A Hybrid Evolutionary System for Automatic Software Repair

Yuan Yuan
Michigan State University
East Lansing, Michigan
yyuan@msu.edu

Wolfgang Banzhaf
Michigan State University
East Lansing, Michigan
banzhafw@msu.edu

ABSTRACT

This paper presents an automatic software repair system that combines the characteristic components of several typical evolutionary computation based repair approaches into a unified repair framework so as to take advantage of their respective component strengths. We exploit both the redundancy assumption and repair templates to create a search space of candidate repairs. Then we employ a multi-objective evolutionary algorithm with a low-granularity patch representation to explore this search space, in order to find simple patches. In order to further reduce the search space and alleviate patch overfitting we introduce replacement similarity and insertion relevance to select more related statements as promising fix ingredients, and we adopt anti-patterns to customize the available operation types for each likely-buggy statement. We evaluate our system on 224 real bugs from the Defects4J dataset in comparison with the state-of-the-art repair approaches. The evaluation results show that the proposed system can fix 111 out of those 224 bugs in terms of passing all test cases, achieving substantial performance improvements over the state-of-the-art. Additionally, we demonstrate the ability of ARJA-e to fix multi-location bugs that are unlikely to be addressed by most of existing repair approaches.

CCS CONCEPTS

• **Software and its engineering → Genetic programming**; **Search-based software engineering**;

KEYWORDS

Program repair, genetic programming, evolutionary multi-objective optimization

ACM Reference Format:
Yuan Yuan and Wolfgang Banzhaf. 2019. A Hybrid Evolutionary System for Automatic Software Repair. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3321707.3321830

1 INTRODUCTION

Automatic software repair [9, 30, 38] aims to fix bugs in software automatically, which generally relies on a specification. When a test suite is considered as the specification, the paradigm is called *test-suite based repair* [30]. 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* [23] or a *plausible patch* [34].

Evolutionary computation (EC) based repair approaches [38] are a popular category of techniques for test-suite based repair. These approaches determine a *search space* potentially containing correct patches, then use the *evolutionary algorithm* (EA) to explore that search space. GenProg [18, 19] is a pioneering approach of this kind, which uses three types of statement-level edit operations (i.e., deletion, replacement and insertion) to rearrange the statements already extant in the buggy program and uses genetic programming [3, 14] to search for plausible patches. Although GenProg exhibits promising performance [18], there are deficiencies in both of its two key elements: search space and EA, which inspire further examination of EC-based repair approaches.

In regard to the search space, GenProg exploits the *redundancy assumption* [26] (also called *plastic surgery hypothesis* [2]) and only uses existing statements in the buggy program for replacement or insertion. The problem here is that the 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 the bug. To relieve the issue, Kim *et al.* [13] 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 [44], the redundancy-based approaches can really fix some bugs that cannot be fixed by typical template-based approaches (e.g., PAR and ELIXIR [36]) which implies that combining the redundancy assumption and repair templates to generate fix statements could further improve repair effectiveness.

As for the aspect of the EA, the genetic operators in GenProg are only executed on high-granularity edits and the crossover therein will not produce any new edits. A recent study [32] indicates that evolving such high-level units will strongly limit the ability of GenProg to effectively traverse the search space. This may partly explain why GenProg usually cannot generate repairs beyond simple ones equivalent to a single functionality deletion [34]. Inspired by [32], a very recent EC-based approach ARJA [44] incorporates a three-part lower-granularity patch representation (i.e., decoupling

GECCO '19, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6111-8/19/07…$15.00
https://doi.org/10.1145/3321707.3321830

the search subspaces of likely-buggy locations, operation types and potential fix statements), where genetic operators act on each part separately. Due to its improved search ability, an ARJA version working over GenProg's search space can fix many more real bugs from Defects4J [11] than GenProg, and it can even handle several multi-location bugs which cannot be fixed by most of the existing repair approaches.

Another problem with GenProg lies in *patch overfitting*. That is, GenProg usually produces patches that are incorrect beyond passing the test suite. This problem has attracted a lot of attention since the empirical work by Qi *et al.* [34]. Strictly speaking, a repair approach itself is not responsible for patch overfitting but a weak test suite is [29]. However it is common that the test suite for a bug is not strong enough. So it is necessary to develop some strategies to alleviate this practical problem so as to focus more search efforts on finding correct patches. Tan *et al.* [37] propose to prohibit a set of specified program transformations (called anti-patterns) in GenProg that can easily result in unreasonable changes to the program behavior. Another strategy is from ARJA [44], which considers to explicitly minimize patch size during the evolutionary process. Besides having better comprehensibility and maintainability, smaller patches generally have better generalization ability to unseen tests according to Occam's razor, so there is less risk of patch overfitting.

