Towards Better Evolutionary Program Repair: An Integrated Approach

YUAN YUAN, Michigan State University, USA
WOLFGANG BANZHAF, Michigan State University, USA

Bug repair is a major component of software maintenance, which requires a huge amount of manpower. Evolutionary computation, particularly genetic programming (GP), is a class of promising technique for automating this time-consuming and expensive process. Although recent research in evolutionary program repair has made significant progress, major challenges still remain. In this paper, we propose ARJA-e, a new evolutionary repair system for Java code that aims to address challenges for the search space, search algorithm and patch overfitting. To determine a search space that is more likely to contain correct patches, ARJA-e combines two sources of fix ingredients (i.e., the statement-level redundancy assumption and repair templates) with contextual analysis based search space reduction, thereby leveraging their complementary strengths. To encode patches in GP more properly, ARJA-e unifies the edits at different granularities into statement-level edits, and then uses a lower-granularity patch representation that is characterized by the decoupling of statements for replacement and statements for insertion. ARJA-e also uses a finer-grained fitness function that can make full use of semantic information contained in the test suite, which is expected to better guide the search of GP. To alleviate patch overfitting, ARJA-e further includes a post-processing tool that can serve the purposes of overfit detection and patch ranking. We evaluate ARJA-e on 224 real Java bugs from Defects4J and compare it with the state-of-the-art repair techniques. The evaluation results show that ARJA-e can correctly fix 39 bugs in terms of the patches ranked first, achieving substantial performance improvements over the state of the art. In addition, we analyze the effect of the components of ARJA-e qualitatively and quantitatively to demonstrate their effectiveness and advantages.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; **Software testing and debugging**;

Additional Key Words and Phrases: Evolutionary computation, program repair, genetic programming, genetic improvement

ACM Reference Format:
Yuan Yuan and Wolfgang Banzhaf. 2010. Towards Better Evolutionary Program Repair: An Integrated Approach. *ACM Trans. Softw. Eng. Methodol.* 9, 4, Article 39 (March 2010), 54 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Automatic program repair [28, 70] 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* [70]. The test suite should contain at least one negative (i.e., initially failing) test case that triggers the bug to be fixed and a number of positive (i.e., initially passing) test cases that define the expected program behavior. In terms of test-suite based repair, a bug is regarded to

Authors' addresses: Yuan Yuan, Michigan State University, Department of Computer Science and Engineering, East Lansing, MI, 48824, USA, yyuan@msu.edu; Wolfgang Banzhaf, Michigan State University, Department of Computer Science and Engineering, East Lansing, MI, 48824, USA, banzhafw@msu.edu.

1049-331X/2010/3-ART39 $15.00
https://doi.org/0000001.0000001

be *fixed* or *repaired*, if a created patch allows the entire test suite to pass. Such a patch is referred to as a *test-adequate patch* [61] or a *plausible patch* [79].

Evolutionary repair approaches [92] are a popular category of techniques for test-suite based repair. These approaches determine a search space potentially containing correct patches, then use evolutionary computation (EC) techniques, particularly genetic programming (GP) [8, 11, 39], to explore that search space. A major characteristic of evolutionary repair approaches is that they have great potential to fix multi-location bugs, since GP can manipulate multiple likely faulty locations simultaneously. In contrast, it is hard or even impossible for other kinds of repair techniques (e.g., [55, 73, 82, 96, 103]) to generate multi-location repairs. Considering that multi-location bugs are common in real software projects [86, 87, 111], this characteristic makes evolutionary repair approaches very attractive and promising. However, GenProg [26, 45, 48, 94], the most well-known approach of this kind, does not fully use the potential in multi-location bug fixing according to large-scale empirical studies [61, 79], partly due to the search ability of the underlying GP [74, 79, 108]. To tackle this issue, our previous work introduced ARJA [108], which uses a novel multi-objective GP approach with better search ability to explore GenProg's search space. Although ARJA has improved performance compared to GenProg and also demonstrated its strength in multi-location repair, major challenges [46] still remain for evolutionary program repair.

The first challenge is how to construct a reasonable search space that is more likely to contain correct patches. In this respect, GenProg and ARJA exploit the statement-level *redundancy assumption* [65] (also called *plastic surgery hypothesis* [9]). That is, they only conduct statement-level changes and use existing statements in the buggy program for replacement or insertion. There are two limitations for such a search space: (i) the statements randomly excerpted from somewhere in the current program may have little pertinence to the likely-buggy statement to be manipulated, leading to nonsensical patches [79, 89]; (ii) the fix ingredients usually do not exist in a buggy program at the statement level [9, 65] which is too coarse-grained, so such a search space does not contain correct patches for many bugs. Given these two limitations, an evolutionary repair approach called PAR [37] exploits *repair templates*. Each template specifies one type of program transformation and is derived from common fix patterns. Due to using templates, PAR can conduct more targeted code changes (e.g., adding a null-pointer checker), or conduct changes at a finer granularity (e.g., replacing a method name) than statement-level approaches. However, there are other limitations in PAR's search space: (i) this search space does not contain correct patches that involve complex statement-level transformations, whereas they are available in GenProg's and ARJA's search space; (ii) the finer-grained changes defined in PAR's templates can only apply to three types of abstract syntax tree (AST) nodes (i.e., method name, parameter and expression in a conditional branch), limiting its repair effectiveness. Considering all the above, we have the following insights for determining a more promising search space in evolutionary program repair: (i) we can exploit both statement-level redundancy assumption and repair templates, in order to leverage their complementary advantages; (ii) we can conduct contextual analysis to select more related replacement and insertion statements when using the statement-level redundancy assumption; (iii) we can introduce several additional templates from recent non-evolutionary repair approaches (e.g., [55, 82, 99]), in order to conduct finer-grained changes for more types of AST nodes.

The second challenge is how to design a search algorithm that can navigate the search space more effectively. The combination of statement-level redundancy assumption and repair templates will lead to a much larger search space, thereby making this challenge more serious. Recent studies [74, 108] have indicated that compared to using GenProg's patch representation, using a lower-granularity patch representation that decouples the partial information of an edit can significantly improve the search ability of GP in bug repair. One possible reason is that good partial information

of an edit (e.g., a useful operation type) can be quickly propagated from one solution to others. However such lower-granularity representations have two weakness: (i) they are specially designed for statement-level edits and cannot be directly used for template-based edits (usually occurring at the expression level); (ii) they do not discern the statements for replacement and those for insertion, but promising replacement and insertion statements usually have different characteristics and should be evolved by GP separately. Besides the patch representation, the fitness function is another important factor that influences the search ability of GP. In existing evolutionary repair approaches (e.g., GenProg, PAR and ARJA), the fitness function is generally defined based on how many test cases a patched program passes. However this kind of fitness function can only provide a binary signal (i.e, passed or failed) for a test case and cannot measure how close a modified program is to pass a test case. The consequence is that there may be a large number of plateaus in the search space [25, 46, 79], thereby trapping the search of GP.

The third challenge is how to alleviate patch overfitting [85]. Evolutionary repair approaches can usually find a number of plausible patches within a computing budget. But most of these patches in general may be incorrect, by just overfitting the given test suite. To select correct patches more easily, it is necessary to include a post-processing step for these approaches, which can filter out incorrect patches (i.e., *overfit detection*) or rank the plausible patches found (i.e., *patch ranking*). However, almost all existing evolutionary repair systems, including GenProg, PAR, and ARJA, do not implement such a step. Although there are some overfit detection (e.g., [98, 100, 104]) and patch ranking (e.g., [43, 55, 96, 101]) techniques available in the literature, their applicability in our proposed repair system is unclear. For overfit detection, a recent technique [100] exploits the similarity of complete-path spectra between test executions. But this technique is sensitive to patch complexity, that is, it would probably fail when a correct patch significantly changes the control flow of a positive test execution (e.g., using new algorithmic procedures or method invocations). In addition, this technique would also probably fail when an incorrect patch does not change the control flow of any positive test execution. As for patch ranking, most of the existing techniques (e.g., [43, 55, 101]) depend on a specific repair approach and cannot be generalized to others. Furthermore, these ranking techniques (e.g., [43, 55, 96, 101]) generally only rely on syntax information rather than semantic information, which may limit their ranking accuracy.

To address the above three challenges, we propose a set of new techniques for better evolutionary program repair. These techniques are implemented in a new repair system called ARJA-e, which significantly *enhances* our original ARJA system [108]. The main contributions of this paper are summarized as follows:

(1) We determine a search space by combining the statement-level redundancy assumption with repair templates generalized from PAR and recent non-evolutionary repair approaches, in order to leverage their complementarity.
(2) When exploiting the statement-level redundancy assumption, we distinguish statements for replacement and statements for insertion, and introduce two context-related metrics in order to select the most promising replacement and insertion statements, respectively.
(3) We use repair templates in a novel way by converting various template-based edits (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, making it possible to encode patches in a unified lower-granularity representation.
(4) We introduce a new lower-granularity patch representation which is characterized by the decoupling of statements for replacement and statements for insertion, thereby making GP evolve the two kinds of statements separately.

(5) We develop a finer-grained fitness function that can capture how well a program variant satisfies each assertion of the unit test cases. This is expected to provide smoother gradients for GP to traverse to find a solution.
(6) We propose a new overfit detection approach called CIP, which is based on the assumption that even a buggy program can function **c**orrectly on the test **i**nputs encoded in the **p**ositive test cases. CIP does not rely on measuring changes in control flow and is thus not sensitive to the complexity of patches.
(7) Based on CIP, we present a heuristic patch ranking procedure, which considers both the syntactic and semantic changes introduced by the patch. This procedure can be easily generalized to other repair systems.
(8) In order to evaluate ARJA-e we conduct an extensive empirical study on 224 real Java bugs in Defects4J [34]. Our results show that ARJA-e can fix 106 bugs and correctly fix 39 bugs, which outperforms state-of-the-art repair approaches by a large margin in terms of overall performance. Moreover, we verify the effectiveness of the components of ARJA-e (e.g., the finer-grained fitness function and overfit detection approach) and obtain some new findings and insights.

The rest of this paper is organized as follows. Section 2 provides the background and motivation for our study. Section 3 describes the proposed repair system in detail. Section 4 provides some important implementation details of our system. Section 5 presents the experimental design and Section 6 reports our experimental results. Section 7 discusses threats to validity. Section 8 reviews related work. Section 9 concludes.

2 BACKGROUND AND MOTIVATION

In this section, we first briefly introduce three typical evolutionary repair approaches which are closely related to our work. Then, we describe the goal and motivation of this study with examples. Lastly, we use a single running example to demonstrate how ARJA-e works.

2.1 Typical Evolutionary Repair Approaches

2.1.1 GenProg. GenProg [26, 45, 48, 94] uses GP to search for patches of a buggy program, as validated by test cases. Given a buggy program and an associated test suite (including positive test cases and at least one negative test case) as input, GenProg first employs a fault localization strategy to find a number of likely-buggy statements (LBSs) that can be manipulated by GP. Fault localization can substantially reduce the search space, which is critical to the scalability of GenProg and has been an essential module for almost all test-suite based repair systems.

For each LBS localized, GenProg can choose a modification using either of the three types of statement-level edits: (i) *delete* the LBS; (ii) *replace* the LBS with another statement; (iii) *insert* another statement before the LBS. In case of "replace" and "insert", a second statement (called *ingredient statement*) is required. In GenProg, it is assumed that this statement comes from elsewhere in the buggy program, according to the redundancy assumption [65]. So GenProg does not invent any new code.

After specifying where and how to modify the buggy program, GenProg encodes a program variant as a genome in GP. Earlier versions [26, 48, 94] of GenProg used an extended AST as a genetic representation for GP. However, such a representation usually leads to unaffordable memory consumption for large programs, limiting its scalability. Le Goues et al. [45] addressed this issue by introducing the *patch representation*. Instead of directly encoding a program variant as an AST, this patch representation encodes a program variant as a sequence of concrete statement-level edits to the buggy program, as illustrated in Fig. 1(a). With this representation, the underlying GP in

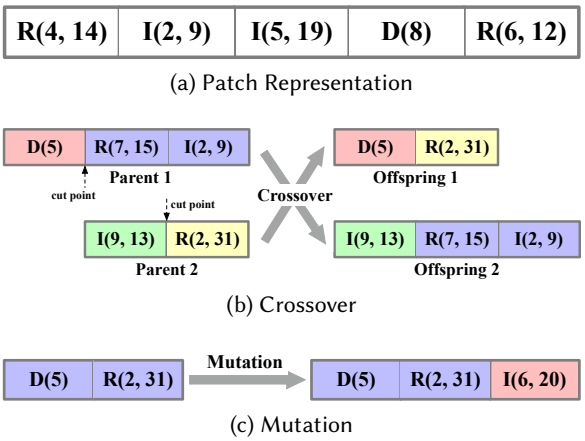


Fig. 1. Illustration of the patch representation and genetic operators in GenProg. For brevity, "D" denotes a delete operation; "R" a replace; and "I" an insert. The integers denote the AST node numbers of the corresponding statements. D(a) means that delete "a"; R(a, b) means that replace "a" with "b"; I(a,b) means that insert "b" before "a".

