

Fig. 19.8 shows a correct patch generated by ARJA-e for M94. The following function wants to compute greatest common divisor (GCD) of two integers. Certainly, if one of the integer is 0, GCD is equal to the sum of the absolute values. The bug is that if `u` or `v` is a large integer (e.g., `u = 3145728` and `v = 294912`), then `u * v` can be equal to 0 by mistake due to overflow. The human will just change `u * v == 0` to `u == 0 || v ==0`. But ARJA-e fixes it in a different way by changing `u` to `sign(u)`, where `sign(u)` is the sign function, to avoid the problem leading to the bug.

Fig. 19.9 shows the correct patch generated by ARJA-e. A human developer fixes this bug just by replacing line 5 with `int len = size - strLen + 1;`, where `size` is the number of characters in the array `buffer`. Instead, the patch by ARJA-e first replaces `buffer` in line 3 with `toCharArray()` which copies all characters in `buffer` into a new array with length exactly equal to `size`. Now `thisBuff.length` is equivalent to `size`. However, the value of `len` is still one less than the value it

```
// MathUtils.java
public static int gcd(int u, int v) {
-  if (u * v == 0) {
+  if (u == 0 || v == 0) {  // human-written patch
+  if (sign(u) * v == 0) {  // ARAJ-e patch
      return (Math.abs(u) + Math.abs(v));
   }
 ...  ...
}
```

Fig. 19.8: Human-written patch and correct patch generated by ARJA-e for bug M94.

should be, according to the human-written patch. To address this, ARJA-e further changes `i < len` to `i <= len`, achieving semantic equivalence.

```
// StrBuilder.java
public int indexOf(String str, int startIndex) { ...
-     char[] thisBuf = buffer;
+     char[] thisBuf = toCharArray();
      int len = thisBuf.length - strLen;
-     outer: for (int i = startIndex; i < len; i++) {
+     outer: for (int i = startIndex; i <= len; i++) {
       ... } ... }
```

Fig. 19.9: Correct patch generated by ARJA-e for bug L61.

Fig. 19.10 shows the correct patch generated by ARJA-e for bug M56. The human-written patch fixes this bug by firstly deleting lines 3–9 and then replacing line 10 with `indices[last] = index - count;`. Compared to this human-written patch, the ARJA-e patch just does a slight modification (i.e., replacing `idx` with `MathUtils.sign(idx)`). Since `idx` is positive, its sign `MathUtils.sign(idx)` is always equal to 1. Hence after line 9, `idx` is just equal to `index - count`, where `count` refers to its initial value at line 3. Consequently, the ARJA-e patch is semantically equivalent to the human-written patch and is therefore correct.

Fig. 19.11 shows a plausible patch generated by ARJA-e for bug M104. This bug is triggered because the maximum allowed numerical error (MANE) is too large. To fix the bug, the loop should be terminated until `Math.abs(an)` reaches a smaller value. So the human-written patch changes the initial value of of `epsilon` from `10e-9` to `10e-15` in order to ensure a smaller MANE. The ARJA-e patch shown in Fig. 19.11 achieves a similar functionality in a different way, which changes the method invocation `abs` to `sqrt`. Although this patch is not semantically equivalent to the human-written patch, it can make the entire test suite pass and is also indicative of the cause of the bug.

```
// MultidimensionalCounter.java
...
int idx = 1;
while (count < index) {
-    count += idx;
+    count += MathUtils.sign(idx);
     ++idx;
}
--idx;
indices[last] = idx;
return indices;
```

Fig. 19.10: Correct patch generated by ARJA-e for bug M56.

```
// Gamma.java
...  ...
-    while (Math.abs(an) > epsilon && n < maxIterations) {
+    while (Math.sqrt(an) > epsilon && n < maxIterations) {
         n = n + 1.0;
         an = an * (x / (a + n));
         sum = sum + an;
     }
```

Fig. 19.11: Plausible patch generated by ARJA-e for bug M104.

19.8.3 Effectiveness of Overfit Detection (RQ3)

In this subsection, we will evaluate our overfit detection approach described in Section 19.6.1. To demonstrate its effectiveness, we compare it with Xiong et al.'s approach (XA) [54], which is currently the state-of-the-art technique for overfit detection and shares certain similarities with our approach. To ensure a fair comparison, we use the version without test case generation for XA. According to [54], this simplified version has already achieved competitive performance compared to the version relying on new test cases.

