Chapter 26
Making Better Use of Repair Templates in Automated Program Repair: A Multi-Objective Approach

Yuan Yuan and Wolfgang Banzhaf

Abstract The automation of program repair can be coached in terms of search algorithms. Repair templates derived from common bug-fix patterns can be used to determine a promising search space with potentially many correct patches, a space that can be effectively explored by GP methods. Here we propose a new repair system, ARJA-p, extended from our earlier ARJA system of bug repair for JAVA, which integrates and enhances the performance of the first approach that combines repair templates and EC, PAR. Empirical results on 224 real bugs in Defects4J show that ARJA-p outperforms state-of-the-art repair approaches by a large margin, both in terms of the number of bugs fixed and of their correctness. Specifically, ARJA-p can increase the number of fixed bugs in Defects4J by 29.2% (from 65 to 84) and the number of correctly fixed bugs by 42.3% (from 26 to 37).

Key words: Program repair, evolutionary multi-objective optimization, genetic programming, repair templates

26.1 Introduction

Automated program repair [9, 32] aims to fix bugs in software automatically and has shown promise recently. Such techniques generate a patch for a bug that can satisfy a specification. Our study focuses on the test-suite based program repair in JAVA where the specification is given by a test suite.

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

© Springer Nature Switzerland AG 2020
W. Banzhaf et al. (eds.), *Evolution in Action: Past, Present and Future*,
Genetic and Evolutionary Computation, https://doi.org/10.1007/978-3-030-39831-6_26

A test suite should contain at least one initially failing, negative test case which triggers the bug to be repaired plus any number of initially passing, positive test cases that define the expected functionality of the program. In terms of test-suite driven repair, a bug is said to be *fixed* or *repaired*, if a patch can allow the modified program to pass the entire test suite. Such a patch is called *test-adequate* [26].

GenProg [20, 21] is among the most well-known approaches for test-suite based repair. This approach uses three types of statement-level mutations/edits (i.e., replace a destination statement with another, insert another statement before a destination or delete a destination statement) to rearrange the extant code of the buggy program. To explore the search space, GenProg uses genetic programming [3, 16] to search for potential patches that fulfill the test suite. However, due to the randomness of mutation operations, GenProg often generates patches which simply overfit the test suite [36]. To relieve this problem, Kim et al. [15] proposed PAR, an approach using common *fix patterns* (e.g., adding a null checker) manually learned from human-written patches. In PAR, a *repair template* is the central concept, a kind of program transformation schema derived from fix patterns. Unlike GenProg, PAR generates new program variants by using such predefined repair templates. To find a patch, PAR also employs an EC technique like GenProg, but focuses the search on more meaningful program transformations compared to GenProg, resulting in better chances to produce test-adequate or correct patches. PAR was the first repair approach that combined repair templates with EC techniques.

However, the performance of PAR is far from satisfactory. As reported by Le et al. [19], a reimplementation of PAR for JAVA can only fix very few bugs correctly in the Defects4J [13] dataset. Note that there are generally two key elements in a successful repair approach: the search space and the search algorithm [36]. The search space should contain as many correct patches as possible while the search algorithm should be powerful enough to explore such a large search space. PAR has limitations in both regards. Regarding the search space, although the templates used in PAR define potentially useful program transformations, they are often not sufficiently general and cover only a very limited number of fix patterns. Regarding the search algorithm, PAR uses an EC framework similar to that of GenProg, which recombines and mutates high-granularity edits via crossover and mutation operators. Recent studies [34, 45] have shown, however, that evolving such high-level units strongly limits the ability to effectively traverse a search space, which may be a reasons why GenProg usually generates patches that are equivalent to a single functionality deletion [36].

Our work is motivated by recent progress on both issues. Several repair approaches such as ELIXIR [38], SPR [24] or Cardumen [28] use a richer set of repair templates than PAR to generate program variants, achieving state-of-the-art performance on well-known datasets of bugs. Also, very recent studies [34, 45] suggest that evolving patches with lower-granularity patch representations via advanced EC techniques can lead to a substantially improved search ability.

Given the above, we want to exploit the benefits of both, recent template based approaches (they work over a richer and more promising search space) and EC approaches, more powerful than previous ones, in order to improve over the PAR

algorithm. We hence develop a new repair approach for Java, extended from ARJA [45], called ARJA-p. In ARJA-p, the repair templates in PAR are made more general to cover a larger set of fix patterns for potential use in patches. To bridge the gap between template based edits (usually in the expression-level) and the lower-granularity patch representation of ARJA (only applies to statement-level edits), we execute the templates offline, thereby abstracting various template based edits into two types of statement-level edits (i.e., replacement and insertion). We can then introduce a lower-granularity patch representation for template based repair. Lastly, we formulate program repair as a multi-objective search problem and use a classical multi-objective evolutionary algorithm (i.e., NSGA-II [8]) to search for simpler patches, following the paradigm of search-based software engineering [2, 11, 12].

ARJA-p is evaluated on 224 real bugs in Defects4J [13] and compared to other state-of-the-art approaches. We can show that ARJA-p outperforms all the other approaches with a significant margin. Overall, ARJA-p is able to increase the number of bugs fixed in Defects4J by 29.2% (from 65 to 84) and the number of correctly fixed bugs even more, by 42.3% (from 26 to 37). Notably, ARJA-p can correctly fix several multi-location bugs, impossible for most of the existing repair approaches.

The rest of this paper is structured as follows. Section 26.2 provides background knowledge and a motivating example. Section 26.3 describes our repair approach in detail. Section 26.4 presents the experimental design and Section 26.5 reports our empirical results.

26.2 Background and Motivation

Search-based repair approaches determine a search space potentially containing correct patches and employ metaheuristic search techniques or random search to find test-adequate patches.

GenProg [20, 21] is a representative approach that uses genetic programming (GP) to search for test-adequate patches. This approach is based on the redundancy assumption [4, 29] (i.e., the ingredients for a fix exist elsewhere in the current program) and performs three kinds of statement-level edits, replacement, insertion and deletion. [22] studied the influence of different solution representations and genetic operators in GenProg; [35] presents RSRepair that replaces GP in GenProg with random search; [39] suggested a set of anti-patterns to inhibit GenProg from generating nonsensical patches; [34] introduce a lower-granularity patch representation and several related crossover operators. PAR [15] is a pioneering approach that exploits repair templates to generate bug fixes. ARJA-p extends repair templates iof PAR to accommodate more useful fix patterns and conducts more effective search via evolutionary multi-objective optimization. ARJA [45] is a GP based approach for JAVA, characterized by a novel patch representation, multi-objective search and several auxiliary techniques for speeding up fitness evaluation and reducing the search space. Unlike ARJA, which follows the redundancy assumption, ARJA-p uses templates to generate fix ingredients either for replacement or for insertion and puts

them into two separate evolvable segments. Other typical search-based approaches include AE [40], SPR [24], HDRepair [19], ACS [42], ssFix [41] and Cardumen [28].