GenProg can use a single-point crossover operator to generate offspring, as illustrated in Fig. 1(b). The mutation operator in GenProg is very important, because it is responsible for introducing new edits. This operator is executed by appending a new randomly generated edit to the genome under mutation, as illustrated in Fig. 1(c). To evaluate the fitness or desirability of each genome, GenProg defines a *fitness function* via the test cases, which returns a weighted sum of all test cases passed by the program variant.

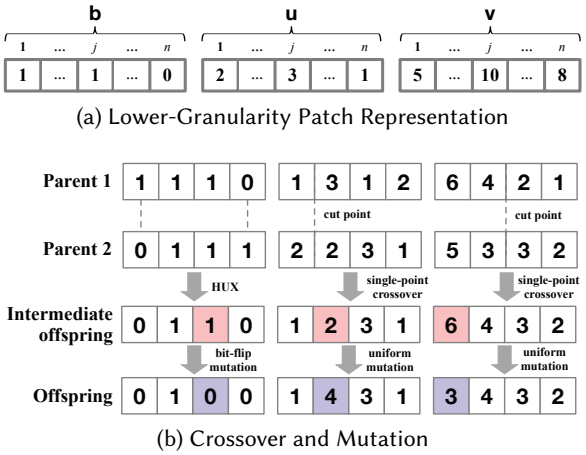
The GP procedure in GenProg is summarized as follows. First, an initial population of size N is generated by applying independent mutation to N copies of empty patches. Then the iteration of GP starts. In each generation of GP, tournament selection chooses $N/2$ individuals from the current population as parents for mating; crossover is performed on these selected parents to generate $N/2$ offspring; lastly each parent and offspring undergo mutation and the mutated parents together with the mutated offspring will constitute the next population. GenProg also includes a post-processing step which removes unnecessary edits of the patches obtained using delta-debugging.

2.1.2 PAR. PAR [37] is an evolutionary repair technique based on *repair templates*. Unlike GenProg which performs random statement replacement, insertion and deletion, 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 an `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 evolutionary process of PAR starts with an initial population of program variants and subject it to iterations of repeating two tasks: reproduction and selection. In the reproduction stage, each program variant derives a new one by applying templates to the selected suspicious statements. In the selection stage, tournament selection is used to choose better (in terms of passing test cases) program variants to constitute the next population. Iterations terminate when a program variant passes all tests.

From an EC perspective, PAR uses an evolutionary process similar to that in GenProg, but only relies on mutation (based on templates) and does not use any crossover operator. 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 via variation.

2.1.3 ARJA. ARJA [108] is a recently proposed repair approach for automated repair of Java programs. ARJA basically works with GenProg's search space, but introduces several key changes to improve the GP algorithm. In ARJA, a lower-granularity patch representation is used instead of GenProg's patch representation, which decouples the search subspaces of likely-buggy locations, operation types and potential fix ingredients. Suppose that there are n LBSs considered and for the j-th LBS, O_j is the set of available statement-level operation types and D_j is the set of ingredient statements. A patch in ARJA is encoded as a three-part vector $\mathbf{x} = (\mathbf{b}, \mathbf{u}, \mathbf{v})$, where $\mathbf{b}$, $\mathbf{u}$ and $\mathbf{v}$ are all integer vectors with size n. In genome $\mathbf{x}$, $b_j \in \{0, 1\}$ determines whether or not the patch chooses to edit the j-th LBS, $u_j \in \{1, 2, \ldots, |O_j|\}$ determines that the patch chooses the u_j-th operation type in O_j for the j-th LBS, and $v_j \in \{1, 2, \ldots, |D_j|\}$ determines that if a replace or insert operation is chosen, the patch chooses the v_j-th ingredient statement in D_j for the j-th LBS. Fig. 2(a) illustrates the lower-granularity patch representation in ARJA. For producing offspring in GP, crossover and mutation are applied to each part separately, as illustrated in Fig. 2(b). Due to the lower-granularity patch representation and associated genetic operators, good partial information of an edit (e.g., a promising operation type, an accurate faulty location, and a useful ingredient statement) can be propagated from one solution to others, enabling GP to traverse the search space more effectively.



(a) Lower-Granularity Patch Representation

(b) Crossover and Mutation

Fig. 2. Illustration of the patch representation and genetic operators in ARJA.

Unlike other evolutionary repair approaches, ARJA formulates program repair as a multi-objective optimization problem with two objectives, namely *patch size* (denoted by $f_1(\mathbf{x})$) and *weighted failure rate* (denoted by $f_2(\mathbf{x})$). The two objectives are defined as follows:

$$f_1(\mathbf{x}) = \sum_{j=1}^{n} b_j \tag{1}$$

$$f_2(\mathbf{x}) = \frac{|\{t \in T_{pos} \mid \mathbf{x} \text{ fails } t\}|}{|T_{pos}|} + w \times \frac{|\{t \in T_{neg} \mid \mathbf{x} \text{ fails } t\}|}{|T_{neg}|} \tag{2}$$

where T_{neg} is the set of negative test cases, T_{pos} is the reduced set of positive test cases obtained via test filtering, and $w \in (0, 1]$ is a global weight parameter that can introduce a bias toward negative test cases. $f_1(\mathbf{x})$ counts the number of edit operations contained in the patch. $f_2(\mathbf{x})$ is similar to the canonical fitness function used in GenProg with $f_2(\mathbf{x}) = 0$ indicating that $\mathbf{x}$ does not fail any test case and represents a plausible patch. ARJA employs NSGA-II [21] to simultaneously minimize the two objectives in order to find simpler patches. Moreover, the empirical results in [108] have shown that multi-objective search can achieve higher success rates of repair than a single-objective search.

ARJA also includes several auxiliary techniques. To speed up the fitness evaluation of GP, ARJA adopts a test filtering procedure that allows to ignore test cases that are unrelated to the LBSs considered. In order to reduce the search space, ARJA applies three types of rules that can be used to avoid unnecessary code manipulations. In addition, an optional type matching strategy is provided which can create new potential fix ingredients by exploiting statements that are out of scope.

2.2 Goal and Motivation

The overall goal of this study is to develop an improved GP-based approach for evolutionary program repair. The motivation is based on the weaknesses in existing evolutionary repair systems, particularly ARJA, GenProg and PAR. We analyze these weaknesses as follows.

2.2.1 Single Source of Fix Ingredients. Most evolutionary approaches for software repair acquire potential fix ingredients from a single source. For example, in GenProg or ARJA, the statements used for replacement or insertion are just excerpted from elsewhere in the buggy program; while in PAR, the potential fix code is generated only with the help of predefined repair templates. The problem is that using fix ingredients from just a single source may severely limit the repair ability of evolutionary approaches. In the following, we use real bugs as an illustration.

Fig. 3 shows the human-written patch for bug Math75 from the Defects4J [34] dataset. To correctly fix this bug, a slight modification (i.e., change the method name `getCumPct` to `getPct`) is required. However GenProg or ARJA cannot find a correct patch since the fix statement used for replacement (i.e., `return getPct((Comparable<?>)v);`) does not happen to appear elsewhere in the buggy program. In contrast, it is quite possible for PAR to find a correct patch, because PAR uses a repair template called "Method Replacer" that can target this bug.

```
1  // Frequency.java
2  public double getPct(Object v) {
3  -    return getCumPct((Comparable<?>) v);
4  +    return getPct((Comparable<?>) v);
5  }
```

Fig. 3. The human-written patch for bug Math75.

Fig. 4 shows a human-written patch for bug Math39 from Defects4J. To correctly fix this bug, an `if` statement with relatively complex control logic should be inserted before the buggy code. However, for PAR, the bug is hard to fix correctly, because this fix generally does not belong to a common fix pattern and is difficult to encode with templates. In contrast, GenProg and ARJA 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; }
```

is found somewhere else in the buggy program and is semantically equivalent to the one inserted by a human developer.

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

Fig. 4. The human-written patch for bug Math39.

These two examples demonstrate that evolutionary repair approaches that only rely on the statement-level redundancy assumption (e.g., GenProg and ARJA) and those only relying on templates (e.g., PAR) can uniquely repair some bugs but not others. Clearly it should be promising to combine both of sources of fix ingredients, which may significantly improve the repair performance. In this study, we will investigate this combination.

We note that the search space will become larger when multiple sources of fix ingredients are used, posing a greater challenge to the evolutionary search. Recent studies [74, 108] have shown that a lower-granularity patch representation can improve the search ability and is helpful for evolutionary repair approaches to find multi-edit patches. However, these lower-granularity representations (e.g., as shown in Fig. 2(a)) are specially designed for statement-level edits and cannot be directly applied to expression-level edits that are used by template-based repair actions. So here we shall also present a new lower-granularity patch representation for the purpose of combining the statement-level redundancy assumption and repair templates, in order to allow the evolutionary search explore the resulting larger search space more effectively.

2.2.2 Coupling of Replacement and Insertion. No matter the replace or insert operation, GenProg and ARJA choose an ingredient statement in the same way based on the statement-level redundancy assumption. However for a LBS, promising replacement and insertion statements can be quite different. For example, suppose that the statement at line 4 in Fig. 5 is a LBS. Then a programmer can easily figure out that it is potentially useful to insert an ingredient statement `i++;` before the LBS, though it is generally not promising to replace the LBS with `i++;`.

```
// EigenDecomposition.java
 private SchurTransformer transformToSchur(final RealMatrix matrix) { ...
    imagEigenvalues[i] = z;
    realEigenvalues[i] = x + p;   //a likely buggy statement
...
}
```

Fig. 5. A code snippet for illustrating the difference between the promising statements for replacement and those for insertion.

In this study, we will approach this problem from two angles. First, we will distinguish candidate statements for replacement and those for insertion when selecting a potential fix ingredient for each LBS. Second, we will use a new lower-granularity patch representation that decouples the representation of replacement and insertion statements, so that the two kinds of statements can be evolved separately. This is different from ARJA's patch representation (see Fig. 2(a)), where replacement and insertion statements are considered without distinction and the corresponding information is mixed in a single sub-structure (i.e., **v** as shown in Fig. 2(a)).

2.2.3 Inadequate Analysis of Program Context. GenProg checks the validity of the replacement or insertion code only according to the variable scope at destination. Besides variable and method scope, ARJA utilizes more constraints enforced by the compiler to further filter out some invalid ingredient statements. The goal of these techniques is to render program variants produced during evolution with a higher chance to be successfully compiled thereby improving search efficiency. However, there exist many statements that do not violate any compiler constraint but have very little potential to become a fix ingredient. For example, in Fig. 6, suppose that the statement at line 6 is a LBS. GenProg and ARJA will regard `this.totalIterations = 0;` as a candidate statement for insertion because this statement resides in the same Java file and the insertion will not make the modified program fail to compile. But indeed it is unlikely that such an insertion is helpful for fixing the bug, because the inserted statement has little relevance to the program context around it.

```
1  // MultiStartUnivariateRealOptimizer.java
2  public double[] getOptimaValues() throws IllegalStateException {
3     if (optimaValues == null)
4        throw MathRuntimeException.createIllegalStateException();
5  +  this.totalIterations = 0;
6     return optimaValues.clone();  //a likely-buggy statement
7  }
```

Fig. 6. Illustration of a modification without considering the program context.

In this study, we will employ light-weight contextual analysis of a program when selecting candidate statements for replacement/insertion. It is expected that this analysis can discard many potential edit operations which are satisfying compiler constraints but are not promising, significantly reducing the search space and helping to avoid patch overfitting. In fact, contextual analysis has been conducted by several non-evolutionary repair approaches (e.g., ELIXIR [82] and CAPGEN [96]) for prioritizing the validation of patches. A similar analysis for constraining the search space has not yet been investigated in evolutionary repair frameworks. Again, in line with the insight in Section 2.2.2, we will distinguish replacement and insertion operations in contextual analysis, which also differs from previous work.

2.2.4 Coarse-Grained Fitness Function. The fitness function of existing evolutionary repair approaches such as GenProg and ARJA is generally based on the number of test cases passed by a patched program. This limits the value of such a fitness function because each test case only provides a binary signal (i.e., passed or failed). There may not be smooth gradients for the evolutionary algorithm to effectively explore the search space [46, 79]. Take a real bug named Math67 in Defects4J [34] as an example. If we select 60 LBSs for manipulation, then there are 5 related positive test cases after test filtering (i.e., $|T_{pos}| = 5$) and 1 negative test case (i.e., $|T_{neg}| = 1$). According to Eq. (2), there are at most $(|T_{pos}| + 1) \times (|T_{neg}| + 1) - 1 = 11$ different fitness values for as many as

10^{60} different solutions (assuming only 10 available modifications for each LBS) in the search space. This implies that this fitness function is virtually blind and cannot guide a search algorithm well, since a vast number of solutions will have the same fitness value, likely reducing the evolutionary algorithm to a random search in this situation.

Nevertheless, a test case can potentially provide more information for the failing execution which can be exploited to further distinguish program variants. To understand this, we have to recall, that in Java a JUnit [16] test case is a method containing a number of assertions, as shown in Fig. 7. When executing a test case, if all *executed* assertions hold, then this test case is passed; otherwise if any assertion is violated, an exception is thrown right away and the test case fails. Here we can have two insights. First, different failing executions of a test case may have a different number of failing assertions. Second, for each violated assertion, we can measure the degree to which it does not hold. For example, the assertion statement at line 6 in Fig. 7 checks whether `z.getReal()` is equal to -9.0 within a positive 1.0e-5 accuracy, but a program variant returning `z.getReal()` as -9.1 is usually better than one returning it as 5. This insight is conceptually similar to the *branch distance* in search-based software testing [27, 66]. Note that Arcuri [4, 6] also opted for a kind of distance function for bug repair, but their implementation was largely a proof-of-concept and cannot be applied to real-world software with standard unit tests.