The above mentioned improvements over GenProg in regard to search space, EA and patch overfitting can all to some extent result in better repair performance. However, to our knowledge, there is no single approach endowed with all of them. Our basic idea in this paper is to combine the strengths from characteristic elements that contribute to all of these improvements so as to construct a new EC-based repair approach that represents the state of the art. To this end, we propose a hybrid evolutionary repair system for Java, namely ARJA-e (an enhanced version of ARJA [44]), which incorporates the redundancy assumption, repair templates, lower-granularity patches, anti-patterns and the minimization of patch size into a unified repair framework. Overall, ARJA-e employs a multi-objective evolutionary algorithm (MOEA) [45] with a lower-granularity patch representation to search for smaller patches over a promising search space determined by both the redundancy assumption and repair templates, where anti-patterns are used to assign available operation types for each likely buggy statement. We evaluate ARJA-e on 224 real bugs from Defects4J [11], and compare it with almost all the approaches ever tested on this database. The results show that ARJA-e performs much better than state-of-the-art repair approaches in terms of both the number of bugs fixed and correctly fixed. Specifically, ARJA-e can fix 111 out of 224 bugs in terms of plausible repair. Furthermore, ARJA-e is able to correctly fix several multi-location bugs that cannot be handled by most of the existing repair approaches.

The overall contribution of this paper is the construction of a new EC-based repair system that integrates the characteristic components of several existing ones. The extension and enhancement of each component along with how to seamlessly combine all the components constitute the detailed technical contribution that can be summarized as follows:

1) We use both the redundancy assumption and repair templates as the source for potential fix statements, in order to leverage their complementarity.

2) To reduce the search space and alleviate patch overfitting, we present two metrics in order to select the most related statements for consideration among those copied from the buggy program.

3) We extend the templates of PAR to make them more general and accommodate more potentially useful fix patterns.

4) We convert various template-based repair actions (usually occurring at the expression level) into two types of statement-level edits so that all kinds of edits can be decomposed into the same partial information, which makes it possible to encode patches with a unified lower-granularity representation.

5) We introduce a new lower-granularity patch representation specifically for crossover which is characterized by the decoupling of statements for replacement and statements for insertion. This insight has two reasons: (i) the potentially useful replacement or insertion statement for a faulty statement usually have quite different features and it would be better to evolve them separately; (ii) any potential fix statement generated by templates is used either for replacement or for insertion.

2 BACKGROUND AND MOTIVATION

2.1 Related Work

Our system belongs to the class of EC-based repair approaches which explore a repair search space using evolutionary algorithms. GenProg [18, 19], PAR [13], GenProg with anti-patterns [37] and ARJA [44] all fall into this category. Their basic ideas have been described in Section 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 [33] uses random search and AE [39] uses an adaptive search strategy).

Inspired by the idea of using templates [13], some repair approaches (e.g., SPR [21] and ELIXIR [36]) employ a set of richer templates (or code transformations) that are defined manually. Genesis [20] aims to automatically infer such code transformations from successful patches. Cardumen [25] 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 [17] uses mined historical bug fixes to guide its random search. ACS [42] uses the information of javadoc comments to rank variables. SearchRepair [12] and ssFix [41] 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 [31] is a pioneer in this category. Other typical approaches of this kind include DirectFix [27], QLOSE [5], Angelix [28], Nopol [43], JFix [15] and S3 [16]. Recently, machine learning techniques have been used in software repair. Prophet [22] uses a probabilistic model to rank the candidate patches over the search space of SPR. DeepFix [10] uses deep learning to fix common programming errors.

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. 1 show the human-written patch for bug Math85 from the Defects4J [11] dataset. To correctly fix this bug, only a slight modification is required (i.e., change >= to >), as shown in Fig. 1. However, redundancy-based approaches (e.g., GenProg [18, 19], RSRepair [33] and AE [39]) 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 [7, 24] and ELIXIR [36]) 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 [34] by deleting the whole buggy `if` statement: `if (fa * fb >= 0){...}`), leading to a plausible but incorrect patch.

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

Figure 1: The human-written patch for bug Math85.

Fig. 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. 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.

```
1  public void integrate (...) throws ... { ...
2  +  if (forward) {
3  +    if ( stepStart + stepSize >= t) { stepSize = t - stepStart ; }
4  +  } else {
5  +    if ( stepStart + stepSize <= t) { stepSize = t - stepStart ; } }
6     ... }
```

Figure 2: The human-written patch for bug Math39.

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

3 APPROACH

This section describes the details of our approach, which is implemented as a tool called ARJA-e that repairs Java software.