Besides search-based approaches, semantics-based approaches [7, 14, 17, 30, 31, 33, 43] are other techniques that have been extensively studied. There, semantic constraints are inferred from the given test-suite which are then used to generate test-adequate patches by solving the resulting constraint satisfaction problem. Very recently, some emerging techniques (e.g., deep learning) have been introduced into program repair, leading to several novel repair systems [6, 10, 25, 23, 38].

Another line of research focuses on the empirical aspects of program repair, including the problem of patch overfitting [36, 44], performance evaluation of different repair systems [18, 26] and analysis of real-world bug fixes [27, 48].

26.2.1 Brief Introduction to PAR

PAR [15] is an automatic program repair technique based on repair templates. Like GenProg, PAR takes a buggy program and a test suite with at least one negative test as input, with the goal to find a patch that allows all test cases to pass. Unlike GenProg which uses random statement replacement, as well as insertion and deletion to edit a program, PAR exploits repair templates to generate new program variants. Each repair template represents a common way to fix a specific kind of bug. For example, a specific bug is the access to a `null` object reference, and a common fix is to add `if` statement to check whether the object is `null` (this template is called "Null Pointer Checker" in PAR). PAR collects 10 repair templates by manually inspecting human-written patches and adopts an evolutionary process to use these templates.

The overall procedure of PAR is summarized as follows: First, PAR uses a simple fault localization strategy to find a number of suspicious statements. Its evolutionary process starts with an initial population of program variants iterating through two tasks: reproduction and selection. In reproduction, each program variant derives a new one by applying templates to the selected suspicious statements. In the selection, a tournament selection scheme chooses better (in terms of passing tests) program variants for the next generation. Iterations are stopped when a program variant passes all the tests.

From an EC perspective, PAR uses an evolutionary process similar to that in GenProg, but PAR only relies on template-based mutation and does not use any crossover at all. In other words, individual programs in the population of PAR do not exchange information with each other, so good genetic material cannot be propagated from one individual to another.

26.2.2 A Motivating Example

In this subsection, we take a bug as an example to highlight the key insights underlying ARJA-p which motivates our algorithm design.

Figure 26.1 (a) shows a correct patch for the real bug Math98 in Defects4J [13]. The two methods `operate(BigDecimal[] v)` and `operate(double[] v)` implement similar functionality: multiply the current matrix $\mathbf{A}_{m \times n}$ by a n-dimensional vector $\mathbf{x}$ (stored in the array `v`) and return the product $\mathbf{Ax}$ that is a m-dimensional vector. However, in the buggy program, a vector with size n (i.e., `v.length`) is used to store the m-dimensional vector $\mathbf{Ax}$ by mistake. To correctly fix the bug, `v.length` in lines 4 and 11 should both be changed to m (represented by the variable `nRows` in the code).

Based on two modifications in Figure 26.1 (a), if we make an additional modification as shown in Figure 26.1 (b), the synthesized patch (containing three edits) can still pass the whole test suite but is indeed incorrect. To understand this, we have to look at the meaning of line 3 in Figure 26.1 (b): This line checks whether the current matrix $\mathbf{A}$ can be multiplied by vector $\mathbf{x}$. Obviously, the only requirement is that the column dimension of $\mathbf{A}$ is equal to the vector dimension of $\mathbf{x}$. The original program checked this correctly, but the test-adequate (but incorrect) patch adds a further condition `isSingular()` (lines 4–5) that judges whether matrix $\mathbf{A}$ is singular. The patched program introduces some unexpected program behavior: if $\mathbf{A}$ is not singular, it can be multiplied by vector $\mathbf{x}$ with any dimension. However, this behavior cannot be detected by the test suite associated with Math98.

```
1  // BigMatrixImpl.java
2  public BigDecimal[] operate(BigDecimal[] v) throws ... {
3  ...
4  -   final BigDecimal[] out = new BigDecimal[v.length];
5  +   final BigDecimal[] out = new BigDecimal[nRows];
6  ...
7  }
8  // RealMatrixImpl.java
9  public double[] operate(double[] v) throws ... {
10 ...
11 -   final double[] out = new double[v.length];
12 +   final double[] out = new double[nRows];
13 ...
14 }
```

(a) Correct patch

```
1  // BigMatrixImpl.java
2  public BigDecimal[] operate(BigDecimal[] v) throws ... {
3  -   if (v.length != this.getColumnDimension()) {
4  +   if (v.length != this.getColumnDimension() &&
5  +         isSingular()) {
6            throw new IllegalArgumentException("...");
7      }
8  ...
9  }
```

(b) An additional modification

Fig. 26.1: Correct patch and test-adequate patch for Math98.

We can make three observations using this example:

(1) The bug fixing Math98 needs to replace a qualified name (i.e., `v.length`) with a variable `nRows`. However, this fix pattern cannot be handled by any of the 10 repair templates defined in PAR [15], because such a replacement in PAR can only be applied to a method parameter whereas `v.length` is an array index.

(2) The correct patch shown in Figure 26.1(a) requires multi-line changes that fix multiple buggy locations. A single modification in either line 4 or 11 cannot produce

a functionally correct program. To repair such multi-location bugs is hard or even impossible for almost all existing repair approaches. Since PAR uses an evolutionary framework similar to GenProg it can change multiple locations of a program in principle. However, [36] showed that seemingly complex patches generated by such search mechanisms are equivalent to single line modifications in the overwhelming majority of cases. PAR can only fix 4 bugs correctly in Defects4J, none of which was claimed to be a multi-location bug [19].
(3) As shown in Figure 26.1(b), an additional modification will turn the correct patch into a test-adequate but incorrect one. This implies that simpler or smaller patches should be preferred. With a weak test suite, looking for a smaller patches avoids making undesirable modifications. However, most of existing repair approaches including PAR do not explicitly take simplicity of a patch into account.

Here we aim to improve template-based repair approaches, particularly PAR, in the following way. First, we extend the repair templates used in PAR so as to make them more general and accommodate more fix patterns. Second, we propose a new evolutionary framework with a lower-granularity patch representation for template-based repair, which allows to fix multi-location bugs by leveraging its stronger search ability. Third, we formulate program repair as a multi-objective search problem and use evolutionary multi-objective optimization techniques to discover smaller patches.