```
1 @Test
2 public void testMultiply() {
3     Complex x = new Complex(3.0, 4.0);
4     Complex y = new Complex(5.0, 6.0);
5     Complex z = x.multiply(y);
6     Assert.assertEquals(-9.0, z.getReal(), 1.0e-5);      //assertion
7     Assert.assertEquals(38.0, z.getImaginary(), 1.0e-5); //assertion
8 }
```

Fig. 7. A sample JUnit test case from the Math project in Defects4J.

In this study, we will develop a finer-grained fitness function based on the above two insights, in order to provide better guidance for the evolutionary search in program repair.

2.2.5 Patch Overfitting. Patch overfitting [79, 85] in program repair means that a patch is not correct beyond passing the test cases given. This problem is not unique to evolutionary repair approaches but critical for all test-suite based repair approaches. The underlying reason is that a test suite is usually weak and cannot fully describe the program specification. To address this issue, non-evolutionary repair approaches [31, 57, 82, 96, 101] generally use certain well-defined metrics to prioritize the validation of candidate patches, and the first patch validated by test cases is more likely to be correct. However, such a strategy is not applicable in evolutionary repair approaches. Because the stochastic nature of evolutionary algorithm makes it hard to control which solution is to be examined first it is better to first leverage the strong search ability to find a number of plausible patches within a given computing budget, then use a post-processing step to select one or several patches that are most likely to be correct among these plausible patches. Current evolutionary repair approaches including ARJA , GenProg and PAR do not have such a post-processing step.

In this study, we will develop a post-processing tool for addressing patch overfitting in evolutionary program repair. On one hand, this tool can detect overfitting patches. On the other hand, it can use a heuristic multi-level procedure to rank plausible patches found.

2.3 A Running Example

Fig. 8 shows a correct patch for bug Math22 in Defects4J. To fix this bug, ARJA-e first finds a number of likely-buggy statements (LBSs) by fault localization. For this example, 15 LBSs are found according to the threshold of the suspiciousness, including the statements at lines 3, 8 and 12 of Fig. 8. Then, ARJA-e discards the positive test cases that are unrelated to the 15 LBSs, which reduces the number of positive test cases from 4,116 to 79. For each of the 15 LBSs, ARJA-e can decide whether or not to edit it. In ARJA-e, there are three possible ways to act on a LBS (i.e., delete it, replace it with another statement, or insert another statement before it). For replacement or insertion, ARJA-e acquires candidate statements from the following two sources.

(1) **Current Buggy Program.** Take the LBS at line 8 of Fig. 8 for example, one of its candidate statements for insertion is `if (x <= 0) return 0;`, which is just copied from elsewhere in the buggy program. In ARJA-e, the statements for replacement are distinguished from those for insertion (e.g., `if (x <= 0) return 0;` is not included as a candidate replacement for the LBS at line 8). Next, ARJA-e conducts a contextual analysis allowing it to ignore some statements copied from the buggy program. For example, although an existing statement `random = null;` can also be inserted before the LBS at line 8 without violating compiler constraints, it will not be used as a candidate insertion for this LBS because of little contextual relevance.

(2) **Repair Templates.** For a LBS, ARJA-e can use a number of templates to generate new statements for replacement/insertion. For example, for the LBS at line 8, one of the candidate statements for replacement is `final double logx = FastMath.log(numeratorDegreesOfFreedom, x);`, which does not exist in the buggy program and is generated by applying a template called "Method Parameter Adjuster". As for the two replacement statements (at lines 4 and 13) in the correct patch, they are generated by a template called "Element Replacer" which can negate a boolean literal.

In addition, ARJA-e may disable several operation types for each LBS according to the predefined rules. For example, the deletion operation is disabled for the LBS at line 8, since this LBS is a variable declaration statement.

```
1  // UniformRealDistribution.java
2  public boolean isSupportUpperBoundInclusive() {
3  -    return false;
4  +    return true;
5  }
6  // FDistribution.java
7  public double density(double x) { ...
8      final double logx = FastMath.log(x);
9  ...
10 }
11 public boolean isSupportLowerBoundInclusive() {
12 -    return true;
13 +    return false;
14 }
```

Fig. 8. A correct patch generated by ARJA-e for bug Math22.

For each of the 15 LBSs, ARJA-e maintains a set of available operation types, a set of candidate statements for replacement and a set of candidate statements for insertion, which constitutes the

search space of patches for ARJA-e. In this example, the search space contains about 7.15×10^{24} candidate patches. ARJA-e employs GP to explore this large search space. ARJA-e's GP patch representation contains four parts: the first part encodes information about which LBSs are to be edited, and the remaining three parts encode choices related to the three sets maintained by ARJA-e, respectively. In order to generate offspring in GP, ARJA-e applies crossover and mutation operators to each of the four parts separately. ARJA-e's GP is guided by a multi-objective fitness function which instruments the test cases so as to track the execution of each assertion, as described in Section 2.2.4.

Finally, ARJA-e returns a number of plausible patches created with GP, among which only one patch (as shown in Fig. 8) is correct. Different from this correct patch, the obtained overfitting patches manipulate other LBSs in the file `FDistribution.java` instead of the LBS at line 12 of Fig. 8. For example, an overfitting patch inserts `if (x <= 0) return 0;` before line 8. However, all these LBSs in `FDistribution.java` manipulated by the overfitting patches have lower suspiciousness than the LBS at line 12. So if ARJA-e uses patch ranking as a post-processing step, the correct patch shown in Fig. 8 will be ranked first, because the total suspiciousness is the primary metric for the patch ranking procedure.

3 TECHNICAL APPROACH

3.1 Overview

Algorithm 1 gives the overall flow of the system. ARJA-e takes as input a buggy program associated with a JUnit test suite. The test suite should contain a number of positive test cases specifying the required program behavior and at least one negative test case exposing the bug to be fixed.

Fault Localization (step 1 of Algorithm 1). Given an input, ARJA-e first uses a fault localization technique called Ochiai [1, 14] to locate a list of likely-buggy statements (LBSs). Each LBS is assigned a suspiciousness $susp_j \in [0, 1]$ by Ochiai, indicating the likelihood of the LBS containing the fault. To reduce search space, we consider at most $n_{\max}$ LBSs with the largest suspiciousness 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.

Test Filtering (step 2 of Algorithm 1). Suppose n LBSs are selected according to $n_{\max}$ and $\gamma_{\min}$. We next conduct coverage analysis to remove positive test cases that are unrelated to the manipulation of these n LBSs.

Exploiting the Statement-Level Redundancy Assumption (steps 5–10 of Algorithm 1). ARJA-e can use three types of statement-level edit operations to manipulate a LBS (i.e., deletion, replacement and insertion). For the latter two types, another statement is needed, one source of which is the current buggy program (step 5 of Algorithm 1), following the redundancy assumption [9, 65]. Unlike existing approaches based on this assumption [26, 45, 48, 77, 93, 94, 108], ARJA-e separates the statements for replacement and those for insertion (steps 8–9 of Algorithm 1). So each LBS corresponds to two different sets of statements denoted by R_j (for replacement) and I_j (for insertion) respectively. To reduce the search space further and minimize overfitting, we conduct a static program analysis to discard replacement/insertion statements that are not well related to the context of the LBS (step 7 of Algorithm 1).

Exploiting Repair Templates (steps 11–17 of Algorithm 1). 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 (step 12 of Algorithm 1). Unlike PAR that executes templates *on-the-fly*, ARJA-e uses templates in an *offline* method so that we can convert various template-based edits (usually at the expression-level) into statement-level edits of two types, replacement and insertion (steps 14–15 of Algorithm 1). This

ALGORITHM 1: Overall algorithm flow of ARJA-e

Input: Buggy program *Prog*, set of positive test cases T_{pos}, set of negative test cases T_{neg}, universal set of statement-level operation types O, set of repair templates $Temp$, set of rules Rl, set of anti-patterns Ap
Output: Plausible patches.

1 $\{\text{LBS}_1, \text{LBS}_2, \ldots, \text{LBS}_n\} \leftarrow$ FaultLocalization($Prog$, T_{pos}, T_{neg});
2 $T_{pos} \leftarrow$ TestFiltering(T_{pos}, $\{\text{LBS}_1, \text{LBS}_2, \ldots, \text{LBS}_n\}$) ;
3 **for** $j \leftarrow 1$ **to** n **do**
4 $\quad R_j \leftarrow \emptyset$; $I_j \leftarrow \emptyset$; $O_j \leftarrow \emptyset$;
5 $\quad S_j \leftarrow$ CollectStatementsWithinSamePackage(LBS_j);
6 $\quad$ **foreach** *statement* $s \in S_j$ **do**
7 $\quad\quad flag_1 \leftarrow$ IsReplacementCandidate(s, LBS_j, Rl); $flag_2 \leftarrow$ IsInsertionCandidate(s, LBS_j, Rl);
8 $\quad\quad$ **if** $flag_1$ **then** $R_j \leftarrow R_j \cup s$;
9 $\quad\quad$ **if** $flag_2$ **then** $I_j \leftarrow I_j \cup s$;
10 $\quad$ **end**
11 $\quad$ **foreach** *repair template* $tp \in Temp$ **do**
12 $\quad\quad S_{tp} \leftarrow$ ApplyRepairTemplate(LBS_j, tp);
13 $\quad\quad$ **foreach** *statement* $s \in S_{tp}$ **do**
14 $\quad\quad\quad$ **if** *s is a statement for replacement* **then** $R_j \leftarrow R_j \cup s$;
15 $\quad\quad\quad$ **else** $I_j \leftarrow I_j \cup s$;
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 $\quad O_j \leftarrow$ InitializeOperationTypes(LBS_j, O, Rl, Ap);
19 **end**
20 $\mathcal{R} \leftarrow \{R_1, R_2, \ldots, R_n\}$; $\mathcal{I} \leftarrow \{I_1, I_2, \ldots, I_n\}$; $\mathcal{O} \leftarrow \{O_1, O_2, \ldots, O_n\}$;
21 $P_0 \leftarrow$ InitializePopulation(); $g \leftarrow 0$;
22 **foreach** *individual* $\mathbf{x} \in P_0$ **do** FitnessEvaluation($\mathbf{x}$, $\mathcal{R}$, $\mathcal{I}$, $\mathcal{O}$, T_{pos}, T_{neg}) ;
23 **while** *termination criterion is not met* **do**
24 $\quad Q_g \leftarrow$ GenerateOffspringByGeneticOperators(P_g);
25 $\quad$ **foreach** *individual* $\mathbf{x} \in Q_g$ **do** FitnessEvaluation($\mathbf{x}$, $\mathcal{R}$, $\mathcal{I}$, $\mathcal{O}$, T_{pos}, T_{neg}) ;
26 $\quad U_g \leftarrow P_g \cup Q_g$;
27 $\quad P_{g+1} \leftarrow$ RemoveDuplicatesAndSelectElites(U_g);
28 $\quad g \leftarrow g + 1$;
29 **end**
30 $P_g \leftarrow$ SelectNondominatedPlausible(P_g);
31 **if** *overfit detection mode is enabled* **then** $P_g \leftarrow$ FilterOutIncorrectByOverfitDetection(P_g) ;
32 **else if** *patch ranking mode is enabled* **then** $P_g \leftarrow$ RankPlausibleByPatchRanking(P_g) ;
33 **return** P_g;

approach allows the integration of redundancy-based statement-level edits and template-based edits into a unified evolutionary framework with a lower-granularity patch representation.

Initialization of Operation Types (step 18 of Algorithm 1). 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 [108] and anti-patterns [89] to determine the set of available operation types (denoted by O_j) for each LBS.

Searching for Patches by Multi-Objective Evolution (steps 21–30 of Algorithm 1). Now that we have determined the search space, there are 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). To find simple

patches using multi-objective GP, a customized NSGA-II [21] is used to explore the search space, with the guidance of our finer-grained fitness function.

Alleviating Patch Overfitting (steps 31–32 of Algorithm 1). For a bug, ARJA-e can usually find a number of plausible patches through evolutionary search. However, many of these patches may overfit the test suite and are thereby not correct. To alleviate the patch overfitting issue, we develop a post-processing tool which can identify overfitting patches (step 31 of Algorithm 1) or rank plausible patches found by ARJA-e (step 32 of Algorithm 1).

In the following subsections, we will detail how to determine the search space (i.e., steps 3–19 of Algorithm 1), how to search for patches (i.e., steps 21–30 of Algorithm 1), and how to alleviate patch overfitting (i.e., steps 31–32 of Algorithm 1).

3.2 Search Space Determination

3.2.1 Exploiting the Statement-Level Redundancy Assumption. For each LBS selected, we first identify the Java package in which it resides. 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, see Table 2 in [108]) introduced in the original ARJA [108] system. The motivation for the 6 rules is that although some statements can pass the 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 [108] (see Table 3 in [108]), among which 3 rules (denoted by F_b) are related to the replacement operation and the other 3 (denoted by F_c) are related to the insertion operation. For example, to avoid disrupting a program too severely, one of the rules in F_b is to not replace a variable declaration statement with other kinds of statements.

Note that even if s follows F_b (or F_c), we do not immediately regard it as a candidate statement for replacement (or insertion). Our insight is that if replacing the LBS with s is a promising manipulation, s should generally exhibit a certain *similarity* to the LBS; and if it is potentially useful to insert s before the LBS, s should generally have a certain *relevance* to the context surrounding the LBS. In the following, we describe how to quantify such similarity and relevance.

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

$$sim(s, \text{LBS}) = \frac{|V_{\text{LBS}} \cap V_s|}{|V_{\text{LBS}} \cup V_s|} \tag{3}$$

Note that when collecting fields used by a statement, we also consider the fields accessed by invoking the methods in the current class. For example, if the LBS is `return x + getVal()` in the following code, then $V_{\text{LBS}} = \{\texttt{x}, \texttt{val}\}$. The field `val` is used by invoking `getVal()`.