3.1 Overview

The input of ARJA-e is a buggy program associated with a JUnit test suite. The test suite should contain at least one negative test exposing the bug to be fixed and a number of positive tests specifying the required program behavior. ARJA-e basically aims to make the code pass all these tests by modifying the buggy program, which consists of the following main steps:

1) **Fault Localization:** Given an input, ARJA-e first uses a fault localization technique called Ochiai [1] to locate a list of likely-buggy statements (LBSs). Each LBS is assigned a suspiciousness value $susp_j \in [0, 1]$ by Ochiai, indicating the likelihood of the LBS containing the bug. To reduce the search space, not all LBSs returned by Ochiai are considered. We select at most $n_{\max}$ LBSs with the largest $susp_j$ values, and moreover the LBSs with $susp_j$ smaller than a threshold $\gamma_{\min}$ will be ignored. $n_{\max}$ and $\gamma_{\min}$ are parameters to be set.

2) **Test Filtering:** Suppose n LBSs are selected according to $n_{\max}$ and $\gamma_{\min}$. We conduct coverage analysis to filter out positive tests that are unrelated to the manipulation of these n LBSs.

3) **Exploiting the Redundancy Assumption:** Similar to GenProg [18], ARJA-e can use three types of statement-level edit operations to manipulate a LBS (i.e., *delete* the LBS, *replace* the LBS with another statement and *insert* another statement before the LBS). For the latter two types, another statement is needed, one source of which is the current buggy program, following the redundancy assumption [2, 26]. Unlike existing approaches based on the redundancy assumption[8, 18, 19, 33, 39, 40, 44], however, ARJA-e separates the statements for replacement and those for insertion. So each LBS corresponds to two different sets of statements denoted by R_j (for replacement) and I_j (for insertion) respectively.

4) **Exploiting Repair Templates:** In ARJA-e, an additional source of statements for replacement (in R_j) and insertion (in I_j) is a generic set of repair templates. We extend the repair templates used in PAR [13], making them more general and accommodate more fix patterns. Unlike PAR that executes templates *on-the-fly*, ARJA-e uses templates in an *offline* manner so that we can convert various template-based edits (usually at the expression-level) into two types of statement-level edits (i.e., replacement and insertion). The benefit of this is that we can integrate redundancy-based statement-level edits and template-based edits into a unified evolutionary framework with a lower-granularity patch representation.

5) **Initialization of Operation Types:** As mentioned before, ARJA-e uses three types of edit operations. However, for some LBSs, not all operation types are desirable. In this phase, we use some rules [44] and anti-patterns [37] to determine the set of available operation types (denoted by O_j) for each LBS.

6) **Multi-Objective Evolution of Patches:** Now we have n LBSs for consideration, each associated with two sets of statements (i.e., R_j and I_j) and a set of available operation types (i.e., O_j). The

goal of ARJA-e is to find a plausible patch that consists of a list of edits (each edit is a statement-level deletion, replacement or insertion), where smaller patches are preferred. We formulate this problem as a multi-objective optimization/search problem and employ NSGA-II [6] to explore the search space.

In the rest of this section, we detail how to exploit the redundancy assumption and repair templates to obtain statements for replacement and insertion, how to customize operation types for each LBS and how to evolve patches using multi-objective optimization.

3.2 Exploiting the Redundancy Assumption

For each LBS selected, we first identify the Java package the LBS resides in. Then we collect all the statements within this package. We scan these statements one by one. For each of them (denoted by **s**), we first examine whether the variables and method invocations included in the statement **s** are in-scope at the destination of the LBS. If **s** is out of the variable or method scope, we just ignore it, otherwise we further check whether **s** follows the *second* type of rules (6 rules in total, denoted by F_a) introduced in ARJA [44]. The motivation for the 6 rules is that although some statements can pass a check of variable and method scope, they may violate other Java language specifications. For example, a `continue` statement does not include any variable or method invocation, but it can only be used in a loop. If **s** follows F_a, we further use the *third* type of rules defined in ARJA [44], among which 3 rules (denoted by F_b) are related to the replacement operation and the other 3 (denoted by F_c) to the insertion operation. For example, to avoid disrupting the program too much, one of the rules in F_b is to not replace a variable declaration statement with other kinds of statements. Even if **s** follows F_b/F_c, we do not put it into the set R_j/I_j right away. 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 in ARJA-e.

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

$$sim(\mathbf{s}, \text{LBS}) = \frac{|V_{\text{LBS}} \cap V_{\mathbf{s}}|}{|V_{\text{LBS}} \cup V_{\mathbf{s}}|} \tag{1}$$

In the method where the LBS resides, suppose V_{bef} and V_{aft} are the sets of variables used by the statements before and after the LBS, respectively. Here "after" includes the LBS itself. We define the relevance of **s** to the context of LBS as follows

$$rel(\mathbf{s}, \text{LBS}) = \frac{1}{2}\left(\frac{|V_{\mathbf{s}} \cap V_{\text{bef}}|}{|V_{\mathbf{s}}|} + \frac{|V_{\mathbf{s}} \cap V_{\text{aft}}|}{|V_{\mathbf{s}}|}\right) \tag{2}$$

This equation indeed averages the percentages of the variables in $V_{\mathbf{s}}$ that are covered by V_{bef} and V_{aft}.