26.3 Approach

The input of ARJA-p is a buggy program associated with a number of JUnit tests. Among these tests, there is at least one negative test. The others are positive tests defining the expected program functionality. The basic goal of ARJA-p is to modify a buggy program so as to allow it to pass all tests cases. ARJA-p consists of the following four main steps.

(1) **Fault Localization:** Given an input, ARJA-p first applies a spectrum-based fault localization technique called Ochiai [1] to locate a list of likely-buggy statements (LBSs). Each LBS is given a suspiciousness value $susp \in [0,1]$ by Ochiai, which indicates the likelihood of this LBS to contain the bug. In ARJA-p, we only consider some of the LBSs returned by fault localization in order to reduce the search space. This is determined by two parameters denoted by $\gamma_{\min}$, the minimal suspiciousness value, and $n_{\max}$, the maximal number of LBSs.

(2) **Generating Potential Fix Ingredients:** After fault localization, we apply the predefined repair templates to each LBS in the reduced set. By executing the transformations specified by repair templates each LBS can derive a number of new statements, which we call *ingredient statements* to be either inserted before or replacing the corresponding LBS.

(3) **Test Filtering:** Before entering into the evolutionary search, we conduct coverage analysis to filter out those positive test cases that are not influenced by the manipulation of selected LBSs. Specifically, we run the positive tests one by one,

and record the statements visited by this test. If none of the LBSs is touched, this positive test can be ignored. Thus, modified program variants can be validated on a reduced test suite, which speeds up fitness evaluation in our MOEAs.

(4) **Searching Test-Adequate Patches:** Now that we have selected a number of LBSs, each of which with a number of ingredient statements for replacement or insertion we try to find test-adequate patches consisting of replacement/insertion edits, with smaller patches preferred. In ARJA-p, we formulate this problem as a multi-objective combinatorial optimization/search problem and use MOEAs to explore the search space.

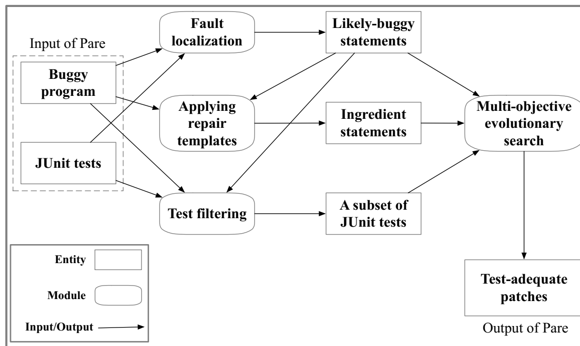


Fig. 26.2: Overview of our repair approach (i.e.,ARJA-p).

Figure 26.2 shows an overview of the proposed repair approach. In the remainder of this section, we detail the second and fourth steps (i.e., how to apply the repair templates to generate potential fix ingredients and how to evolve the patches via MOEAs), which are two characteristic procedures in ARJA-p.

26.3.1 Repair Templates

ARJA-p uses 7 repair templates that are mainly derived from templates in PAR. Each template specifies a type of transformation of code.

(1) **Element Replacer:** This template replaces an abstract syntax tree (AST) node element in a LBS with another compatible one. Table 26.1 lists the elements that can be replaced and illustrates alternative replacers for each kind of element. Note that templates "Parameter Replacer" and "Method Replacer" used in PAR are just a subset of a template here. Some replacement rules shown in Table 26.1 with a widened type follow, e.g. rules from ELIXIR [38] or REFAZER [37].

(2) **Method Parameter Adder or Remover:** This template is applicable for the method invocation that has overloaded methods. To restrict the complexity, ARJA-p only considers to add or remove a single element from the parameter list of the

Table 26.1: 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`

current method invocation. The order of current parameters can be rearranged provided that their types are compatible to the corresponding types declared in the overloaded method. When adding a parameter, ARJA-p collects all fields and local variables within the scope of an LBS's location, and candidates to be added must be type-compatible with the corresponding parameter type in the method declaration.

(3) **Boolean Expression Adder or Remover:** This template is applicable for an LBS that has a conditional branch. Take the `if` statement as example and suppose the LBS is like `if (c1 && c2){...}`. When adding a boolean expression in the predicate, this template collects all boolean expressions that are in the same file with the LBS, and those within the scope of the LBS's location can be alternatives to be added. Suppose `c3` is chosen, ARJA-p uses the following four transformation schemas to edit the predicate in the LBS: (i) `c1 && c2 && c3` (ii) `c1 && c2 || c3` (iii) `c1 && c2 && !c3` (iv) `c1 && c2 || !c3`. When removing a term in the predicate, ARJA-p can select any one (e.g., `c1` or `c2`) to remove.

Note that ARJA-p extends this template from PAR in two ways: The added boolean expression can be any one from the file that meets the scope, not just the one in the predicate. Second, ARJA-p adopts more transformation schemata.

(4) **Null Pointer Checker:** For a LBS, this template first extracts all objects in the statement that have object references (e.g., `o1` and `o2`). Then it creates a predicate (e.g., `o1!= null && o2 != null`) to assure that all these objects cannot be `null` when executing the LBS. With this predicate, ARJA-p uses the following 6 transformation schemata to manipulate the LBS.

(i) `if (o1 != null && o2 != null)buggyStatement;`
(ii) `if (!(o1 != null && o2 != null))return sth;`
(iii) `if (!(o1 != null && o2 != null))throw exception;`
(iv) `if (!(o1 != null && o2 != null))break;`
(v) `if (!(o1 != null && o2 != null))continue;`
(vi) `if (o1 == null)o1 = new Obj1();`
`if (o2 == null)o2 = new Obj2();`

The first schema makes the LBS a part of the `if` statement whereas each of the others inserts the entire `if` statement before the LBS. The second and third schemata need to be instantiated since they require another `return` and `throw` statement, respectively. For `return` statements, the return type of the method containing the LBS is first checked, then the corresponding values according to this type are returned: (i) `boolean`: return true or false; (ii) `void`: return nothing; (iii) the other primitives: return 0 or 1; (iv) an object: return null. ARJA-p always uses the last `return` statement in the method. As for `throw` statements, ARJA-p collects alternative thrown exceptions in three ways: (i) find the method declaration where the LBS located and use its declared thrown exception types; (ii) if the considered objects are in method parameters, consider the `IllegalArgumentException`; (iii) consider the exception types defined in the buggy program that are extended from the `NullPointerException`. The last schema uses the basic constructor to initialize `null` objects having references.