```
1 class Example {
2    int val;
3    int getVal() { return val; }
4    int fun(int x) { return x + getVal(); }
5 }
```

In the method containing the LBS, suppose V_{bef} and V_{aft} are the sets of variables used by k statements before and after the LBS, respectively, where k is set to 5 by default in this paper. We

define the relevance of s to the context of LBS as follows:

$$rel(s, \text{LBS}) = \frac{1}{2}\left(\frac{|V_s \cap V_{\text{bef}}|}{|V_s|} + \frac{|V_s \cap V_{\text{aft}}|}{|V_s|}\right) \tag{4}$$

Eq. (4) indeed averages the percentages of the variables in V_s that are covered by V_{bef} and V_{aft}.

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

For each LBS considered, Fig. 9 summarizes the procedure to check whether statement s can become a candidate statement for replacement or insertion.

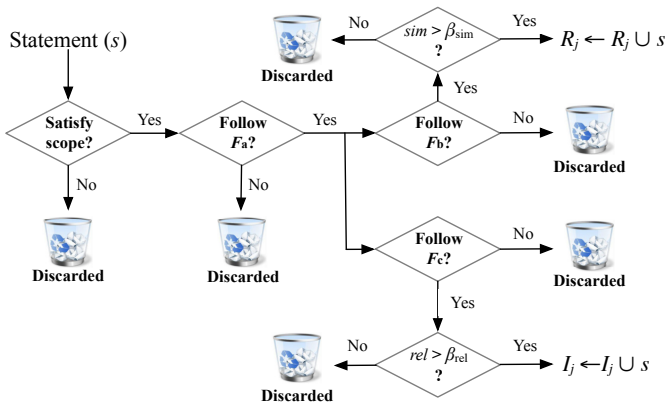


Fig. 9. The procedure to check whether s can become a candidate statement for replacement and insertion.

3.2.2 Exploiting Repair Templates. In this study, we also use repair templates to generate potential fix ingredients. Note that any repair approach using templates risks overfitting fix patterns in a bug dataset tested [86], because the design of these templates may rely on manual inspection of human-written patches in the test dataset. To alleviate this risk, we do not invent brand new repair templates. Instead, we basically adopt templates that have been used in existing repair approaches [22, 37, 53, 55, 80, 82] and just make several reasonable generalizations according to some characteristics of Java. We focus on how to make better use of repair templates in evolutionary search.

Table 1 describes the repair templates used in our approach. The first four templates (i.e., NPC, RC, CC and DC) are very similar and only differ in the context to be checked. NPC, RC and CC are inherited from PAR [37]. DC is included because a divide-by-zero error is another important source of software failure [58] similar to a null-pointer error. For each of the four templates, we use several alternative transformation schemata to manipulate the LBS. Take NPC as an example and suppose that a LBS includes an object reference o, then these schemata are illustrated as follows:

```
(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();
```

These are all natural ways to avoid executing the LBS (i.e., (i)–(v)) or to initialize o (i.e., (vi)), when o is null. The `if` statement in the first schema is used to replace the LBS, whereas each in the other

Table 1. The Description of Repair Templates Used in this Study

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

five should be inserted before the LBS. Not all six schemata are applicable to every LBS. The first schema cannot be used when the LBS is a variable declaration statement, and the fourth and fifth schema can only be used when the LBS is in a `for` or `while` loop, otherwise compile errors will occur. Note that in the second and third schema, an expression is needed in the `return` or `throw` statement. For the `return` statement, we choose to use any existing `return` statement in the method where the LBS resides or return the default value of the method's return type. As for the `throw` statement, we collect alternative thrown exceptions in three ways: (i) the thrown exceptions specified in the method declaration where the LBS resides; (ii) the thrown exceptions specified in the import declarations of the Java file where the LBS resides; (iii) the `IllegalArgumentException` is considered if `o` is a method parameter. Compared to our approach, PAR only uses the first transformation schema, limiting its ability to handle null pointer bugs. In addition, for template RC we also consider 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.

The next template, MPA, targets the invocation of a method that has overloaded methods. Similar transformation schemata have been adopted by PAR [37], ssFix [99], ELIXIR [82], SOFIX [53] and SKETCHFIX [31]. Template BEAR faithfully follows that in PAR. The parameters (in MPA) or terms (in BEAR) that can be added are from compatible variables or expressions in the same scope. The last template ER replaces an AST node element in a LBS with another compatible one. Table 2 lists the types of the AST node elements that can be replaced and describes the alternative replacers for each type. In Java, the field access and qualified name can be seen as special variables. For the replacement rules in Table 2, replacement of a primitive type with a widened type follows ELIXIR, replacement of a variable `x` with `f()` or `f(x)` is borrowed from the schema in REFAZER [80] and SOFIX, and the others are common mutation operations in mutation testing [32] that have been frequently used in template-based repair approaches [22, 37, 55].

In our system, we exploit repair templates in a different way compared to PAR and other related approaches [43, 55, 82]. We convert all template-based edits (usually occurring at the expression level) into statement-level edits. Such a conversion is straightforward. For example, the template-based edit that replaces the AST node `a` in the statement `a.callX();` with another node `b` is equivalent to the statement-level edit: replace the statement `a.callX();` with the statement `b.callX();`. With this in mind, for the j-th LBS, $j = 1, 2, \ldots, n$, we apply all possible transformations

Table 2. Replacement Rules for Different Types of AST Node Elements in ER

No.	Element	Format	Replacer
1	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)`
2	Field access	`this.a` or `super.a`	The same as No. 1
3	Qualified name	`a.b`	The same as No. 1
4	Method name	`f(...)`	The name of another visible method with compatible parameter and return types
5	Primitive type	e.g., `int` or `float`	A widened type, e.g., `float` to `double`
6	Boolean literal	`true` or `false`	The opposite boolean value
7	Number literal	e.g., `1` or `0.5`	The number literal located in the same method
8	Infix operators	e.g., `+` or `>`	A compatible infix operator, e.g., `+` to `-` , `>` to `>=`
9	Prefix/Postfix operators	e.g., `++` or `--`	The opposite prefix/postfix operator, e.g., `++` to `--`
10	Assignment operators	e.g., `+=` or `*=`	The opposite assignment operator e.g., `+=` to `-=` , `*=` to `\=`
11	Conditional expression	`a ? b : c`	`b` or `c`

defined by the templates in Table 1 one by one, with each transformation deriving a candidate statement either for replacement or for insertion that can be put into R_j or I_j. Fig. 10 illustrates the process. Note that in order to avoid combinatorial explosion, we apply a template to only a single AST node at a time in each transformation. For example, we do not simultaneously modify `a` and `callX` in `a.callX()`.

So unlike PAR which applies templates during search, the repair templates are invisible to the evolutionary search in our approach. Instead, it applies the templates to the LBSs to arrive at statement-level transformations ahead of time, then hands over this space of transformations to the search procedure. The benefit of such a strategy is that we can encode a patch in the combined search space (determined by statement-level redundancy assumption and repair templates) with a unified lower-granularity patch representation (see Section 3.3.1) that decouples the partial information of an edit. As found by very recent studies [74, 108], lower-granularity representations can lead to improved search ability for GP compared to the canonical one shown in Fig. 1(a).

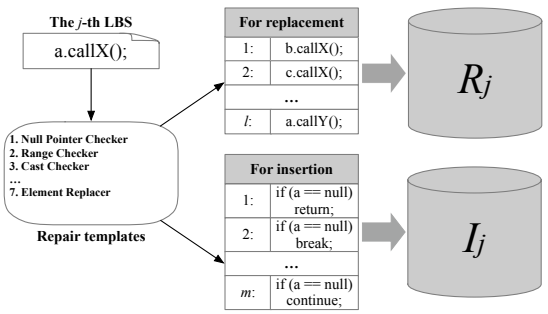


Fig. 10. Illustration of the execution of repair templates ahead of the search.

3.2.3 Initialization of Operation Types. Sections 3.2.1 and 3.2.2 have described how to determine R_j and I_j for the j-th LBS. Now we consider how to determine O_j (i.e., the set of available operation types for the j-th LBS).

First, it is obvious that if R_j (or I_j) is an empty set, the replacement (or insertion) operation should not be contained in O_j. As for the deletion operation, it should be executed carefully because it can easily lead to one of the following two problems: It can (i) cause a compiler error of the modified program; or (ii) generate overfitting patches [79]. To address the first problem, we use the two rules defined in [108], 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`, the deletion operation will not be included in O_j for this LBS. To address the second problem, we use the 5 anti-delete patterns defined in [89] (see Table 2 in [89]). If a LBS follows any of the 5 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.

Suppose $O = \{\text{“Delete”}, \text{“Replace”}, \text{“Insert”}\}$ is the universal set of statement-level operation types. With the above procedure, we can determine O_j for the j-th LBS, where $O_j \subseteq O$ and $j \in \{1, 2, \ldots, n\}$.

3.3 Searching for Patches by Multi-Objective Evolution

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

A solution (i.e., a patch) to the program repair problem is encoded as $\mathbf{x} = (\mathbf{b}, \mathbf{u}, \mathbf{p}, \mathbf{q})$, which contains four different parts each being a vector of size n. In the solution $\mathbf{x}$, $b_j \in \{0, 1\}$ indicates whether the j-th LBS is to be edited or not; $u_j \in \{1, 2, \ldots, |O_j|\}$ indicates the u_j-th operation type in O_j is used for the j-th LBS; $p_j \in \{1, 2, \ldots, |R_j|\}$ means that if replace operation is used, the p_j-th statement in R_j will be selected to replace the j-th LBS; and $q_j \in \{1, 2, \ldots, |I_j|\}$ means that if insert operation is used, the q_j-th statement in I_j will be inserted before the j-th LBS. Fig. 11 illustrates the new lower-granularity patch representation. Suppose the j-th LBS is `a.callX();` in this figure, then the edit on the j-th LBS is: replace `a.callX();` with `b.callX();`.

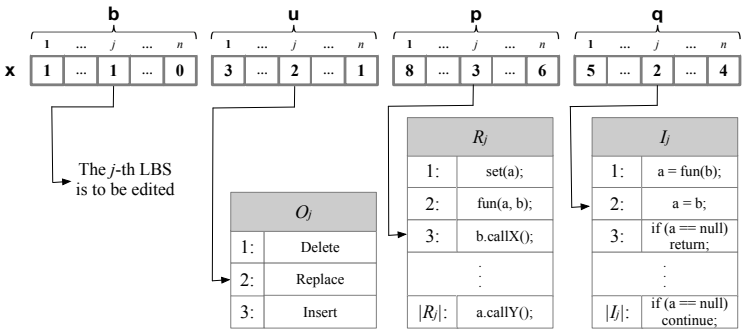


Fig. 11. Illustration of the new lower-granularity patch representation.

Different from the patch representation in ARJA (see Fig. 2(a)), replacement and insertion statements have now been decoupled into two evolvable sub-structures (i.e., $\mathbf{p}$ and $\mathbf{q}$).

3.3.2 Finer-Grained Fitness Function. To evaluate the fitness of an individual $\mathbf{x}$, we again use a bi-objective function as in the original ARJA [108]. The first objective (i.e., $f_1(\mathbf{x})$) is to minimize the size of the patch, defined in Eq. (1). The second objective (i.e., $f_2(\mathbf{x})$) is to minimize the weighted failure

rate. Different from Eq. (2), we compute $f_2(\mathbf{x})$ through finer-grained analysis of test execution in this study, in order to provide a smoother gradient for the genetic search to navigate the search space. Since our repair system targets Java, our implementation is based on the JUnit [16] framework.

Table 3. The Description of Typical Assertion Methods in JUnit

No.	API	Description
1	`assertEquals(double expected, double actual)`	Asserts that two doubles are equal.
2	`assertEquals(double expected, double actual, double delta)`	Asserts that two doubles are equal to within a positive delta.
3	`assertNotEquals(double unexpected, double actual, double delta)`	Asserts that two doubles are not equal to within a positive delta.
4	`assertNotEquals(long unexpected, long actual)`	Asserts that two longs are not equals.
5	`assertArrayEquals(double[] expecteds, double[] actuals, double delta)`	Asserts that two double arrays are equal to within a positive delta.
6	`assertArrayEquals(long[] expecteds, long[] actuals)`	Asserts that two long arrays are equal.
7	`assertTrue(boolean condition)`	Asserts that a condition is true.
8	`assertSame(Object expected, Object actual)`	Asserts that two objects refer to the same object.

In JUnit, assertion methods are all static and defined in the `Assert` class. Table 3 describes typical assertion methods in JUnit. Most the other assertion methods can be regarded as similar to one of them. For example, for the first method in Table 3, there are similar versions with the types `int`, `short`, `char`, etc. According to the insights in Section 2.2.4, we need to define a metric to measure the degree of violation for each assertion, which we call *assertion distance*. Suppose an assertion (denoted by e) invokes the third method in Table 3 like this: `assertEquals`(x, y, δ), then the assertion distance $d(e)$ is computed as:

$$d(e) = \begin{cases} v(|x-y|-\delta), & |x-y| \geq \delta \\ 0, & |x-y| < \delta \end{cases} \tag{5}$$

Here, $v(x)$ is a normalizing function in $[0, 1]$ and we use the one suggested in [3]: $v(x) = x/(x+1)$. We can use similar procedures to compute assertion distances for the other assertion methods.

Now we can formally define the finer-grained version of fitness function $f_2(\mathbf{x})$. First, after executing a program variant $\mathbf{x}$ over a test case t, we can compute a metric $h(\mathbf{x}, t) \in [0, 1]$ to indicate how badly $\mathbf{x}$ has failed test case t based on the collected assertion distances. This metric is defined as follows:

$$h(\mathbf{x}, t) = \frac{\sum_{e \in E(\mathbf{x}, t)} d(e)}{|E(\mathbf{x}, t)|} \tag{6}$$

where $E(\mathbf{x}, t)$ is the set of executed assertions by $\mathbf{x}$ over t, and $d(e)$ is the assertion distance for assertion e. Note that $E(\mathbf{x}, t)$ can only be determined through running t and is not equal to the number of assertions contained in t, since the assertions could be located in control statements (e.g., `if` or `for` statements). Based on $h(\mathbf{x}, t)$, $f_2(\mathbf{x})$ in Eq. (2) can be reformulated as follows:

$$f_2(\mathbf{x}) = \frac{\sum_{t \in T_{pos}} h(\mathbf{x}, t)}{|T_{pos}|} + w \times \left(\frac{\sum_{t \in T_{neg}} h(\mathbf{x}, t)}{|T_{neg}|} \right) \tag{7}$$

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

In the original ARJA system [108], mutation was applied to each part independently. However, due to genetic redundancy, this kind of mutation usually does not change the phenotype of the genome. For example, if $b_j = 0$ and is not mutated, then any mutation to u_j, p_j and q_j will have no effect. Here we apply a mutation directed at a single selected LBS. To be specific, we first use roulette wheel selection to choose a LBS, where the j-th LBS is chosen with a probability of $susp_j / \sum_{j=1}^{n} susp_j$; suppose that the j-th LBS is finally selected, then we apply a bit flip mutation to b_j and a uniform mutation to u_j, p_j and q_j, respectively.

Fig. 12 illustrates the crossover and mutation operations, where only a single offspring is shown for brevity.

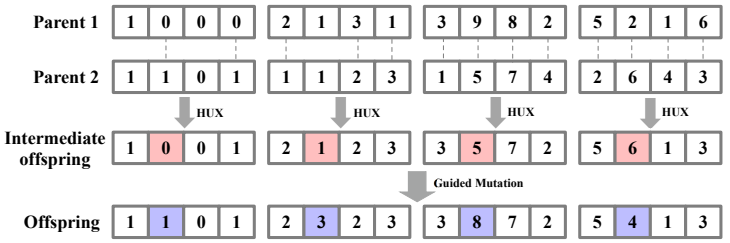


Fig. 12. Illustration of the crossover and mutation.

3.3.4 Computational Search. With the patch representation, fitness function and genetic operators designed above, any multi-objective evolutionary algorithm can serve the purpose of searching for patches. In this work, we employ the NSGA-II [21] algorithm as the multi-objective search framework. To initialize the population, we combine the fault localization result and randomness: for the first part (i.e., **b**), b_j is initialized to 1 with a probability of $susp_j \times \mu$, where $\mu \in (0, 1)$ is a predefined parameter; the remaining three parts (i.e., **u**, **p**, **q**) are initialized randomly. After population initialization, the search algorithm iterates over generations until the stopping criterion is satisfied.

In the g-th generation, genetic operators are first applied to the current population P_g of size N to generate an offspring population Q_g of the same size. Then we remove duplicates in the union population $U_g = P_g \cup Q_g$ (i.e., if more than one individual in U_g represent the same patch, only one of them is kept and the others are discarded). Last, fast non-dominated sorting with crowding distance comparison [21] is used to select N individuals from U_g to constitute the population P_{g+1} in the next generation. Note that here we use an additional procedure to remove duplicates in order to further prevent premature convergence, which is different from the original NSGA-II.

Once the search has terminated, the non-dominated solutions with $f_2 = 0$ in the final population are output as plausible patches. If no such solution exists, our approach fails to fix the bug.