For the subjects, we consider the first plausible patch found by ARJA-e for each bug (according to RQ1). In addition, we include the patches generated by jGenProg and jKali, which are collected from Martinez et al.'s empirical study [33] on Defects4J. In the end, we collect a dataset of 122 plausible patches by ignoring unsupported patches, where 97 patches are incorrect and 25 patches are correct. The correctness of ARJA-e patches is judged by ourselves, while the correctness of jGenProg and jKali patches is according to Martinez et al.'s analysis [33]. Table 19.6 shows the statistics of this dataset.

Tables 19.7 show the comparison results on the dataset per tool. From Tables 19.7, for the patches of ARJA-e and jGenProg, our approach can filter out more

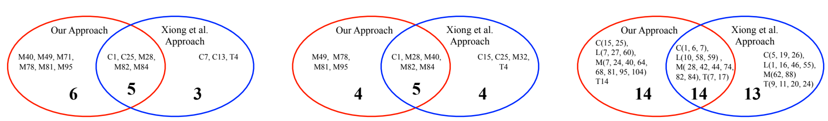
Table 19.6: Dataset of Plausible Patches Used in RQ3

Project	ARJA-e		jGenProg		jKali		Total	
	Incorrect	Correct	Incorrect	Correct	Incorrect	Correct	Incorrect	Correct
Chart	9	3	6	0	6	0	21	3
Lang	16	4	0	0	0	0	16	4
Math	23	11	13	5	13	1	49	17
Time	7	1	2	0	2	0	11	1
Total	55	19	21	5	21	1	97	25

incorrect patches than XA, while for the patches of jKali, the two approaches can identify the same number of incorrect patches. Moreover, our approach does not filter out any correct patch obtained by jGenProg and jKali, while XA filters out one correct patch (for bug M53) by jGenProg. Note that it was reported in [54] that XA does not exclude any correct patch by jGenProg. This inconsistency may be due to different computing environments. For the patches of ARJA-e, both approaches exclude correct patches by mistake, but our approach only excludes 3 out of 19 correct patches whereas XA excludes 7.

Table 19.7: Comparison Between Our Approach and Xiong et al.'s Approach in Overfit Detection (The Patches are Categorized by Repair Tools)

Tool	Incorrect	Correct	Incorrect Excluded		Correct Excluded	
			Our Approach	XA	Our Approach	XA
ARJA-e	55	19	28(50.91%)	27(49.09%)	3(15.79%)	7(36.84%)
jGenProg	21	5	11(52.38%)	8(38.10%)	0(0.00%)	1(20.00%)
jKali	21	1	9(42.86%)	9(42.86%)	0(0.00%)	0(0.00%)
Total	97	25	48(49.48%)	44(45.36%)	3(12.00%)	8(32.00%)



(a) jGenProg (b) jKali (c) ARJA-e

Fig. 19.12: Intersection of incorrect patches identified by our approach and Xiong et al.'s approach.

To further understand the performance difference between our approach and XA, Fig. 19.12 shows the intersection of incorrect patches identified by the two approaches. It is interesting to see that our approach complements to XA very well.

Specifically, our approach can identify 6 incorrect patches by jGenProg, 4 incorrect patches by jKali and 14 incorrect patches by ARJA-e, respectively, which cannot be identified by XA. In addition, we note that none of the 8 correct patches excluded by XA is also excluded by our approach. Given this strong complementarity, it is very promising to further try to improve the accuracy of overfit detection by properly combining the strength of the two approaches.

19.9 Conclusion

In this chapter, we have proposed a new repair system, called ARJA-e, for better evolutionary software repair. By combining two sources of fix ingredients, ARJA-e can conduct complex statement-level transformations, targeted code changes (e.g., adding a null pointer checker), and code changes at a finer-granularity than statement level, which gives ARJA-e great potential to fix various kinds of bugs. To reduce the search space and avoid nonsensical patches, ARJA-e uses a strategy based on a light-weight contextual analysis, which can filter out unpromising replacement and insertion statements, respectively. In order to harness the potential repair power of the search space, ARJA-e first unifies the edits at different granularities into statement-level edits, so as to encode patches in the search space with a lower-granularity patch representation that is characterized by the decoupling of statements for replacement and insertion. With this new patch representation, ARJA-e employs multi-objective GP to navigate the search space. To better guide the search of GP, ARJA-e uses a finer-grained fitness function that can make full use of semantic information provided by existing test cases. Moreover, ARJA-e includes a post-processing tool for alleviating patch overfitting. This tool can serve the purposes of overfit detection and patch ranking.