To avoid compile errors, sometimes not all 6 schemata are applicable. Specifically, the first schema can only be applied to an LBS which is not a variable declaration statement and the fourth and fifth schema can only be used if the LBS is in a `for` or `while` loop.

We introduce three further repair templates that are similar to "Null Pointer Checker". They use the same 6 schemata mentioned above to edit a buggy program, but their predicates in the `if` statement check different contexts.

(5) **Range Checker:** This template mainly checks whether all array or list element accesses are valid in an LBS (i.e., indices cannot exceed the upper and lower bounds of the size of an array or list). Different from PAR, it also considers to check the validity of `char` access (in the form of `charAt` or `substring`) for `String` objects since `String` is a list of characters and is frequently used in Java.

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

(7) **Divide-by-Zero Checker:** This template checks whether all the divisors are not equal to 0. It is not used in PAR.

26.3.2 Offline Execution of Templates

As described in Section 26.3.1, repair templates in ARJA-p enrich those used in PAR, so that ARJA-p can potentially handle more fix patterns. Our system exploits repair templates in a way different from PAR and other related approaches [19, 37, 41]. PAR executes templates *on-the-fly* (i.e., during the evolutionary process), whereas our system uses them *offline*. Specifically, ARJA-p executes all possible transformations defined by templates for all considered LBSs *before* searching for patches. Each LBS can derive a number of new statements and each of these statements can either replace the LBS or be inserted before it. These template based repair actions (some occur in the AST node level) are abstracted into two kinds of statement-level edits (i.e., replacement and insertion). Figure 26.3 illustrates this abstraction for a supposed LBS of `a.add(x,y)`. In order to avoid combinatorial explosion, ARJA-p only applies a template to a single node at a time. For example, ARJA-p does not use the template "Element Replacer" to simultaneously change `a` and `add` in `a.add(x,y)`.

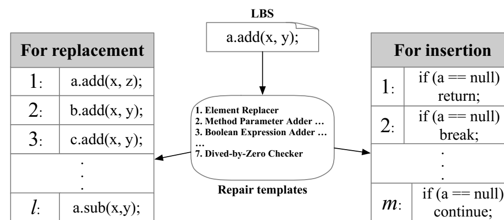


Fig. 26.3: Illustration of the offline template execution.

After offline template execution, repair templates are invisible to the evolutionary search in ARJA-p. Only two kinds of statement-level edits are visible to the search.

26.3.3 Evolving Patches

Suppose n likely-buggy statements (LBSs) are selected by fault localization, with each of them having two sets of ingredient statements (one for replacement and another for insertion) that are produced through the offline execution of templates. We can then view template based patch generation as a three-level decision process: (1) Choose to edit some statements among n LBSs; (2) Select which operation ("replace" or "insert before") to apply for each LBS to be edited; (3) Choose statements from the corresponding ingredient statements for replacement/insertion. ARJA-p uses a lower-granularity patch representation that properly decouples the search subspaces of potentially buggy locations, operation types and replacement/insertion statements. Based on this representation, we can evolve patches via multi-objective

evolutionary algorithms (MOEAs) that effectively explore this search space. The details are described as follows.

Patch Representation

To encode a patch as a MOEA individual, we first sequence the n LBSs in a random order. For the j-th LBS, $j = 1, 2, \dots, n$, its associated two sets of ingredient statements are denoted as R_j (for replacement) and I_j (for insertion) respectively, and the statements in R_j/I_j are numbered from 1 to $|R_j|/|I_j|$ in any order. These ID numbers are fixed throughout the evolutionary process.

A solution (i.e., a patch) to the program repair problem is represented as a colated vector $\mathbf{x} = (\mathbf{b}, \mathbf{c}, \mathbf{u}, \mathbf{v})$,with four parts each being a vector of size n itself. $\mathbf{b} = (b_1, b_2, \dots, b_n)$ is a binary vector, where $b_j \in \{0, 1\}$, $j = 1, 2, \dots, n$ indicates whether the j-th LBS is to be edited or not. $\mathbf{c} = (c_1, c_2, \dots, c_n)$ is also a binary vector, where $c_j = 0$ ($c_j = 1$) means the "replace" ("insert before") operator is chosen for the j-th LBS. $\mathbf{u} = (u_1, u_2, \dots, u_n)$ is an integer vector and u_j indicates that patch $\mathbf{x}$ selects the u_j-th ingredient statement in the set R_j. Similar to $\mathbf{u}$, $\mathbf{v} = (v_1, v_2, \dots, v_n)$ is also an integer vector in which v_j means the v_j-th statement in the set I_j is selected by patch $\mathbf{x}$. Suppose the j-th LBS is `a.add(x,y)`, Figure 26.4 illustrates the patch representation in ARJA-p. Figure 26.5 describes how to apply a patch $\mathbf{x}$ to the buggy program (i.e., decoding procedure) so as to obtain a modified program.

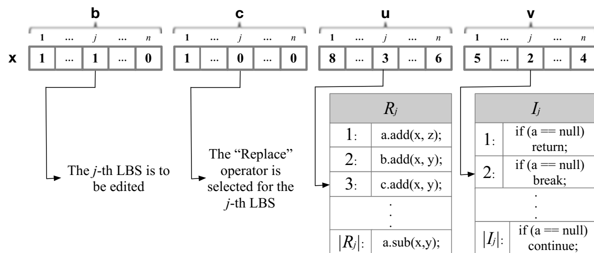


Fig. 26.4: Illustration of the patch representation in ARJA-p.

Fitness Function

The automated program repair is formuated as a multi-objective search problem. To evaluate the fitness of each individual $\mathbf{x}$, we employ a multi-objective function to simultaneously minimize the *patch size* (denoted by $f_1(\mathbf{x})$) and a *weighted failure rate* (denoted by $f_2(\mathbf{x})$). Patch size is defined as $f_1(\mathbf{x}) = \sum_{i=1}^{n} b_i$, refering to the number of edits contained in the patch. Weighted failure rate is defined as

```
Input: A patch x = (b, c, u, v); the buggy program; n LBSs; .
Output: A modified program.
1 for j = 1 to n do
2     if b_j = 1 then
3         st ← the j-th LBS;
4         if c_j = 0 then
5             st* ← the u_j-th statement in the set R_j;
6             Replace st with st*;
7         else
8             st* ← the v_j-th statement in the set I_j;
9             Insert st* before st;
```

Fig. 26.5: The procedure to apply a patch.

$$f_2(\mathbf{x}) = \frac{|\{t \in T_f \mid \mathbf{x} \text{ fails } t\}|}{|T_f|} + w \times \frac{|\{t \in T_c \mid \mathbf{x} \text{ fails } t\}|}{|T_c|} \tag{26.1}$$

where T_f is the set of negative test cases, T_c is the reduced set of positive test cases, and $w \in (0,1]$ is a global parameter used to emphasize the passing of negative test cases. $f_2(\mathbf{x})$ measures how well the modified program behaves in terms of passing the given test cases. $f_2(\mathbf{x}) = 0$ means $\mathbf{x}$ does not fail any test case and is thus a test-adequate patch.