3.4 Alleviating Patch Overfitting

To alleviate patch overfitting, we developed a post-processing tool which can analyze the plausible patches found by our repair approach further. This tool can be used for two purposes: overfit detection and patch ranking.

3.4.1 Overfit Detection. For overfit detection, we take a buggy program, a set of positive test cases and a plausible patch as input, and determine whether or not this plausible patch is an overfitting patch. Our overfit detection approach is called CIP, which is based on the assumption that the buggy program will perform **c**orrectly on the test **i**nputs encoded in the **p**ositive test cases.

Fig. 13 shows the overall process of CIP. First, given a plausible patch and a buggy program, we can localize the methods where the statements will be modified by the patch. Then we instrument the bytecode of these methods in the buggy program. With the instrumented buggy program, we run the positive test cases so that we can capture a number of input-output pairs for the localized methods. Suppose that there are K such pairs, denoted by a set $PA = \{(In_1, Out_1), (In_2, Out_2), \ldots, (In_K, Out_K)\}$. According to our assumption, all these input-output pairs should reflect the correct program behavior. In order to judge patch correctness, we will feed these inputs $In_1, In_2, \ldots, In_K$ into the corresponding methods in the patched program so as to see whether the correct outputs can be obtained. More specifically, we first apply the plausible patch to the buggy program to obtain a patched program. Then similar to the instrumentation of the buggy program, we instrument those localized methods in the patched program in order to capture the method outputs at runtime. Next, for each input-out pair $(In_i, Out_i) \in PA$ collected previously, we deserialize In_i from the file and use the Java reflection technique to invoke the corresponding method in the instrumented patched program with the deserialized input In_i, so that we can collect the method output Out'_i. Lastly, we compare every Out'_i with the corresponding Out_i, and if there exists any difference, we identify the plausible patch as an overfitting patch that is incorrect.

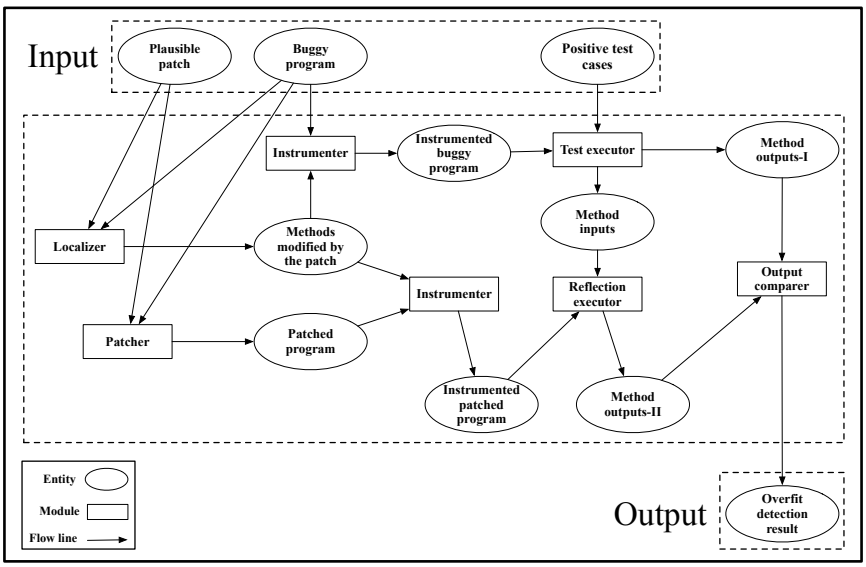


Fig. 13. The overview of CIP.

3.4.2 Patch Ranking. Patch ranking aims to rank a number of plausible patches for a bug. The patch ranked higher is more likely to be correct. Suppose that a plausible patch is denoted by $\mathcal{P}$ and it modifies D LBSs with the IDs $i_1, i_2, \ldots, i_D$. For the i_j-th LBS, the corresponding type of repair action in $\mathcal{P}$ is denoted by t_j. In our repair system, $t_j \in \{\text{NPC}, \text{RC}, \text{CC}, \text{DC}, \text{MPA}, \text{BEAR}, \text{ER}, \text{SR}, \text{SI}, \text{SD}\}$,

where SR and SI refer to statement replacement and statement insertion based on the statement-level redundancy assumption, respectively; SD means statement deletion; and the others are all template-based repair actions which can be referred to in Table 1.

For the ranking purpose, we first define three metrics for $\mathcal{P}$. The first metric represents the summation of the suspiciousness for the LBSs modified by $\mathcal{P}$, which is defined as

$$Susp(\mathcal{P}) = \sum_{j=1}^{D} susp_{i_j} \tag{8}$$

The second metric is based on CIP. Recall that for the purpose of overfit detection, we only need to know whether there is a difference between Out_i and Out'_i, where $i = 1, 2, \ldots, K$. Here we want to quantify such a difference. To do this, we deserialize Out_i and Out'_i, and extract all primitive data and string data contained in the two outputs in a recursive way. Similar to the computing of assertion distance, we can easily compute the distance for each corresponding primitive/string data in Out_i and Out'_i, and normalize it to $[0, 1]$. Then we compute the average of these normalized distances between the primitive/string data and define it as the distance between Out_i and Out'_i, denoted by $dist(Out_i, Out'_i)$. With the definition of $dist(Out_i, Out'_i)$, we define the second metric for $\mathcal{P}$ as follows:

$$Dist(\mathcal{P}) = \frac{1}{K} \sum_{i=1}^{K} dist(Out_i, Out'_i) \tag{9}$$

This measures how close the method outputs obtained by the patched program are to the expected outputs. Smaller *Dist* is better.

Before defining the third metric, we determine a preference relation of repair action types in our system: NPC/RC/CC/DC $\prec$ MPA $\prec$ ER $\prec$ BEAR $\prec$ SR/SI $\prec$ SD. The rationality of this preference relation is explained as follows:

(1) Template-based repair actions are preferred over repair actions based on the statement-level redundancy assumption, since they are usually more targeted.
(2) As indicated by several empirical studies [61, 79], SD can easily result in overfitting. So we regard SD as the least preferred repair action.
(3) NPC, RC, CC and DC are similar and they do not introduce regressions in general, so these repair actions are the most preferred.
(4) MPA is preferred over ER, because ER can apply to more AST node types and thus has a higher chance of overfitting the test suite.
(5) BEAR adds or removes a boolean expression, so it usually leads to larger syntactic changes than ER. Given this, ER is preferred over BEAR.

To quantify the preference, we assign a score for each type of repair action: SD is scored 1, SR and SI are scored 2, BEAR is scored 3 and so on. With these scores, the third metric is defined as the sum of scores of repair action types contained in $\mathcal{P}$:

$$Pref(\mathcal{P}) = \sum_{j=1}^{D} score(t_j) \tag{10}$$

where $score(t_j)$ is the score of repair action type t_j.

With the three metrics, we use a heuristic multi-level procedure to rank the plausible patches. The rules for comparing two patches are as follows, which should be applied in sequence:

(1) The patch with higher *Susp* is ranked higher.
(2) If *Susp* values of two patches are equal, *Dist* values are compared, and the patch with smaller *Dist* is ranked higher.

(3) If *Dist* values of the two patches are also equal, $Pref$ values are compared, and the patch with higher $Pref$ is ranked higher.
(4) If *Susp*, *Dist* and $Pref$ cannot distinguish two patches, the patch found earlier is ranked higher.

Here *Susp* is considered with the highest priority, because a bug can be correctly fixed only when we find where the bug is in the first place. *Dist* is a kind of semantic distance while $Pref$ is largely a syntactic-level metric, so we favor *Dist* over $Pref$ in comparison.

3.5 Discussion

In this section, we will discuss two issues: (i) other possible integration schemes for our repair approach; and (ii) when to use overfit detection or patch ranking in practice.

3.5.1 Other Possible Integration Schemes. In ARJA-e, potential fix ingredients come from two sources: the statement-level redundancy assumption and repair templates. It is possible to use source code repositories as an additional source just like SearchRepair [36] and ssFix [99]. The technical difficulty lies in the question of how to extract promising fix code from a large amount of source code. It is also possible to exploit information derived from Javadoc comments just like ACS [101] does to generate more accurate conditions for instantiating template BEAR.

ARJA-e conducts a light-weight contextual analysis to select replacement and insertion statements. In this phase, it is possible to leverage value/control-flow analysis [102] or machine learning techniques [57, 82] to refine the definition of replacement similarity and insertion relevance.

In ARJA-e, we use a finer-grained fitness function based on the assertion distance. It is possible to further improve this fitness function by combining it with external information such as from bug reports [82] or historical bug fixes [43].

As for the GP algorithm, the underlying GP in GenProg can also be used to explore ARJA-e's search space. However, as indicated by recent studies [74, 108], a GP based on this high-granularity patch representation (see Fig. 1(a)) can lead to inadequate search ability. We shall demonstrate this later with experiments in the context of ARJA-e's search space.

Note that it is non-trivial to make these possible integrations really work. For example, just incorporating more sources for fix ingredients may hurt the performance if the search space reduction strategy or the search algorithm is not effective enough. Overall, the final repair performance depends on how well we address the three challenges mentioned in Section 1.

3.5.2 Overfit Detection vs. Patch Ranking. For a single bug, evolutionary repair approaches can usually return a number of plausible patches. Overfit detection aims to filter out incorrect patches. However, it is worth noting that even the state-of-the-art overfit detection approaches can only identify about 50% incorrect patches [98, 100]. This implies that, after overfit detection, there may still be many remaining patches which require confirmation by a human developer. So we think that it is more suitable to use overfit detection when the human developer has enough time and energy and the correct fixing of the bug is prioritized. In contrast, if the human developer can only afford to examine very few or even a single patch, it may be more suitable to use patch ranking. However there is a risk that the correct patch is not ranked first or very high, because in order to completely distinguish patches, patch ranking usually exploits certain heuristic rules, which may not be very reliable for indicating the possibility of patch overfitting.

We would like to emphasize that CIP is independent of our actual repair approach. That is, it can be applied to patches generated by any repair approach. Our patch ranking technique uses a metric $Pref$, which depends on the types of repair actions used in the repair approach. Adapting $Pref$ accordingly can make this ranking technique easily applicable to other repair approaches.

4 TOOL IMPLEMENTATION

In this section, we describe several implementation issues regarding the assertion distance computation and overfit detection. Readers can refer to our previous work [108] for the libraries or frameworks we used to implement the other building blocks of ARJA-e, including fault localization, test filtering, source code parsing/manipulation, and multi-objective GP.

4.1 Assertion Distance Computation

For computing assertion distance, some notable details for implementation are as follows:

(1) For an assertion comparing arrays (e.g., No. 5 in Table 3), we compute a distance for each dimension like Eq. (5). Then the final assertion distance is the average of these distances.
(2) For the comparison of two strings, we use the Levenshtein distance.
(3) For an assertion comparing two objects (e.g., `assertEquals(Object expected, Object actual)`), we consider the situation that the objects are the instances of `Number`, `String` or `Character`, where the assertion distance can be computed as that for basic data types.
(4) For `assertTrue` or `assertFalse`, we consider the situation that the asserted condition is a comparison expression. To record the assertion distance in this case, we use the bytecode testability transformation technique [50] to instrument the comparisons.
(5) For an assertion comparing object references (e.g., No. 8 in Table 3), only a binary signal can be provided. So the assertion distance is just set to maximum (i.e., 1) when the assertion fails.

For the concrete implementation, we provide a proxy method for every JUnit assertion method in a newly defined class (e.g., `AssertTracer`). As an illustration, Fig. 14 shows the proxy method for `assertEquals(double expected, double actual, double delta)`. This implementation avoids conducting a rewrite from scratch. Instead, we first call the original assertion method (line 6 in Fig. 14). Since the JUnit assertion method will throw an exception once the assertion fails, we can capture this exception and calculate the assertion distance in the `catch` block (lines 9–11 in Fig. 14). With `AssertTracer`, we modify the bytecode of the given test suite, that is, we replace all the invocations of JUnit assertion methods with the invocations of the corresponding proxy methods in `AssertTracer`, which is realized using a Java bytecode manipulation tool called ASM [12]. When evaluating a program variant on a test case during evolution, we run it over the modified bytecode of the test case so that the assertion distances can be collected.

4.2 Overfit Detection

In overfit detection, the bytecode instrumentation is based on the ASM [12] library. For each method that is modified by the patch, the instrumentation is conducted at its entry point and all its possible exit points. At the entry point, we inject new bytecode to save the *input* of the method, including all method parameters and the current object `this` (i.e., the object whose method is being called), into a file. To save the objects, we leverage the Java serialization technique. This technique can convert the object state into a byte stream that can be reverted back into a copy of the object. In our implementation, we use Kryo [17] as the serialization framework.

Note that if In_i contains instances of non-static inner classes, deserialization of In_i is error prone due to several technical reasons [75]. Should an error occur during deserialization, we immediately give up using Java reflection. Instead, we run the positive test cases over the instrumented patched program to collect another set of input-output pairs $PB = \{(In'_1, Out'_1), (In'_2, Out'_2), \ldots, (In'_L, Out'_L)\}$. For each pair $(In_i, Out_i) \in PA$, we check whether there is a pair $(In'_j, Out'_j) \in PB$ where In'_j is identical to In_i and both pairs are refer to the same method. If such a pair (In'_j, Out'_j) exists, we compare Out'_j with Out_i. In contrast to Java reflection, this technique may not be able to exploit all