If $|V_{\text{LBS}} \cup V_{\mathbf{s}}| = 0$, $sim(\mathbf{s}, \text{LBS})$ is set to 1, and if $|V_{\mathbf{s}}| = 0$, $rel(\mathbf{s}, \text{LBS})$ is set to 0. So $sim(\mathbf{s}, \text{LBS}) \in [0, 1]$ and $rel(\mathbf{s}, \text{LBS}) \in [0, 1]$. Only when $sim(\mathbf{s}, \text{LBS}) > \beta_{\text{sim}}$, can **s** be put into R_j, and only when $rel(\mathbf{s}, \text{LBS}) > \beta_{\text{rel}}$, can **s** be put into I_j, where β_{sim} and β_{rel} are predetermined threshold parameters.

For each LBS considered, Fig. 3 summarizes the procedure in ARJA-e to check whether statement **s** can become a potentially useful statement for replacement or insertion.

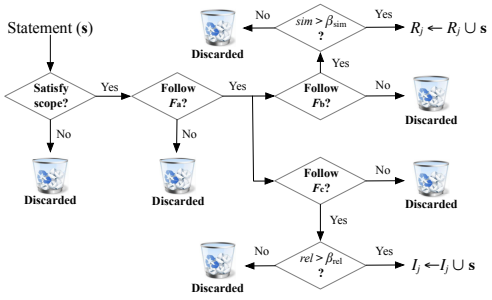


Figure 3: The procedure in ARJA-e to check whether s is a potentially useful statement for replacement and insertion.

3.3 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.

1) **Element Replacer (ER):** This template replaces an abstract syntax tree (AST) node element in a LBS with another compatible one. Table 1 lists the elements that can be replaced and also shows alternative replacers for each kind of elements. This template generalizes 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 [36] and replacing `x` with `f(x)` follows the transformation schema in REFAZER [35]).

Table 1: 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	this.a or super.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 or double	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., ++ or −−	The opposite prefix/postfix operator, e.g., ++ to −−
Assignment operators	e.g., += or *=	The opposite assignment operator e.g., += to −=, *= to \=
Conditional expression	a ? b : c	b or c

2) **Method Parameter Adjuster (MPA):** This template changes a method invocation in a LBS to another overloaded method. When adding a parameter, we consider all the visible fields and local variables at the LBS's location. Candidates are those type-compatible with the corresponding parameter type in the overloaded method declaration. When removing a parameter, we can choose any one in the current parameter list.

3) **Boolean Expression Adder or Remover (BEAR):** This template applies to the LBS having a conditional branch (i.e., `if` or

while statement). It adds a term to the predicate (with && ||) or just removes a term. When adding a term, we collect all boolean expressions that are in the file where the LBS resides.

4) **Null Pointer Checker (NPC):** This template assures that all object references in a LBS cannot be null. Suppose a LBS includes an object reference *o*, then we use 6 alternative transformation schemata to manipulate the LBS:

```
(i) if (o != null) LBS;          (ii) if (o == null) return sth;
(iii) if (o == null) throw sth;  (iv) if (o == null) break;
(v) if (o == null) continue;     (vi) if (o == null) o = new Obj();
```

The if statement in the first schema is used to replace the LBS, whereas each in the other five should be inserted before the LBS.

Compared to ARJA-e, PAR only uses the first schema, limiting its ability to handle null pointer bugs. In the following, we introduce another three repair templates similar to NPC. These templates check different contexts but still use the above 6 transformation schemata to edit the LBS.

5) **Range Checker (RC):** For a LBS, this template examines whether all array or list element accesses are valid (i.e., an index cannot exceed the low and upper bounds of the array/list size). Unlike PAR, ARJA-e also considers to check the validity of the char access (in the form of charAt or substring) for String objects, since String is a list of characters and is commonly used in Java.

6) **Cast Checker (CC):** This template checks whether the variable or expression to be converted is an instance of casting type (using instanceof operator) in each class-casting expression.

7) **Divide-by-Zero Checker (DC):** This template ensures all divisors in a LBS are different from 0. This is not used in PAR.

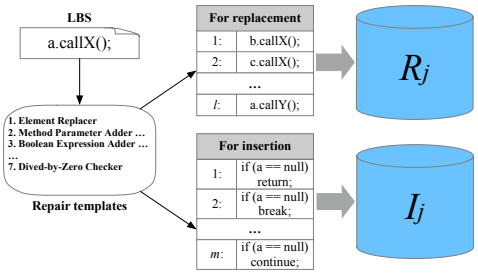


Figure 4: Illustration of the offline execution of templates.