By simultaneously minimizing these two objectives, we introduce search bias toward smaller patches. Note that if the modified program fails to compile or runs out of time, $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$ are both set to $+\infty$. Moreover, $f_1(\mathbf{x}) = 0$ is meaningless since no modifications would be made to the buggy program. Should $f_1(\mathbf{x})$ reach 0, we reset both $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$ to $+\infty$, in order to eliminated this solution by selection.

Population Initialization

We initialize the population by combining prior knowledge and randomness: For each individual **x**, **b** is initialized using fault localization information. Suppose $susp_j$ is the suspiciousness of the j-th LBS, then b_j is initialized to 1 with probability $susp_j \times \mu$ and to 0 with $1 - susp_j \times \mu$, where $\mu \in (0,1)$ is a predefined parameter. The remaining three parts are randomly initialized.

Genetic Operators

Genetic operators crossover and mutation are used to produce offspring in MOEAs. In ARJA-p, crossover and mutation are applied to each part of the patch representation separately, in order to inherit good traits from parents. For **b** and **c** we employ half uniform crossover (HUX) and bit-flip mutation. For **u** and **v**), we use single-point crossover and uniform mutation due to their integer nature. Figure 26.6 demonstrates how crossover and mutation are performed on two parents.

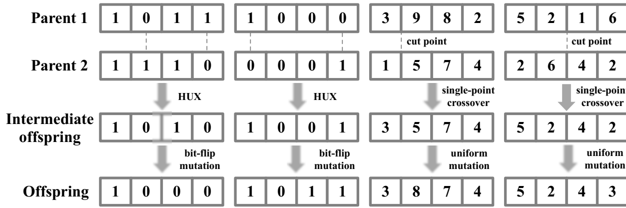


Fig. 26.6: Illustration of the crossover and mutation in ARJA-p. Only one offspring shown.

Multi-Objective Search

As a selection (including mating and environmental selection) framework, we use NSGA-II [8], a classical Pareto dominance-based MOEA. Note that recently proposed MOEAs like θ-DEA [46] and MOEA/D-DU [47] can also be used here.

The NSGA-II search procedure works as follows: First, an initial population with N (the population size) individuals is generated using the initialization strategy introduced in Section 26.3.3. Then the algorithm iterates over generations until a termination criterion is met. In the g-th generation, binary tournament selection [8] and the genetic operators of Section 26.3.3 are applied to the current population P_g so as to create an offspring population Q_g of size N. Then the fast non-dominated sorting and crowding distance comparison [8] (based on two objectives formulated in Section 26.3.3) are used to select the best N individuals from the union population $P_g \cup Q_g$, arriving at the next population P_{g+1}.

When the search is terminated, the obtained non-dominated solutions with $f_2 = 0$ are output as test-adequate patches found by ARJA-p. If there is no such solution, ARJA-p has failed to repair the bug.

26.4 Experimental Design

Our evaluation is conducted on four open-source Java projects (i.e., Chart, Time, Lang and Math) from Defects4J [13], a database widely used for evaluating Java repair systems [6, 19, 26, 28, 38, 41, 42, 45]. Table 26.2 shows the the basic information of the four projects. There are 224 real-world bugs in total: 26 bugs from Chart (C1–C26), 27 bugs (T1–T27) from Time, 65 bugs (L1–L65) from Lang and 106 bugs (M1–M106) from Math. Note that Defects4J indeed contains another project, namely Closure. We do not use Closure because its customized testing format is incompatible with GZoltar [5], a third-party fault localization tool used in our repair system.

Table 26.3 shows the basic parameter setting for ARJA-p in our empirical study, where n is the number of LBSs determined by $\gamma_{\min}$ and $n_{\max}$ together (see Section 26.3). We run 5 random trials in parallel for each bug. Each trial is terminated after

Table 26.2: The descriptive statistics of Defects4J dataset

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

3 hours following the practice in refs. [26, 28]. Our experiments are performed on HPC machines with 2.4 GHz Intel Xeon E5 Processors and 20 GB memory.

Table 26.3: The parameter setting for ARJA-p in the experiments

Parameter	Description	Value
N	Population size	40
$\gamma_{\min}$	Threshold for the suspiciousness	0.1
$n_{\max}$	Maximum number of LBSs considered	60
w	Refer to Section 26.3.3	0.5
μ	Refer to Section 26.3.3	0.06
p_c	Crossover probability	1.0
p_m	Mutation probability	$1/n$

26.5 Results and Discussions

Table 26.4 shows the bugs in Defects4J that can be fixed (i.e., test-adequate patches are found) and correctly fixed by ARJA-p. Note that only non-dominated solutions with $f_2 = 0$ are meaningful for program repair. Among all 224 bugs considered, ARJA-p can generate test-adequate patches for 84 bugs.

A major concern raised recently [36] was whether test-adequate patches are correct beyond passing the test suite. Following previous work [26, 38, 41, 42, 45], we manually checked the correctness of these patches found by ARJA-p. A test-adequate patch is deemed as *correct* if it is exactly the same as or semantically equivalent to a human-written patch. After a careful manual study, we confirmed that ARJA-p found correct patches for 37 bugs.