We have conducted an extensive empirical study on 224 real bugs in Defects4J. The evaluation results show that ARJA-e outperforms 13 existing repair approaches by a considerable margin in terms of both the number of bugs fixed and correctly fixed. Interestingly, we found that ARJA-e can fix some bugs in a creative way, which is usually beyond the exceptions of human programmers. In addition, we have shown that the proposed overfit detection technique shows several advantages over a state-of-the-art approach [54].

In the future, we plan to incorporate additional sources of fix ingredients (e.g., source code repositories [19, 53]) into our repair framework, which may increase the potential for fixing more bugs. Moreover, we would like to investigate new mating and survival selection methods [15, 40, 58] in GP, so as to further improve the evolutionary search algorithm for bug repair.

References

1. Arcuri, A.: It does matter how you normalise the branch distance in search based software testing. In: Proceedings of the Third International Conference on Software Testing, Verification and Validation, pp. 205–214. IEEE (2010)
2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic programming: An introduction, vol. 1. Morgan Kaufmann San Francisco (1998)
3. Barr, E.T., Brun, Y., Devanbu, P., Harman, M., Sarro, F.: The plastic surgery hypothesis. In: Proceedings of the 22nd International Symposium on Foundations of Software Engineering, pp. 306–317. ACM (2014)
4. Brameier, M.F., Banzhaf, W.: Linear genetic programming. Springer Science & Business Media (2007)
5. Campos, J., Riboira, A., Perez, A., Abreu, R.: Gzoltar: An eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 378–381. ACM (2012)
6. Chen, L., Pei, Y., Furia, C.A.: Contract-based program repair without the contracts. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 637–647. IEEE (2017)
7. Contributors, J.: A programmer-oriented testing framework for java (2004). URL https://github.com/junit-team/junit4
8. D'Antoni, L., Samanta, R., Singh, R.: Qlose: Program repair with quantitative objectives. In: Proceedings of International Conference on Computer Aided Verification, pp. 383–401. Springer (2016)
9. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Transactions on Evolutionary Computation **6**(2), 182–197 (2002)
10. Debroy, V., Wong, W.E.: Using mutation to automatically suggest fixes for faulty programs. In: Proceedings of the Third International Conference on Software Testing, Verification and Validation, pp. 65–74. IEEE (2010)
11. Fast, E., Le Goues, C., Forrest, S., Weimer, W.: Designing better fitness functions for automated program repair. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, pp. 965–972. ACM (2010)
12. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In: Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation, pp. 947–954. ACM (2009)
13. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: A survey. IEEE Transactions on Software Engineering **45**(1), 34–67 (2019)
14. Gupta, R., Pal, S., Kanade, A., Shevade, S.: Deepfix: Fixing common c language errors by deep learning. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence, pp. 1345–1351 (2017)
15. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. IEEE Transactions on Evolutionary Computation **19**(5), 630–643 (2015)
16. Hua, J., Zhang, M., Wang, K., Khurshid, S.: Towards practical program repair with on-demand candidate generation. In: Proceedings of the 40th International Conference on Software Engineering, pp. 12–23. ACM (2018)
17. Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X.: Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th International Symposium on Software Testing and Analysis, pp. 298–309. ACM (2018)
18. Just, R., Jalali, D., Ernst, M.D.: Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 437–440. ACM (2014)
19. Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y.: Repairing programs with semantic code search. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 295–306. IEEE (2015)

20. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: Proceedings of the 35th International Conference on Software Engineering, pp. 802–811. IEEE (2013)
21. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection, vol. 1. MIT press (1992)
22. Le, X.B.D., Chu, D.H., Lo, D., Le Goues, C., Visser, W.: Jfix: Semantics-based repair of java programs via symbolic pathfinder. In: Proceedings of the 26th International Symposium on Software Testing and Analysis, pp. 376–379. ACM (2017)
23. Le, X.B.D., Chu, D.H., Lo, D., Le Goues, C., Visser, W.: S3: syntax-and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, pp. 593–604. ACM (2017)
24. Le, X.B.D., Lo, D., Le Goues, C.: History driven program repair. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, pp. 213–224. IEEE (2016)
25. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering, pp. 3–13. IEEE (2012)
26. Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Software Quality Journal **21**(3), 421–443 (2013)
27. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**(1), 54–72 (2012)
28. Lehman, J., Clune, J., Misevic, D., Adami, C., Altenberg, L., Beaulieu, J., Bentley, P.J., Bernard, S., Beslon, G., Bryson, D.M., et al.: The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. arXiv preprint arXiv:1803.03453 (2018)
29. Liu, X., Zhong, H.: Mining stackoverflow for program repair. In: Proceedings of 25th International Conference on Software Analysis, Evolution and Reengineering, pp. 118–129. IEEE (2018)
30. Long, F., Amidon, P., Rinard, M.: Automatic inference of code transforms for patch generation. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, pp. 727–739. ACM (2017)
31. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, pp. 166–178. ACM (2015)
32. Long, F., Rinard, M.: Automatic patch generation by learning correct code. ACM SIGPLAN Notices **51**(1), 298–312 (2016)
33. Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M.: Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empirical Software Engineering **22**(4), 1936–1964 (2017)
34. Martinez, M., Monperrus, M.: Astor: A program repair library for java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 441–444. ACM (2016)
35. Martinez, M., Monperrus, M.: Ultra-large repair search space with automatically mined templates: the cardumen mode of astor. In: International Symposium on Search Based Software Engineering, pp. 65–86. Springer (2018)
36. Martinez, M., Weimer, W., Monperrus, M.: 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, pp. 492–495. ACM (2014)
37. Mechtaev, S., Yi, J., Roychoudhury, A.: Directfix: Looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering, pp. 448–458. IEEE Press (2015)
38. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering, pp. 691–701. ACM (2016)
39. Monperrus, M.: Automatic software repair: A bibliography. ACM Computing Surveys **51**(1), 17 (2018)

40. Mouret, J.B., Clune, J.: Illuminating search spaces by mapping elites. arXiv preprint arXiv:1504.04909 (2015)
41. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: Program repair via semantic analysis. In: Proceedings of the 35th International Conference on Software Engineering, pp. 772–781. IEEE (2013)
42. Oliveira, V.P.L., de Souza, E.F., Le Goues, C., Camilo-Junior, C.G.: Improved representation and genetic operators for linear genetic programming for automated program repair. Empirical Software Engineering **23**(5), 2980–3006 (2018)
43. Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C.: The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering, pp. 254–265. ACM (2014)
44. Qi, Z., Long, F., Achour, S., Rinard, M.: 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, pp. 24–36. ACM (2015)
45. Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., Hartmann, B.: Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering, pp. 404–415. IEEE Press (2017)
46. Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R.: Elixir: Effective object-oriented program repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 648–659. IEEE (2017)
47. Smith, E.K., Barr, E.T., Le Goues, C., Brun, Y.: Is the cure worse than the disease? overfitting in automated program repair. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, pp. 532–543. ACM (2015)
48. Tan, S.H., Yoshida, H., Prasad, M.R., Roychoudhury, A.: Anti-patterns in search-based program repair. In: Proceedings of the 24th International Symposium on Foundations of Software Engineering, pp. 727–738. ACM (2016)
49. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Communications of the ACM **53**(5), 109–116 (2010)
50. Weimer, W., Fry, Z.P., Forrest, S.: Leveraging program equivalence for adaptive program repair: Models and first results. In: Proceedings of the 28th International Conference on Automated Software Engineering, pp. 356–366. IEEE (2013)
51. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering, pp. 364–374. IEEE (2009)
52. Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.C.: Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International Conference on Software Engineering, pp. 1–11. ACM (2018)
53. Xin, Q., Reiss, S.P.: Leveraging syntax-related code for automated program repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 660–670. IEEE Press (2017)
54. Xiong, Y., Liu, X., Zeng, M., Zhang, L., Huang, G.: Identifying patch correctness in test-based program repair. In: Proceedings of the 40th International Conference on Software Engineering, pp. 789–799. ACM (2018)
55. Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., Zhang, L.: Precise condition synthesis for program repair. In: Proceedings of the 39th International Conference on Software Engineering, pp. 416–426. IEEE Press (2017)
56. Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S.L., Durieux, T., Le Berre, D., Monperrus, M.: Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Transactions on Software Engineering **43**(1), 34–55 (2017)
57. Yuan, Y., Banzhaf, W.: Arja: Automated repair of java programs via multi-objective genetic programming. IEEE Transactions on Software Engineering (2018). DOI 10.1109/TSE.2018.2874648
58. Yuan, Y., Xu, H., Wang, B., Yao, X.: A new dominance relation-based evolutionary algorithm for many-objective optimization. IEEE Transactions on Evolutionary Computation **20**(1), 16–37 (2016)