```
public class AssertTracer {
   public static List<Double> distances;
   public static void assertEquals(double expected, double actual, double delta) {
      double dist = 0;
      try {
         org.junit.Assert.assertEquals(expected, actual, delta);
         //call the original JUnit API
      } catch (Throwable t) {
         dist = Math.abs(expected - actual) - delta;
         //calculate the assertion distance
         dist /= (dist + 1); //normalization
      }
      distances.add(dist); //record the assertion distance
   }
   ... ...
}
```

Fig. 14. Illustration of the proxy assertion method that can record assertion distance.

input-output pairs in *PA*, so it is only used when deserialization causes errors. For simplicity, we omit the procedures of this alternative technique in Fig. 13.

5 EXPERIMENTAL DESIGN

5.1 Dataset of Bugs

We perform the empirical evaluation over a database of real bugs, called Defects4J [34], which has been extensively used for evaluating Java repair systems. We use the Defects4J version v1.0.1. Following previous studies [53, 61, 82, 96, 101, 108], we consider four projects in Defects4J, namely Chart, Lang, Time and Math. Table 4 shows descriptive statistics of these four projects, where KLoC is short for kilo (thousands) of lines of code. In total, there are 224 real bugs: 26 from Chart (C1–C26), 65 from Lang (L1–L65), 106 from Math (M1–M106) and 27 from Time (T1–T27). Note that there is another project called Closure in Defects4J. We ignore Closure because it uses the customized testing format instead of the standard JUnit tests [61, 101].

Table 4. The Descriptive Statistics of Defects4J Dataset

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

5.2 Research Questions

In the context of our study, we intend to answer the following research questions (RQs).

RQ1, On the Overall Performance: How does our repair approach perform on real bugs?

This RQ concerns the overall repair performance and is the basis for applying ARJA-e in practice, which consists of five sub-questions as follows:

RQ1.1: How many and which bugs in Defects4J can be fixed and correctly fixed by ARJA-e?
RQ1.2: Can ARJA-e outperform existing repair approaches?
RQ1.3: Can ARJA-e fix multi-location bugs?
RQ1.4: Are the incorrect patches generated by ARJA-e still meaningful?

The remaining RQs are mainly concerned with the components of our repair system.
RQ2, On the Search Space: How reasonable is ARJA-e's search space?
This RQ investigates the effect of the search space used by ARJA-e, which is composed of two detailed sub-questions as follows:

RQ2.1: To what extent do the statement-level redundancy assumption and repair templates contribute to the overall performance of ARJA-e?
RQ2.2: Is it beneficial to use the search space reduction strategy based on replacement similarity and insertion relevance?

RQ3, On the Search Algorithm: How powerful is our search algorithm in finding patches?
This RQ is more concerned with our innovations in fitness function and patch representation. It consists of the following three sub-questions:

RQ3.1: Can the finer-grained fitness function provide better guidance for search?
RQ3.2: Is our GP in ARJA-e more effective than the underlying GP in GenProg?
RQ3.3: What are the benefits of decoupling the statements for replacement and for insertion in the lower-granularity patch representation?

RQ4: On Alleviating Patch Overfitting: How well do our approaches help to alleviate patch overfitting?
This RQ investigates the performance of our approaches for alleviating patch overfitting which are described in Section 3.4. It consists of three detailed sub-questions as follows:

RQ4.1: What percentage of incorrect patches can be identified by CIP?
RQ4.2: What are the advantages of CIP compared to the heuristic approach proposed by Xiong et al. [100]?
RQ4.3: Do all three metrics used in patch ranking contribute to the final ranking performance?

In RQ4.2, Xiong et al.'s approach is selected for comparison, because it is a state-of-the-art approach for overfit detection and it is somewhat similar to CIP.

5.3 Experimental Settings

In Table 5, we show the parameter settings for ARJA-e in the experiments. Each trial of ARJA-e is terminated after it reaches the maximum number of generations (i.e., 50) or its execution time exceeds one hour. Note that crossover and mutation operators presented in Section 3.3.3 are always executed, so their probability (i.e., 1) is omitted in this table. Compared to ARJA, two additional parameters β_{sim} and β_{rel} are involved in ARJA-e. Currently we set β_{sim} to 0.3 and β_{rel} to 0.2, and ARJA-e performs reasonably well with the two threshold values in our experiments. Fine-tuning of β_{sim} and β_{rel} will be left for future work. The other parameters except n_{max} use the same setting as that in ARJA [108]. n_{max} is set to 40 in ARJA. In this work, we increase this value to 60. So ARJA-e generally works over a much larger search space than ARJA not only because it considers a larger number of potential fix ingredients, but also because it considers more possibly faulty locations. Note that without a search algorithm that is powerful enough, larger search spaces do not necessarily lead to better repair performance.

Table 5. The Parameter Setting for ARJA-e in Our Experiments

Parameter	Description	Value
N	Population size	40
G	Maximum number of generations	50
$\gamma_{\min}$	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	0.2
w	Refer to Section 3.3.2	0.5
μ	Refer to Section 3.3.4	0.06

Besides the parameter settings of Table 5, we use several special experimental settings for RQ1, RQ3 and RQ4, since they concern different aspects of our repair system. These settings are explained as follows:

(1) In RQ1, we test ARJA-e on all 224 real bugs described in Table 4. The problem here is how many ARJA-e trials should be run for each bug. If only a single trial is run for each bug, the repair performance of ARJA-e is likely to be largely underestimated given its stochastic nature. However if we run many trials instead, ARJA-e would require too much CPU time for each bug. In this case, it seems unfair to compare the resulting performance of ARJA-e with that of other repair systems which are much less CPU intensive. Moreover, it may also not be practical for a human developer to perform too many trials when applying ARJA-e to a single bug, since time and computing resources are usually limited. Considering all of the above, we decided to run 5 random trials of ARJA-e in parallel for each bug. With this choice, even if we do not have multiple CPUs for parallel computing, the total execution time of ARJA-e for a bug is within 5 hours, which is still comparable to the time budget used by other repair systems in the literature [33, 37, 53, 61, 103].

(2) In RQ3, we will conduct control experiments to investigate whether our new development can enhance the search ability of GP for program repair. Unlike in RQ1, a large enough number of trials for each bug is really required here, in order to properly compare the search ability of different stochastic search algorithms [5]. However, it is very computationally expensive to repeat each search algorithm a large number of times (e.g., 50) on each of the 224 bugs, because the CPU time just for a single algorithm might be up to $224 \times 50 \div 24 \approx 467$ days. While we have access to a high performance computing center (HPCC), such a large time budget is beyond our means. Moreover, for the purpose of distinguishing search ability, it is indeed not meaningful to use bugs whose plausible patches are not in the search space. As a result, we decided to select the multi-location bugs that are fixed by ARJA-e (according to the repair results in RQ1) as the subject in RQ3. The rationale of this selection is as follows:
   (a) Since these bugs can be fixed by ARJA-e, at least one of their plausible patches is in ARJA-e's search space.
   (b) The number of such bugs is reasonably small, so that we can afford to perform a large number of independent trials (50 is used in our experiments) for each of them.
   (c) As indicated by previous studies [69, 79, 108], multi-location bugs pose a greater challenge to search algorithms. So it will be more persuasive to test the search ability on these bugs.

(3) In RQ4.1 and RQ4.2, we want to verify the effectiveness of CIP. Different from the previous RQs, the two sub-RQs take the plausible patches as the subject. We consider the first plausible patch found by ARJA-e for each bug (according to RQ1). In addition, we include the patches generated by jGenProg and jKali, which are collected from Martinez et al.'s empirical study

[61] on Defects4J. However, not all of these patches can be used. For some of them, we are unable to judge their correctness. While for some others, the positive test cases do not visit any modified method, so CIP cannot handle such patches. Here we ignore these unsupported patches. In the end, we collect a dataset of 122 plausible patches, where 97 patches are incorrect and 25 patches are correct. The correctness of ARJA-e patches is judged by ourselves, while the correctness of jGenProg and jKali patches is according to Martinez et al.'s analysis [61]. Table 6 shows the statistics of this dataset.

Table 6. Dataset of Plausible Patches Used in RQ4

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

Note that patch correctness is relevant in RQ1, RQ2 and RQ4. Following previous work [15, 31, 61, 82, 96, 99, 108], we manually examine the correctness of the plausible patches found by our repair approach. We identify a patch as correct if it is exactly the same as or semantically equivalent to a human-written patch. To ensure confidence, we avoid complex semantic analysis and ignore those patches beyond our understanding in the manual analysis.

All the experiments are conducted on the MSU HPCC [18] and use the Intel Xeon E5-2680 2.4 GHz CPU with 20 GB memory. The source code of our repair system has been made available online [109], along with the patches generated by ARJA-e and the dataset of patches used for overfit detection.

6 RESULTS AND DISCUSSIONS

In this section, we present the results of our experimental study in order to address the research questions set out in Section 5.2.

6.1 On the Overall Performance

6.1.1 Performance Evaluation on Defects4J. For each of the 224 bugs in Defects4J, we ran 5 ARJA-e trials in parallel and collected non-dominated plausible patches found in the 5 trials as output.

According to our results, ARJA-e can find plausible patches for 106 bugs. Fig. 15(a) shows the distribution of the number of plausible patches obtained for each bug. As can be seen from Fig. 15(a), ARJA-e returns a single patch for 30 bugs and at least two patches for the remaining 76 bugs; for most of bugs (i.e., 82 out of 106), there are less than 10 patches. To rank the plausible patches for each bug, we use the technique described in Section 3.4.2. Fig. 15(b) shows the rank distribution of correct patches. If there are multiple correct patches for a bug, we only consider the one that is ranked highest. We can see from Fig. 15(b) that the correct patch is ranked first for 39 bugs, ranked 5th for 2 bugs, and ranked 2nd, 7th and 16th for 3 bugs respectively. It can be concluded that our patch ranking technique is quite effective ranking the overwhelming majority of correct patches as first.

In Table 7, we report the bugs that can be fixed (i.e., plausible patches are found) and those that can be correctly fixed by ARJA-e, respectively. Note that here we use a strict criterion for judging whether a bug is correctly fixed by ARJA-e, that is, a bug is regarded as being correctly fixed only

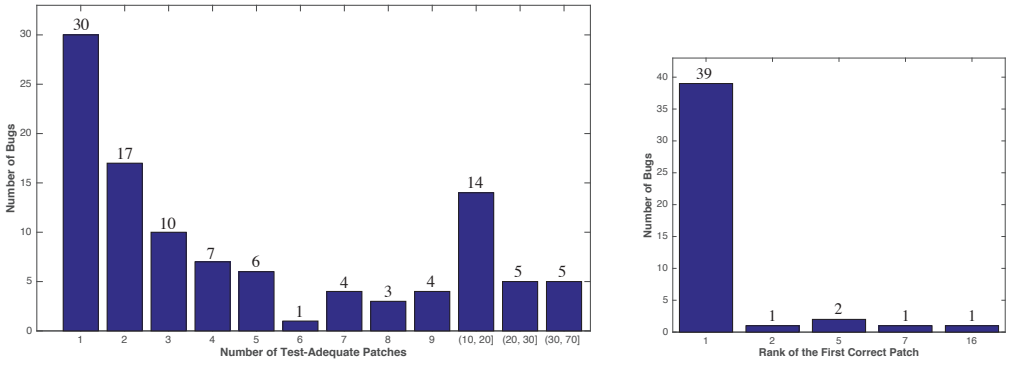


Fig. 15. (a) Distribution of the number of plausible patches; and (b) Distribution of the rank of correct patches.

Table 7. List of the Bugs Fixed and Correctly Fixed by ARJA-e

Project	Plausible	Correct
Chart	C1, C3, C4, C5, C6, C7, C10, C11, C12, C13, C14, C15, C17, C18, C19, C24, C25, C26	C3, C4, C5, C10, C11, C12, C24
	$\sum = 18$	$\sum = 7$
Lang	L1, L7, L10, L14, L15, L16, L20, L21, L22, L24, L27, L33, L34, L39, L41, L43, L44, L45, L46, L50, L51, L55, L57, L58, L59, L60, L61, L63	L7, L20, L24, L33, L34, L39, L43, L44, L61
	$\sum = 28$	$\sum = 9$
Math	M2, M3, M4, M5, M6, M7, M11, M22, M24, M25, M28, M30, M31, M32, M34, M39, M40, M42, M44, M49, M50, M53, M56, M57, M58, M62, M63, M64, M65, M67, M68, M70, M71, M73, M74, M75, M77, M78, M79, M80, M81, M82, M84, M85, M86, M88, M89, M94, M95, M98, M104	M4, M5, M11, M22, M25, M30, M34, M39, M53, M56, M57, M58, M65, M70, M73, M75, M79, M86, M89, M94, M98
	$\sum = 51$	$\sum = 21$
Time	T4, T7, T9, T11, T14, T15, T17, T20, T24	T7, T15
	$\sum = 9$	$\sum = 2$
Total	106 (47.32%)	39 (17.41%)

when the plausible patch ranked first is correct. In addition, among the 39 correct patches ranked first, 13 are exactly the same as the human-written patch, while the remaining 26 are semantically equivalent.

RQ1.1: How many and which bugs in Defects4J can be fixed or correctly fixed by ARJA-e?
Answer to RQ1.1: Among the 224 bugs considered in Defecsts4J, ARJA-e can fix 106 bugs (accounting for 47.32%) in terms of passing all test cases, and can correctly fix 39 bugs (accounting for 17.41%) according to the patches ranked first. The detailed IDs of the fixed bugs can be found in Table 7.

6.1.2 Comparison with Existing Approaches. To show the superiority of ARJA-e over the state of the art, we compare ARJA-e with 7 recent repair approaches that have been tested on Defects4J.

These approaches are ACS [101], ssFix [99], ELIXIR [82], ARJA [108], SimFix [33], CAPGEN [96], and SOFIX [53]. Note that in the literature, the time budget for each repair attempt varied between these approaches: 30 minutes for ACS, 1.5 hours for ELIXIR and CAPGEN, 2 hours for ssFix, 3 hours for SOFIX, and 5 hours for SimFix. For ARJA, the time budget was not specified explicitly in the corresponding study [108]. Due to different algorithm characteristics, it is not meaningful to compare these time budgets. For example, ACS set a small time budget (i.e., 30 minutes) since it works over a much smaller search space by only aiming at condition synthesis, and a larger time budget may not help to improve its repair performance.

Table 8. Comparison with Existing Repair Tools in terms of the Number of Bugs Fixed and Correctly Fixed (Plausible/Correct). The Best Results are Shown in Bold

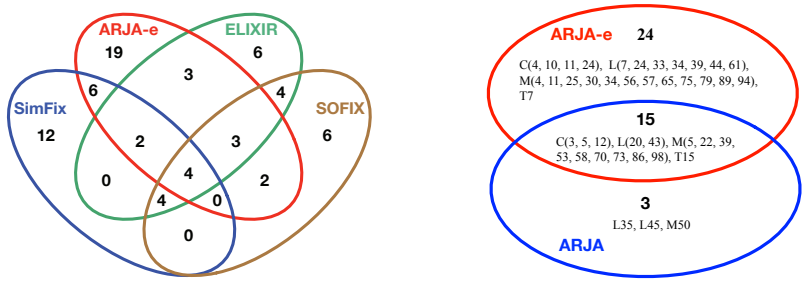
Project	ARJA-e	ACS	ssFix	ELIXIR	ARJA[1]	SimFix	CAPGEN	SOFIX
Chart	**18/7**	2/2	7/2	7/4	9/3	8/4	–/4	–/5
Lang	**28/9**	4/3	12/5	12/8	17/4	13/**9**	–/5	–/4
Math	**51/21**	16/12	26/7	19/12	29/10	26/14	–/12	–/13
Time	**9/2**	1/1	4/0	3/2	4/1	1/1	–/0	–/1
Total	**106/39**	23/18	49/14	41/26	59/18	48/28	–/21	–/23

"–" means the data is not available since it is not reported by the original authors.
[1] In ARJA, a bug is regarded as being fixed correctly if one of its plausible patches is identified as correct.

Table 8 shows the comparison results. From Table 8 we can see that ARJA-e outperforms all other approaches in terms of the number of correctly fixed bugs. We further compare ARJA-e against ELIXIR, SimFix and SOFIX by analyzing the overlaps among their repair results. ELIXIR, SimFix and SOFIX are selected because they perform best in correct bug fixing among the 7 compared approaches. Fig. 16(a) shows the intersection of correctly fixed bugs between ARJA-e, ELIXIR, SimFix and SOFIX, using a Venn diagram. From Fig. 16(a), ARJA-e fixes the highest number of bugs correctly (i.e., 39), where 19 bugs cannot be fixed correctly by any of the other three approaches. So ARJA-e indeed complements the three approaches very well. But it should be noted that the three approaches also show good complementarity to ARJA-e in terms of correct bug fixing. Specifically, ELIXIR, SimFix and SOFIX can correctly fix 14, 14 and 16 bugs that cannot be correctly fixed by ARJA-e, respectively. To figure out why our approach cannot correctly fix these bugs, we further examine the patches reported by the three approaches. We can summarize as follows:

(1) For L6, M33 and M59, there exist correct patches in ARJA-e's search space, but ARJA-e fails to generate any plausible patch. Further enhancing the search algorithm may make ARJA-e correctly fix these bugs.
(2) For C1, C7, M82 and M85, ARJA-e can only find overfitting patches although the correct patches are also located in its search space. A search space reduction strategy with deeper program analysis may help in this case, since more overfitting patches can be removed from the search space and the search algorithm can be better focused on correct patches.
(3) For L57, L59, M80 and T4, ARJA-e can find the correct patches, but they are not ranked first. So a better patch ranking procedure needs to help in this case.
(4) For the other bugs, the correct patches obtained by ELIXIR, SimFix or SOFIX are not in ARJA-e's search space. The major reason is that the three approaches use special kinds of program transformations that cannot be performed by ARJA-e. That is, ELIXIR can synthesize new method invocations for insertion, SimFix can use a code snippet larger than a single statement for replacement, and SOFIX introduces several characteristic templates (e.g., BinaryOperator Inversion). It seems possible to further enhance the performance of ARJA-e by incorporating

these program transformations. However, this would make the search space much larger, so a more powerful search algorithm may be necessary for supporting such an incorporation.



(a) ARJA-e, ELIXIR, SimFix and SOFIX (b) ARJA-e and ARJA

Fig. 16. Venn diagram of correctly repaired bugs.

Fig. 16(b) compares ARJA-e with our previous technique ARJA in terms of correct bug fixing. As can be seen, the performance of ARJA-e dominates that of ARJA. This indicates that ARJA-e inherits almost all repair power from ARJA with significant performance enhancement. The major reason is that ARJA-e explores a more promising search space using a more powerful search algorithm. ARJA can correctly fix 3 bugs (i.e., L35, L45 and M50) that cannot be fixed by ARJA-e. For L45, ARJA-e can find a correct patch but this patch is ranked second. For L35, the correct patch found by ARJA contains an edit which replaces an assignment statement `type = Object.class;` with a thrown exception. As illustrated in Section 2.2.3, we regard such kind of an operation as usually unpromising for bug repair, so this patch does not exist in ARJA-e's search space. As for M50, ARJA deletes a whole `if` statement, but this repair pattern is disabled in ARJA-e since ARJA-e incorporates anti-delete patterns introduced in [89] to inhibit patch overfitting.

Note that jGenProg [61, 63], xPAR [43] and HDRepair [43] are evolutionary repair approaches that have often been compared in the literature. However all of them show much worse repair performance than ARJA-e on Defects4J. Although HDRepair can find correct patches for 16 bugs, only 10 of them were ranked first. jGenProg and xPAR can only correctly fix 5 and 3 bugs, respectively.

RQ1.2: Can ARJA-e outperform existing repair approaches?
Answer to RQ1.2: ARJA-e outperforms 7 recent repair approaches by a considerable margin. Specifically, by comparison with the best result, ARJA-e can can correctly fix 39.3% more bugs than SimFix (from 28 to 39). Moreover, ARJA-e is an effective approach complementary to the state-of-the-art techniques.

6.1.3 Results from Multi-Location Bugs. Compared to non-evolutionary repair approaches, a major advantage of evolutionary repair approaches is that they can naturally handle multi-location bugs since they can modify multiple LBSs simultaneously. However, most of the evolutionary repair approaches (e.g., GenProg and PAR) cannot fix multi-location bugs in practice [43, 61, 79], possibly due to inadequate search algorithm [74, 108].

According to our results, ARJA-e finds multi-edit plausible patches for 22 bugs. By using delta debugging, we have verified that none of these patches can be reduced to a single edit. So the 22 bugs should be regarded as multi-location bugs. Furthermore, 7 out of the 22 bugs are classified as being correctly fixed. Table 9 lists the detailed IDs of multi-location bugs that can be fixed and correctly fixed by ARJA-e. Our previous approach ARJA can fix 14 multi-location bugs in terms of

Table 9. List of the Multi-Location Bugs Fixed and Correctly Fixed by ARJA-e

Project	Plausible	Correct
Chart	C14, C18	
	$\sum = 2$	$\sum = 0$
Lang	L1, L14, L15, L20, L34, L41, L46, L50, L55, L61	L20, L34, L61
	$\sum = 10$	$\sum = 3$
Math	M4, M7, M22, M24, M62, M64, M67, M77, M98	M4, M22, M98
	$\sum = 9$	$\sum = 3$
Time	T15	T15
	$\sum = 1$	$\sum = 1$
Total	22	7

fulfilling the test suite, and 5 of them are correctly fixed. However, almost all of these multi-location bugs can also be (correctly) fixed by ARJA-e. It is worth highlighting that ARJA-e can correctly fix 3 new multi-location bugs (i.e., L34, L61 and M4) compared to ARJA. L34 and L61 have never been fixed correctly by existing approaches in the literature as far as we know.

Note that although not specifically claimed in the original studies [33, 101], both ACS and SimFix can correctly fix 6 multi-location bugs according to the patches available online. However, there is a significant difference between ARJA-e and the two approaches in which multi-location bugs are being fixed. Specifically, there is no multi-location bug that can be correctly fixed by both ARJA-e and ACS, and there is only a single multi-location bug (i.e., M98) that can be correctly fixed by both ARJA-e and SimFix.