We compare ARJA-p with 11 state-of-the-art repair tools in terms of the number of bugs fixed (i.e., test-adequate) and correctly fixed. The 11 tools for comparison are jGenProg [26] (an implementation of GenProg for Java), jKali [26] (an implementation of Kali for Java), xPAR (a reimplementation of PAR by Le et al. [19]), Nopol [26, 43], HDRepair [19], ACS [42], ssFix [41], JAID [6], ELIXIR [38],

Table 26.4: The bugs for which the test-adequate patches and the correct patches are synthesized by ARJA-p

Project	Test-Adequate	Correct
Chart	C1, C3, C4, C5, C7, C10, C11, C13, C14, C15, C17, C19 ,C24, C25, C26	C1, C4, C10, C11, C14, C17,C19, C24
	$\sum = 15$	$\sum = 8$
Time	T4, T11, T14, T17, T20	T4
	$\sum = 5$	$\sum = 1$
Lang	L7, L16, L20, L21, L22, L24, L27, L33, L34, L39, L41, L44, L45, L47, L50, L51, L57, L58, L59, L60, L61, L63	L20, L24, L33, L34, L39, L47, L57, L59, L61
	$\sum = 22$	$\sum = 9$
Math	M2, M3, M5, M6, M7, M22, M24, M25, M28, M30, M32, M34, M40, M42, M49, M50, M56, M57, M58, M62, M63, M65, M70, M71, M73, M75, M77, M78, M79, M80, M81, M82, M84, M85, M88, M89, M94, M95, M96, M98, M104, M105	M5, M22, M25, M30, M34, M56, M57, M58, M70, M75, M79, M80, M82, M89, M94, M98, M105
	$\sum = 42$	$\sum = 19$
Total	84 (37.5%)	37 (16.5%)

ARJA [45] and Cardumen [28], which cover almost all the tools that have ever been tested on Defects4J. Table 26.5 shows our comparison results. Note that for xPAR, HDRepair and Cardumen, some results were not reported by the original authors. ARJA-p performs best and indeed outperforms all 11 other techniques by a large margin, both in terms of the number of bugs fixed and correctly fixed. Specifically, ARJA-p is able to increase the highest number of fixed bugs in Defects4J by 29.2%, from 65 (achieved by Cardumen) to 84; it increases the highest number of correctly fixed bugs in Defects4J by an even larger margin, 42.3%, from 26 (achieved by ELIXIR) to 37. Moreover, xPAR can only correctly fix 3 bugs, which is much less than the number achieved by ARJA-p. Since our ARJA-p is based on PAR, the much improved performance of ARJA-p strongly demonstrates the improvements over PAR.

Figure 26.7(a) presents a Venn diagram showing the intersections of fixed bugs among ARJA-p, ARJA and Cardumen. We select ARJA and Cardumen since they are the best-performing tools among the 11 existing ones in terms of producing test-adequate patches. ARJA-p is able to fix 22 bugs that neither ARJA nor Cardumen could fix. But although ARJA-p fixes more bugs, ARJA and Cardumen can also

Table 26.5: Comparison with 11 techniques in terms of the number of bugs fixed and correctly fixed (Test-Adequate/Correct).

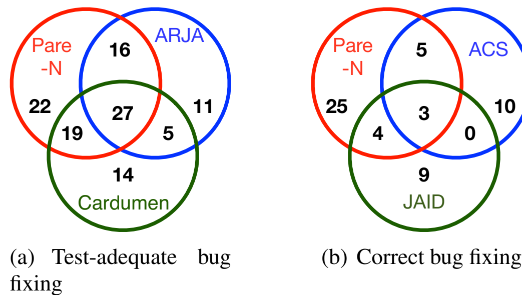
Project	ARJA-p	jGenProg	jKali	xPAR	Nopol	HDRepair
Chart	**15/8**	7/0	6/0	NA/0	6/1	NA/2
Time	**5**/1	2/0	2/0	NA/0	1/0	NA/1
Lang	**22/9**	0/0	0/0	NA/1	7/3	NA/7
Math	**42/19**	18/5	14/1	NA/2	21/1	NA/6
Total	**84/37**	27/5	22/1	NA/3	35/5	NA/16

Project	ACS	ssFix	JAID	ELIXIR	ARJA	Cardumen
Chart	2/2	7/2	4/4	7/4	9/3	15/NA
Time	1/1	4/0	0/0	3/**2**	4/1	6/NA
Lang	4/3	12/5	8/5	12/8	17/4	7/NA
Math	16/12	26/7	8/7	19/12	29/10	37/NA
Total	23/18	49/14	20/16	41/26	59/18	65/NA

"NA" means the data is not available.

fix 16 and 19 bugs, respectively, that cannot be fixed by ARJA-p. This indicates that state-of-the-art repair approaches are complementary to each other, and further attempts at combinations should be envisioned.

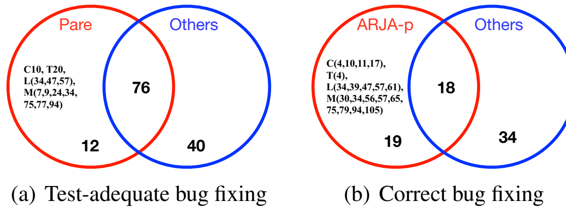
Figure 26.7(b) shows a Venn diagram to compare ARJA-p with two other state-of-the-art approaches, ACS and JAID, in terms of correct bug fixing. Note that the ELIXIR paper reported the highest number of bugs correctly fixed in the literature but did not show which bugs were correctly fixed, so we cannot compare with it here. The Figure shows that ARJA-p, ACS and JAID can uniquely generate a correct patch for 25, 10 and 9 bugs, respectively, compared to their counterparts. Again, these systems exhibit good complementarity in terms of producing correct bug fixes.



(a) Test-adequate bug fixing

(b) Correct bug fixing

Fig. 26.7: Venn diagram of bugs for which test-adequate patches (ARJA-p, ARJA and Cardumen) and correct patches (ARJA-p, ACS and JAID) are found.



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

Fig. 26.8: Venn diagram of bugs for which test-adequate patches and correct patches are found. Our results are compared with the combined results of all the other techniques mentioned in Table 26.5.

Finally, we compare the results of ARJA-p with the combined results of all 11 existing techniques. Figure 26.8 shows that our technique can uniquely find test-adequate patches for 12 bugs and uniquely generate correct patches for 20 bugs. To our knowledge, each of these 12 bugs (listed in Figure 26.8(a)) is fixed for the first time, and each of 20 bugs (listed in Figure 26.8(b)) is repaired correctly for the first time.

26.5.1 Results on Multi-Location Bugs

Among the 37 bugs correctly fixed by ARJA-p, C14, C19, L20, L34, L47, L61, M22 and M98 are deemed multi-location bugs, since the correct patch by ARJA-p for these 8 bugs contains at least two edits at multiple buggy locations.

```
1  // ToStringStyle.java
2  static Map<Object, Object> getRegistry() {
3  - return REGISTRY.get() != null ? REGISTRY.get() :
4  -       Collections.<Object, Object>emptyMap();
5  + return REGISTRY.get();
6  }
7  static boolean isRegistered(Object value) {
8    Map<Object, Object> m = getRegistry();
9  + if (!(m != null)) return false;
10   return m.containsKey(value);
11 }
```

Fig. 26.9: Correct patch generated by ARJA-p for the bug L34.