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. 4 illustrates the way to exploit the templates in ARJA-e. Note that we do not consider similarity and relevance as in Section 3.2 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.

3.4 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 [34]. To address the first problem, we use the two rules defined in [44], 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 [37]. 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.

3.5 Multi-Objective Evolution of Patches

In the previous steps, we have determined the repair search space of ARJA-e. Specifically, we select n LBSs and each LBS corresponds to three sets: R_j, I_j and O_j. The statements in R_j/I_j are collected by exploiting the redundancy assumption and repair templates (see Sections 3.2 and 3.3). To find plausible patches, we use a classical MOEA (i.e., NSGA-II) to explore the search space. The details are given as follows.

3.5.1 Patch Representation. To encode a patch as a MOEA individual, we first number the LBSs and the elements in R_j, I_j and O_j, starting from 1, where $j \in \{1, 2, \ldots, n\}$, in random order. All the IDs are fixed throughout the search. In ARJA-e, a patch (or solution) is represented as a genome $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ with the fixed length n. $\mathbf{x}_j$, $j \in \{1, 2, \ldots, n\}$, encodes the information of the edit on the j-th LBS, which consists of four parts: $\mathbf{x}_j = (x_{j1}, x_{j2}, x_{j3}, x_{j4})$. $x_{j1} \in \{0, 1\}$ indicates whether the j-th LBS is to be edited or not. $x_{j2} \in \{1, 2, \ldots, |O_j|\}$ indicates the x_{j2}-th operation type in O_j is used. $x_{j3} \in \{1, 2, \ldots, |R_j|\}$ means if "Replace" is used, the x_{j3}-th statement in R_j is selected to replace the j-th LBS. $x_{j4} \in \{1, 2, \ldots, |I_j|\}$ means if "Insert" is used, insert the x_{j4}-th statement in I_j before the j-th LBS. Suppose the j-th LBS is a.callX(), Fig. 5 illustrates the patch representation in ARJA-e, where the edit on the j-th LBS is: replace a.callX() with b.callX().

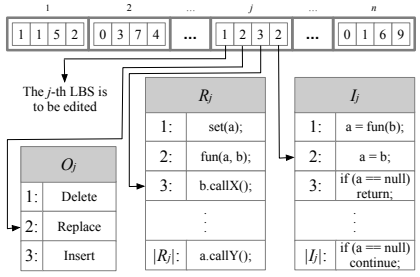


Figure 5: Illustration of the patch representation in ARJA-e.

3.5.2 Fitness Function. We formulate automatic software repair as a multi-objective optimization problem. To evaluate the fitness

of an individual $\mathbf{X}$, we use a multi-objective function that simultaneously minimizes the *patch size* and the *weighted failure rate*. The patch size is defined as $f_1(\mathbf{X}) = \sum_{j=1}^{n} x_{j1}$, which means the number of edits in patch $\mathbf{X}$. The weighted failure rate $f_2(\mathbf{X})$ measures how well the modified program functions in terms of passing tests:

$$f_2(\mathbf{X}) = \frac{|\{t \in T_f \mid \mathbf{X} \text{ fails } t\}|}{|T_f|} + w \times \frac{|\{t \in T_c \mid \mathbf{X} \text{ fails } t\}|}{|T_c|} \quad (3)$$

where T_f is the set of negative tests, T_c is the reduced set of positive tests, and $w \in (0, 1]$ is a global parameter which is used to put more emphasis on the passing of negative tests. If $f_2(\mathbf{X}) = 0$, $\mathbf{X}$ is a plausible patch. By also minimizing $f_1(\mathbf{X})$, we introduce a search bias for simpler patches.

3.5.3 Genetic Operators. Crossover and mutation are conducted to generate offspring individuals in MOEAs. In ARJA-e, each individual can be seen as a list of $4n$ integers. For crossover, the non-matching integers between the two parents are swapped with a fixed probability of 0.5. As for mutation, we first use roulette wheel selection to choose a LBS, where the j-th LBS is chosen with a probability of $susp_j / \sum_{j=1}^{n} susp_j$; then we replace the information of the whole edit (4 integers) corresponding to the chosen LBS with a randomly generated one.

3.5.4 Computational Search. We employ NSGA-II [6] as the multi-objective search framework. When the search is terminated, non-dominated solutions with $f_2 = 0$ in the final population are output as plausible patches. If no such solutions exist, ARJA-e fails to fix the bug.

3.6 Ranking Plausible Patches

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 two metrics for a patch. The first metric, denoted by *Susp*, represents the summation of the suspiciousness for the faulty locations modified by the patch. Before defining the second 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 3.3. We assign a preference score for each operation type: NPC, RC, CC and DC is scored 1, MPA is scored 2 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.