For M22 and M98, ARJA-p can synthesize a correct patch that is exactly the same as a human-written patch. As for the remaining 6 bugs, ARJA-p generates a correct repair semantically equivalent to a human-provided one. It is quite challenging for correctly fixing these bugs. For example, for C14, ARJA-p generates a correct patch that executes the “Null Pointer Checker” template at 4 different LBSs (located in two different Java files). Another interesting example is fixing bug L34, which is shown

in Figure 26.9. To correctly fix this bug, ARJA-p uses two kinds of templates for two LBSs: "Element Replacer" for lines 3–4 and "Null Pointer Checker" for line 10. A human-written patch for L34 differs in that it replaces line 10 with `return m != null && m.containsKey(value);`. Obviously, this modification is functionally equivalent to the null pointer check done by ARJA-p. Note that L61 can also be seen as a single-location bug in terms of a human-written patch that modifies only a single LBS. However, that patch is not in the search space of ARJA-p.

To our knowledge, only two existing approaches (i.e., ACS [42] and ARJA [45]) reported correct fixes for multi-location bugs on Defects4J. For the 8 multi-location bugs correctly fixed by ARJA-p, ACS can only generate correct patches for two of them (i.e., C14 and C19), while ARJA can generate three (i.e., L20, M22 and M98). Bugs L34, L47, L61 are correctly fixed by ARJA-p for the first time. The strength of ARJA-p in fixing multi-location bugs demonstrates the power of evolutionary multi-objective search.

26.5.2 Contribution of Repair Templates

ARJA-p uses 7 repair templates. It is interesting to understand the contribution of each template on the number of bugs fixed (test-adequate or correctly), which should provide insight into the strength and weakness of ARJA-p. If multiple patches are obtained for a bug, we just randomly choose one for analysis. Note that a bug fix may involve more than one template (e.g., L34 in Figure 26.9).

Table 26.6 summarizes the contribution of templates. It is clear that "ER" is the most useful template for the performance of ARJA-p on Defects4J. This template contributes to the generation of a test-adequate patch for 54 bugs and a correct one for 27 bugs. "BEAR" helps to synthesize a test-adequate patch for 17 bugs (just behind "ER"), however only 1 of them is identified as correct. "MPAR" and "NPC" make moderate contribution for both test-adequate and correct bug fixing. "DC", "CC" and "RC" do not contribute much to ARJA-p's performance on Defects4J, and "RC" even contributes nothing.

Table 26.6: Contribution of each repair template

Template	Test-Adequate	Correct
Element Replacer (ER)	54	27
Method Parameter Adder or Remover (MPAR)	6	3
Boolean Expression Adder or Remover (BEAR)	17	1
Null Pointer Checker (NPC)	10	7
Range Checker (RC)	0	0
Cast Checker (CC)	2	1
Divide-By-Zero Checker (DC)	1	1

It is not surprising that "ER" contributes most since it can manipulate many more types of AST nodes than other templates. "BEAR" tries to synthesize a condition by exploiting the intrinsic redundancy of a program, so the synthesized condition may not be so accurate. This could be the reason that most of the test-adequate patches contributed by this template are indeed incorrect. A promising way to enhance the strength of "BEAR" is to incorporate a precise condition synthesis technique like that in ACS [42]. "NPC" contributes much more than three similar templates (i.e., "RC", "CC" and "DC"), implying that the null pointer bug could be quite common in Java. Moreover, the patch produced by "NPC" has a relatively high probability to be correct, may be because the manipulations performed by "NPC" are usually harmless. Note that although "RC", "CC" and "DC' have limited contribution here, they could be helpful for ARJA-p on other bug datasets.

26.5.3 Value of Test Filtering

We select the latest buggy versions of the four projects considered in Defects4J (i.e., C1, T1, L1 and M1) as the subject programs to examine the effect of test filtering. For each buggy program, we vary $\gamma_{\min}$ (i.e., the threshold of suspiciousness) from 0 to 0.2. For each $\gamma_{\min}$ value chosen, we use our test filtering procedure to obtain a subset of original JUnit tests and record two metrics associated with this reduced test suite: the number of tests and the execution time. Fig. 26.10 shows the influence of $\gamma_{\min}$ on the two metrics for each subject program. For comparison purposes, the two metrics have been normalized by dividing by the original number of JUnit tests and the CPU time consumed by the original test suite respectively. Note that the fluctuations of CPU time in Fig. 26.10 are due to the dynamic behavior of the computer system.

Fig. 26.10 shows that test filtering can achieve substantial benefits in terms of reduction of computational costs. As we increase $\gamma_{\min}$ the number of tests that needs to be considered and the corresponding CPU time consumed both decrease significantly and quickly. Even if we set $\gamma_{\min}$ to a very small value, test filtering can result in a considerable reduction of CPU time. Suppose as an example, we set $\gamma_{\min} = 0.01$, then CPU time reduction is about 59% for C1, 31% for T1, 97% for L1 and 37% for M1. Generally, if we set $\gamma_{\min}$ to a larger value, we can consider a smaller test suite for fitness evaluation. However, it is not desirable to use too large a $\gamma_{\min}$ (e.g., 0.5), because it could result in the repair approaches missing the actual faulty location. In practice, we choose a moderate value for $\gamma_{\min}$ (e.g., 0.1) to strike a compromise. Normally, test filtering can significantly speed up the fitness evaluation in such a case. For example, if we set $\gamma_{\min}$ to 0.1 for M1, the number of tests considered is reduced from 5,246 to 118, and the CPU time for one fitness evaluation is reduced from 210 seconds to 3.4 seconds. Suppose the termination criterion of ARJA-p is 2,000 evaluations, then we can save up to 115 hours for just a single repair trial.

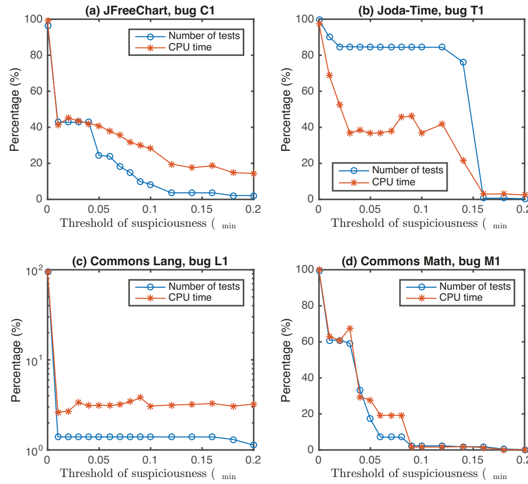


Fig. 26.10: Illustration of the value of the test filtering procedure. The base 10 logarithmic scale is used for the y axis in (c).