When comparing two patches in ranking, the patch with higher *Susp* is ranked higher. If *Susp* values are equal, $Pref$ values are further compared, and the patch with smaller $Pref$ is ranked higher. If *Susp* and $Pref$ cannot distinguish two patches, the patch found earlier is ranked higher.

4 EXPERIMENTAL DESIGN

4.1 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 our repair approach fix multi-location bugs?

RQ3: To what extent do the redundancy assumption and repair templates contribute to the overall performance of ARJA-e, and is it beneficial to exploit both of them?

RQ4: Can the search space be reduced effectively based on the similarity and relevance measures presented in Section 3.2?

Note that when we consider whether a bug is correctly fixed in RQ1 and RQ2, we only select the plausible patch ranked first for analysis. This practice can make the comparison fairer, since the compared approaches in the literature usually choose a single patch for verifying correctness. Whereas in RQ3 and RQ4, we regard a bug as correctly fixed if at least one of the patches is identified as correct, so all correct patches will be used for analysis.

4.2 Dataset of Bugs

We perform the empirical evaluation on a database of real bugs, called Defects4J [11], which has been extensively used for evaluating Java repair systems [4, 23, 25, 36, 41, 42, 44]. We consider four projects in Defects4J, namely Chart, Time, Lang and Math. Table 2 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 2: 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

4.3 Parameter Setting

Table 3 shows the parameter setting for ARJA-e in the experiments. Note that crossover and mutation operators presented in Section 3.5.3 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. Following the practice in [23, 25], every trial is terminated after 3 hours. All experiments are conducted in a high performance computing center and use machines with 2.4 GHz Intel Xeon E5 processor with 50 GB memory.

Table 3: 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 3.5.2	0.5

5 RESULTS AND DISCUSSIONS

This section presents empirical results to address the research questions (RQs 1–4) set out in Section 4.1.

Table 4: Comparison in terms of the number of bugs fixed and correctly fixed (Plausible/Correct).

Project	ARJA-e	jGenProg	jKali	xPAR	Nopol	HDRepair	ACS	ssFix	JAID	ELIXIR	ARJA	Cardumen
Chart	**19/7**	7/0	6/0	NA/0	6/1	NA/2	2/2	7/2	4/4	7/4	9/3	15/NA
Time	**8/2**	2/0	2/0	NA/0	1/0	NA/1	1/1	4/0	0/0	3/2	4/1	6/NA
Lang	**29/8**	0/0	0/0	NA/1	7/3	NA/7	4/3	12/5	8/5	12/8	17/4	7/NA
Math	**55/21**	18/5	14/1	NA/2	21/1	NA/6	16/12	26/7	8/7	19/12	29/10	37/NA
Total	**111/38**	27/5	22/1	NA/3	35/5	NA/16	23/18	49/14	20/16	41/26	59/18	65/NA

"NA" means the data is not available since it is not reported by the original authors.

5.1 Performance Evaluation (RQ1)

To show the advantage of ARJA-e over state-of-the-art tools, we compare ARJA-e with 11 existing tools in terms of the number of bugs fixed and correctly fixed. The 11 tools are jGenProg [23] (an implementation of GenProg for Java), jKali [23] (an implementation of Kali [34] for Java), xPAR (a reimplementation of PAR by Le *et al.* [17]), Nopol [23, 43], HDRepair [17], ACS [42], ssFix [41], JAID [4], ELIXIR [36], ARJA [44] and Cardumen [25], which include almost all approaches that have ever been tested on Defects4J. We manually examined the correctness of patches found by ARJA-e and identified a patch as correct if it is exactly the same as or semantically equivalent to the human-written patch. Table 4 shows the comparison results.

ARJA-e outperforms all other approaches by a large margin. Specifically, by comparison with the best results, ARJA-e can find plausible patches for 70.8% more bugs than Cardumen (from 65 to 111) and can generate correct patches for 46.2% more bugs than ELIXIR (from 26 to 38).

(a) Plausible fixing (b) Correct fixing

Figure 6: Venn diagram of repaired bugs.

Fig. 6 shows the intersection of fixed bugs (in Fig. 6(a)) and correctly fixed bugs (in Fig. 6(b)) between ARJA-e, ACS, ssFix and ARJA, using the Venn diagram. ACS, ssFix and ARJA are selected here since they show prominent performance among the 11 approaches of the comparison. The overwhelming majority of bugs (correctly) fixed by ARJA can also be (correctly) fixed by ARJA-e. Since ARJA-e incorporates core ideas of ARJA, this implies that the incorporation is very effective, making ARJA-e inherit almost all repair power from ARJA. Compared to ARJA, ACS and ssFix show better complementarity to ARJA-e. For example, ACS and ssFix can correctly fix 12 and 7 bugs that cannot 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.

5.2 Results from Multi-Location Bugs (RQ2)

Among the 111 bugs fixed by ARJA-e, the plausible patch for 19 bugs contains at least two edits that manipulate multiple buggy locations. By using delta debugging, we have verified that none of these patches can be reduced to a single edit. So the 19 bugs should be regarded as multi-location bugs. Furthermore, 7 out of the 19 bugs (i.e., T15, L20, L34, L61, M4, M22 and M98) are classified as being correctly fixed. For M22 and M98, ARJA-e can generate a correct patch that is exactly the same as the human-written patch. As for the other 6 bugs, ARJA-e can synthesize semantically equivalent ones. The results on multi-location bugs demonstrate the strong search ability of the customized MOEA in ARJA-e.

```
1  static Map<Object, Object> getRegistry () {
2  -    return REGISTRY.get() != null ? REGISTRY.get() :
3  -        Collections.<Object, Object>emptyMap();
4  +    return REGISTRY.get();
5  }
6  static boolean isRegistered(Object value) {
7      Map<Object, Object> m = getRegistry();
8  +    if (!(m != null)) return false;
9      return m.containsKey(value);
10 }
```

Figure 7: Correct patch generated by ARJA-e for bug L34.

To further understand the strength of ARJA-e in this respect, Fig. 7 shows a correct patch found by ARJA-e for bug L34. As can be seen, ARJA-e simultaneously uses two types of templates to generate the patch, ER for lines 3–4 and NPC for line 10.

Note that for T15 and L61, the human-written patch contains only a single statement-level edit. But these two patches are not within the search space of ARJA-e. ARJA-e fixes them correctly in a creative way. Take L61 for example, Fig. 8 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()` that 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 should be, according to the human-written patch. To address this, ARJA-e further changes `i < len` to `i <= len`, achieving semantic equivalence.

5.3 Redundancy Assumption vs. Repair Templates (RQ3)

Table 5 shows the contribution of the redundancy assumption (including two types of repair actions) and repair templates (corresponding to 7 types of repair actions) to the number of bugs fixed

```
1  public int indexOf(String str, int startIndex) { ...
2  -    char[] thisBuf = buffer;
3  +    char[] thisBuf = toCharArray();
4       int len = thisBuf.length - strLen;
5  -    outer: for (int i = startIndex; i < len; i++) {
6  +    outer: for (int i = startIndex; i <= len; i++) {
7         ... } ... }
```

Figure 8: Correct patch generated by ARJA-e for bug L61.

(i.e., test-adequate) and correctly fixed by ARJA-e. From Table 5, both make substantial contributions. SR and SI contribute to the fixing of 19 and 27 bugs respectively, and both contribute to the correct fixing of 7 bugs. ER has the most contribution among the repair templates. In total, the redundancy assumption contributes to the fixing of 46 bugs and to the correct fixing of 14 bugs, versus 72 and 37 by the repair templates. Although there are small overlaps in these numbers since the repair of several bugs (e.g., L34) uses more than one type of repair action, they basically reflect the overall contribution of the two components to the results of ARJA-e.

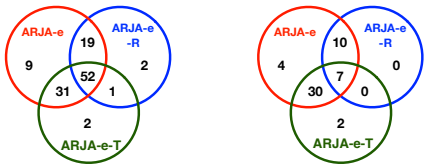
Table 5: Contribution of each operation type

Redundancy Assumption	Plausible	Correct
Statement-Level Replacement (SR)	19	7
Statement-Level Insertion (SI)	27	7
Repair Template	**Plausible**	**Correct**
Element Replacer (ER)	45	24
Method Parameter Adjuster (MPA)	4	3
Boolean Expression Adder/Remover (BEAR)	12	1
Null Pointer Checker (NPC)	7	6
Range Checker (RC)	2	1
Cast Checker (CC)	1	1
Divide-By-Zero Checker (DC)	1	1

To further show the benefits of exploiting both the redundancy assumption and repair templates, we compare ARJA-e with two ARJA-e variants referred to as ARJA-e-R and ARJA-e-T. With the same search algorithm as ARJA-e, ARJA-e-R only uses the redundancy assumption as a source for potential fixes while ARJA-e-T only uses the repair templates. Note that to completely understand the expressive power of the redundancy assumption, in ARJA-e-R we do not use the replacement similarity and insertion relevance to filter statements. Fig. 9 shows the comparison results. ARJA-e can fix almost all the bugs that are fixed by ARJA-e-R and ARJA-e-T, in terms of both plausible and correct bug fixing. Also, given that ARJA-e-R and ARJA-e-T show good performance complementarity, we conclude that ARJA-e successfully leverages the complementary strength of the redundancy assumption and repair templates.