By conducting a post-run validation of the obtained patches on the original test suite we confirm that if a patch can pass the reduced test suite, it can also pass the original one, without any exception.

26.6 Conclusion

In this chapter we have examined a new program repair approach for Java, ARJA-p. It combines very recent techniques from template based and EC based repair approaches to enhance performance of PAR. Our technique is compared with almost all the existing techniques (11 tools) that have ever been evaluated on Defects4J dataset. The empirical results on 224 real bugs in Defects4J show that ARJA-p can generate test-adequate patches for 84 bugs and correct patches for 37 bugs outperforming state-of-the-art approaches with a considerable margin. ARJA-p can correctly fix several multi-location bugs that cannot be handled by almost any existing repair approaches. We have also found that there exists good The complementarity in performance shown between ARJA-p and other state-of-the-art techniques like ACS and ARJA imply that a combination of principles of existing approaches is a promising avenue to further improve repair effectiveness of evolutionary program repair methods.

Acknowledgements W.B. gratefully acknowledges support from the Koza endowment at MSU, made possible with a generous gift by John R. Koza. This would not have been possible without the BEACON Center for the Study of Evolution in Action under the capable leadership of Erik Goodman.

References

1. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: *An evaluation of similarity coefficients for software fault localization*. In: Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on, pp. 39–46. IEEE (2006)
2. Banzhaf, W.: Some remarks on code evolution with genetic programming. In: A. Adamatzky, S. Stepney (eds.) Inspired by Nature, pp. 145–156. Springer (2018)
3. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: *Genetic Programming: An Introduction*. Morgan Kaufmann, San Francisco (1998)
4. Barr, E.T., Brun, Y., Devanbu, P., Harman, M., Sarro, F.: *The plastic surgery hypothesis*. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 306–317. ACM (2014)
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 Press (2017)
7. D'Antoni, L., Samanta, R., Singh, R.: *QLOSE: Program repair with quantitative objectives*. In: International Conference on Computer Aided Verification, pp. 383–401. Springer (2016)
8. 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)
9. Gazzola, L., Micucci, D., Mariani, L.: *Automatic software repair: A survey*. IEEE Transactions on Software Engineering **45**, 34–67 (2017)
10. Gupta, R., Pal, S., Kanade, A., Shevade, S.: *DeepFix: Fixing Common C Language Errors by Deep Learning*. In: AAAI, pp. 1345–1351 (2017)
11. Harman, M., Jones, B.F.: *Search-based software engineering*. Information and software Technology **43**(14), 833–839 (2001)
12. Harman, M., Mansouri, S.A., Zhang, Y.: *Search-based software engineering: Trends, techniques and applications*. ACM Computing Surveys (CSUR) **45**(1), 11 (2012)
13. 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)
14. Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y.: *Repairing programs with semantic code search (t)*. In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, pp. 295–306. IEEE (2015)
15. Kim, D., Nam, J., Song, J., Kim, S.: *Automatic patch generation learned from human-written patches*. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 802–811. IEEE Press (2013)
16. Koza, J.R.: *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, Cambridge, MA (1992)
17. 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 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 376–379. ACM (2017)
18. Le, X.B.D., Lo, D., Le Goues, C.: *Empirical study on synthesis engines for semantics-based program repair*. In: Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on, pp. 423–427. IEEE (2016)
19. Le, X.B.D., Lo, D., Le Goues, C.: *History driven program repair*. In: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, vol. 1, pp. 213–224. IEEE (2016)
20. 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: Software Engineering (ICSE), 2012 34th International Conference on, pp. 3–13. IEEE (2012)

21. 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)
22. Le Goues, C., Weimer, W., Forrest, S.: *Representations and operators for improving evolutionary software repair*. In: Proceedings of the 14th annual conference on Genetic and evolutionary computation, pp. 959–966. ACM (2012)
23. Long, F., Amidon, P., Rinard, M.: *Automatic inference of code transforms for patch generation*. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 727–739. ACM (2017)
24. Long, F., Rinard, M.: *Staged program repair with condition synthesis*. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 166–178. ACM (2015)
25. Long, F., Rinard, M.: *Automatic patch generation by learning correct code*. In: ACM SIGPLAN Notices, vol. 51, pp. 298–312. ACM (2016)
26. 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 pp. 1–29 (2016)
27. Martinez, M., Monperrus, M.: *Mining software repair models for reasoning on the search space of automated program fixing*. Empirical Software Engineering **20**(1), 176–205 (2015)
28. Martinez, M., Monperrus, M.: *Open-ended Exploration of the Program Repair Search Space with Mined Templates: the Next 8935 Patches for Defects4J*. arXiv preprint arXiv:1712.03854 (2017)
29. 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)
30. Mechtaev, S., Yi, J., Roychoudhury, A.: *Directfix: Looking for simple program repairs*. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 448–458. IEEE Press (2015)
31. Mechtaev, S., Yi, J., Roychoudhury, A.: *Angelix: Scalable multiline program patch synthesis via symbolic analysis*. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, pp. 691–701. IEEE (2016)
32. Monperrus, M.: *Automatic software repair: A bibliography*. ACM Computing Surveys (2017)
33. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: *Semfix: Program repair via semantic analysis*. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 772–781. IEEE Press (2013)
34. 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 pp. 1–27 (2018)
35. 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)
36. 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)
37. 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)
38. 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 Press (2017)
39. Tan, S.H., Yoshida, H., Prasad, M.R., Roychoudhury, A.: *Anti-patterns in search-based program repair*. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 727–738. ACM (2016)
40. Weimer, W., Fry, Z.P., Forrest, S.: *Leveraging program equivalence for adaptive program repair: Models and first results*. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 356–366. IEEE Press (2013)

41. 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)
42. 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)
43. 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)
44. Yang, J., Zhikhartsev, A., Liu, Y., Tan, L.: *Better test cases for better automated program repair*. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 831–841. ACM (2017)
45. Yuan, Y., Banzhaf, W.: *ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming*. IEEE Transactions on Software Engineering (2020, in press)
46. 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)
47. Yuan, Y., Xu, H., Wang, B., Zhang, B., Yao, X.: *Balancing convergence and diversity in decomposition-based many-objective optimizers*. IEEE Transactions on Evolutionary Computation **20**(2), 180–198 (2016)
48. Zhong, H., Su, Z.: *An empirical study on real bug fixes*. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, pp. 913–923. IEEE Press (2015)