5.4 Effect of Search Space Reduction (RQ4)

As described in Section 3.2, we use parameters β_{sim} and β_{rel} to restrict the number of statements considered for replacement and insertion, respectively. Table 6 shows the percentage reduction of statements for replacement/insertion at different $\beta_{\text{sim}}/\beta_{\text{rel}}$ values. From Table 6, just a small β_{sim} value can lead to a large reduction of replacement statements (e.g., 65% when $\beta_{\text{sim}} = 0.1$). When $\beta_{\text{rel}} \in [0.1, 0.5]$, the percentage reduction of insertion statements is between 30% and 40%. In our experiments, β_{sim} and β_{rel} are both



(a) Plausible fixing (b) Correct fixing

Figure 9: Venn diagram of repaired bugs.

set to 0.3, so we can discard 78% among those statements that are originally used for replacement and discard 38% for insertion.

Table 6: Percentage reduction of statements

Percentage Reduction	β_{sim} 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Replacement (%)	65	71	78	84	85	91	92	93	93
Percentage Reduction	**β_{rel} 0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
Insertion (%)	31	33	38	42	43	91	92	94	95

The above results only illustrate that the search space can be largely reduced by leveraging similarity and relevance. But it is unclear whether this strategy can still keep most of the useful statements that may constitute correct fixes. To investigate this question, we analyze the 17 correct patches found by ARJA-e-R, which in total contain 12 replacement edits and 8 insertion edits. We find that 11 out of the 12 replacements have a similarity larger than 0.3 and all 8 insertions have a relevance of no less than 0.5. This indicates that with a properly small β_{sim} and β_{rel}, our strategy usually does not ignore fix ingredients of correct repairs, while reducing the search space significantly, as shown in Table 6. Moreover, we analyze the 57 plausible but incorrect patches by ARJA-e-R that contain 66 replacement/insertion edits in total, and we find that 45.5% of them have a similarity/relevance of less than 0.3. So with β_{sim} and β_{rel} not too small, we can avoid a large portion of overfitting patches, thereby alleviating patch overfitting.

6 CONCLUSION

In this paper, we described ARJA-e, a new EC-based repair system, which incorporates the characteristic ideas of GenProg, PAR, ARJA and anti-patterns into a single repair framework, in order to take advantage of their respective component strengths. The evaluation on 224 real bugs from Defects4J shows that, compared to the current state of the art, ARJA-e can fix 70.8% more bugs, raising the number from 65 (achieved by Cardumen) to 111. It can correctly repair 46.2% more bugs from 26 (achieved by ELIXIR) up to 38. The results also show that ARJA-e can correctly fix several multi-location bugs, which is impossible for most existing repair approaches. Moreover, we experimentally show the benefits of combining the redundancy assumption and repair templates, and verify the effectiveness of a strategy for search space reduction. Our work indicates that the hybridization of existing repair techniques is very promising and might continue to produce further progress.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*. IEEE, 39–46.

[2] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*. ACM, 306–317.

[3] Markus F Brameier and Wolfgang Banzhaf. 2007. *Linear genetic programming*. Springer Science & Business Media.

[4] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 637–647.

[5] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *Proceedings of International Conference on Computer Aided Verification*. Springer, 383–401.

[6] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[7] Vidroha Debroy and W Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*. IEEE, 65–74.

[8] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation*. ACM, 947–954.

[9] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[10] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. 1345–1351.

[11] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. ACM, 437–440.

[12] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 295–306.

[13] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 802–811.

[14] John R Koza and John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.

[15] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFix: Semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th International Symposium on Software Testing and Analysis*. ACM, 376–379.

[16] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 593–604.

[17] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 213–224.

[18] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. 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*. IEEE, 3–13.

[19] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[20] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 727–739.

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

[22] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices* 51, 1 (2016), 298–312.

[23] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964.

[24] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 441–444.

[25] Matias Martinez and Martin Monperrus. 2017. Open-ended Exploration of the Program Repair Search Space with Mined Templates: the Next 8935 Patches for Defects4J. *arXiv preprint arXiv:1712.03854* (2017).

[26] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. 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, 492–495.

[27] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 448–458.

[28] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 691–701.

[29] Martin Monperrus. 2014. 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, 234–242.

[30] Martin Monperrus. 2018. Automatic software repair: A bibliography. *Comput. Surveys* 51, 1 (2018), 17.

[31] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 772–781.

[32] Vinicius Paulo L Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G Camilo-Junior. 2018. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering* 23, 5 (2018), 2980–3006.

[33] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 254–265.

[34] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. 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*. ACM, 24–36.

[35] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 404–415.

[36] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 648–659.

[37] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering*. ACM, 727–738.

[38] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. 2010. Automatic program repair with evolutionary computation. *Commun. ACM* 53, 5 (2010), 109–116.

[39] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th International Conference on Automated Software Engineering*. IEEE, 356–366.

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

[41] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 660–670.

[42] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 416–426.

[43] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55.

[44] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated repair of Java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2874648

[45] Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. 2011. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation* 1, 1 (2011), 32–49.