```
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. 17. Correct patch generated by ARJA-e for bug L34.

To further understand the strength of ARJA-e in multi-location repair, Fig. 17 shows a correct patch found by ARJA-e for bug L34. To correctly fix this bug, ARJA-e uses two kinds of templates for two LBSs respectively: ER for lines 3–4 and NPC for line 10. The 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-e.

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

```
1  // StrBuilder.java
2  public int indexOf(String str, int startIndex) { ...
3  -    char[] thisBuf = buffer;
4  +    char[] thisBuf = toCharArray();
5       int len = thisBuf.length - strLen;
6  -    outer: for (int i = startIndex; i < len; i++) {
7  +    outer: for (int i = startIndex; i <= len; i++) {
8            ...
9       } ...
10 }
```

Fig. 18. Correct patch generated by ARJA-e for bug L61.

creative way. Take L61 for example. Fig. 18 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 the line 3 with `toCharArray()` that copies all the 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 should be, according to the human-written patch. To address this, ARJA-e further changes `i < len` to `i <= len`, achieving semantic equivalence.

RQ1.3: Can ARJA-e fix multi-location bugs?
Answer to RQ1.3: ARJA-e can fix 22 multi-location bugs in terms of passing all test cases. Among the 22 bugs, 7 are identified as being correctly fixed in the manual study.

6.1.4 Analysis of Incorrect Patches. As indicated by Qi et al. [79], the overwhelming majority of incorrect patches generated by GenProg, RSRepair [77] and AE [93] are nonsensical patches that are equivalent to a single functionality deletion. We would like to investigate whether this is also the case for ARJA-e by analyzing the patches ranked first.

Our analysis finds that ARJA-e generates incorrect patches that are equivalent to a single functionality deletion only for 4 bugs (i.e., C1, C6, L22 and M85). Moreover, at least for bugs L10, L27, L59, L60, M63 and M104, the patches generated by ARJA-e partially fix the bug and introduce no regressions, which are called *valid patches* by Xin and Reiss [99]; at least for bugs C19, L45 and L57, the patches by ARJA-e introduce regressions but completely fix the bug. In general, the two kinds of patches are semantically close to the correct patches. Take L27 as example, ARJA-e generates a valid patch as shown in Fig. 19. Compared to the valid patch by ARJA-e, the human-written patch for L27 conducts another modification which replaces the predicate at line 5 (i.e., `expPos < decPos`) with `expPos < decPos || expPos > str.length()`. So this patch by ARJA-e does not break the correct program behavior and can make the patched program successfully handle some of the inputs that trigger the bug. Note that it is almost impossible to correctly fix L27 by only relying on the associated test suite, because there are no test inputs exposing the fault at line 5 in this test suite.

RQ1.4: Are the incorrect patches generated by ARJA-e still meaningful?
Answer to RQ1.4: Unlike existing approaches (e.g., GenProg, RSRepair and AE), ARJA-e rarely generates patches that are equivalent to a single functionality deletion. Furthermore, at least for 6 bugs, the incorrect patches by ARJA-e are identified as valid patches, and at least for 3 bugs, the incorrect patches by ARJA-e can completely fix the bug while introducing regressions.

```
// NumberUtils.java
public static Number createNumber(String str) throws NumberFormatException { ...
    if (decPos > -1) {
          if (expPos > -1) {
             if (expPos < decPos) { ... } ...
          } ...
     } else {
          if (expPos > -1) {
+            if (!(expPos <= str.length())) throw new NumberFormatException();
             mant = str.substring(0, expPos);
          } ...
     } ...
}
```

Fig. 19. Valid patch generated by ARJA-e for bug L27.

6.1.5 False Positives. ARJA-e can fix 106 bugs in terms of passing the test suite, but the majority of these bugs are not identified as being correctly fixed, leading to a relatively high false positive rate. Considering that all the existing repair systems can only correctly fix a very limited number of bugs, it is hard to predict wether the false positive rate obtained by ARJA-e or other repair systems is acceptable or not for a human developer in practice. At the current state of program repair research, we think it may be more pressing to increase the number of bugs that can be fixed correctly, although a low false positive rate is very desirable if we have succeeded in generating enough correct repairs. Moreover, if an existing repair system is currently put into industrial practice, it largely acts as a recommender system since human oversight is still essential [60]. According to our experience, overfitting patches recommended by ARJA-e are still very helpful for a developer to better understand the cause of a bug or strengthen the existing test suite, though this benefit needs to be further verified by extensive user studies. Even human developers can make mistakes. An empirical study by Smith et al. [85] has indicated that novice developers are likely to overfit a test suite and do not perform better than repair systems. Lastly, it is worth pointing out that a repair system will have a high risk of overfitting the benchmark problems on which it is tested, if the false positive rate is not reasonably reduced. Take ARJA-e for example: We find that template BEAR only contributes to overfitting repairs. So by ignoring this template, we can significantly reduce the false positive rate obtained by ARJA-e in our experiments. However, this may not be a good practice and lead to other problems, because BEAR has been recognized as a common fix pattern. Simply ignoring it may hurt the performance of ARJA-e on other datasets.

6.2 On the Search Space

6.2.1 Contribution of Sources of Fix Ingredients. Table 10 shows the contribution of the statement-level redundancy assumption (including two types of repair actions) and repair templates (including 7 types of repair actions) to the number of bugs fixed (i.e., plausible) and correctly fixed by ARJA-e for patches ranked first. As can be seen from Table 10, both sources make substantial contributions. SR and SI contribute to the plausible fixing of 17 and 31 bugs, respectively, while for correct fixing, they contribute to 7 and 8 bugs, respectively. ER has the most contribution among repair templates, contributing to the plausible fixing of 48 bugs and to the correct fixing of 19 bugs. In total, the statement-level redundancy assumption contributes to plausible fixing of 48 bugs and to correct fixing of 15 bugs, versus 72 and 29 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, these numbers reflect the overall contribution of the two source of fix ingredients to ARJA-e's repair results.

Table 10. Contribution of Statement-Level Redundancy Assumption and Repair Templates to first ranked patches

Statement-Level Redundancy Assumption	Plausible	Correct
Statement Replacement (SR)	17	7
Statement Insertion (SI)	31	8
Repair Template	**Plausible**	**Correct**
Null Pointer Checker (NPC)	7	5
Range Checker (RC)	2	1
Cast Checker (CC)	1	1
Divide-By-Zero Checker (DC)	1	1
Method Parameter Adjuster (MPA)	3	2
Boolean Expression Adder or Remover (BEAR)	10	0
Element Replacer (ER)	48	19

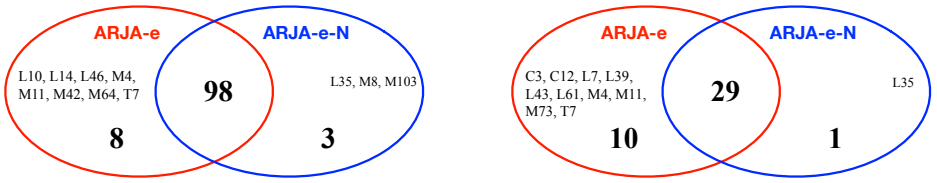
Note that template BEAR contributes to 10 plausible fixes, but 0 correct fixes. We still recommend using this template in ARJA-e, since BEAR is a common repair pattern according to several empirical studies [13, 62, 76] and may help to generate correct fixes when applying ARJA-e to other datasets. How to synthesize more accurate boolean conditions in BEAR is still an open question and needs to be further investigated.

RQ2.1: To what extent do the statement-level redundancy assumption and repair templates contribute to the overall performance of ARJA-e?
Answer to RQ2.1: Both the statement-level redundancy assumption and repair templates contribute substantially to the overall performance of ARJA-e, as shown in Table 10. So it is beneficial to combine these two sources of fix ingredients to determine a search space.

6.2.2 Effect of Search Space Reduction. We developed an ARJA-e variant, denoted as ARJA-e-N, which differs from ARJA-e only in that it does *not* use the search space reduction strategy based on the replacement similarity and insertion relevance. We evaluated ARJA-e-N on 224 real bugs depicted in Table 4 using the same experimental settings as ARJA-e. Fig. 20 summarizes the overlaps of the repair results between ARJA-e and ARJA-e-N in a Venn diagram. In terms of test-adequate bug fixing, ARJA-e-N shows comparable performance to ARJA-e. The reason is that although ARJA-e-N works over a larger search space than ARJA-e, its search space also contains more overfitting patches, so that the chance of finding a plausible patch may not necessarily be decreased. Only for 3 bugs (i.e., L35, M8 and M103), ARJA-e-N can find plausible patches that ARJA-e cannot find. We examine these patches and find that all of them involve edits that do not follow the similarity or relevance set in ARJA-e (i.e., they are really outside ARJA-e's search space).

The search space reduction strategy helps substantially to fix bugs correctly. From Fig. 20(b) we can see that ARJA-e can correctly fix 10 bugs that cannot be fixed by ARJA-e-N. The 10 bugs can be categorized into three classes:

(1) For M4, M11 and T7, ARJA-e-N even fails to pass the test suite. This implies that in these cases, the reduced search space can ease the search of plausible patches.
(2) For L39, L61 and M73, ARJA-e-N can find plausible patches, but none of them are correct. For example, Fig. 21 shows an incorrect patch generated by ARJA-e-N for M73. This patch



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

Fig. 20. Venn diagram of repaired bugs by ARJA-e and ARJA-e-N.

is ranked first because the LBS at line 5 is assigned a suspiciousness of 1.0. The correct patch should insert a statement before line 5 to check whether `yMin` and `yMax` have different signs, and it locates in ARJA-e-N's search space. However, the incorrect patch that affects the same LBS (shown in Fig. 21) inhibits the evolutionary search from finding this correct patch. Thanks to search space reduction, the incorrect patch shown in in Fig. 21 does not exist in ARJA-e's search space according to replacement similarity.

```
1 // BrentSolver.java
2 public double solve(final UnivariateRealFunction f, final double min,
3         final double max, final double initial) throws ... {
4 ...
5 -     return solve(f,min, yMin, max, yMax, initial, yInitial);
6 +     throw MathRuntimeException.createIllegalArgumentException(...);
7 }
```

Fig. 21. An incorrect patch generated by ARJA-e-N for bug M73.

(3) For C3, C12, L7 and L43, ARJA-e-N can also find correct patches, but none of them are ranked first. For example, Fig. 22 shows an incorrect patch generated by ARJA-e-N for C12. The LBS at line 4 has a suspiciousness of 0.58, whereas all the LBSs manipulated by other returned patches (including the correct patch) have a smaller suspiciousness of 0.5. So the incorrect patch shown in Fig. 22 is ranked first among the patches by ARJA-e-N. For ARJA-e, such an incorrect patch will not be generated because of the low replacement similarity, so that the correct patch stands out.

```
1 // AbstractDataset.java
2 public boolean hasListener(EventListener listener) {
3       List list = Arrays.asList(this.listenerList.getListenerList());
4 -     return list.contains(listener);
5 +     return true;
6 }
```

Fig. 22. An incorrect patch generated by ARJA-e-N for bug C12.

We note that ARJA-e finds 796 plausible patches in total for 106 bugs, while ARJA-e-N finds 1041 plausible patches in total for 101 bugs. According to Fig. 20(b), we know that ARJA-e-N does not generate more correct patches than ARJA-e. So we can safely infer that the search space reduction strategy can significantly reduce the false positive rate.

RQ2.2: Is it beneficial to use the search space reduction strategy based on the replacement similarity and insertion relevance?
Answer to RQ2.2: This strategy can help a lot to correctly fix more bugs. Moreover, it can significantly reduce the false positive rate in program repair.

6.3 On the Search Algorithm

In this subsection, we will evaluate ARJA-e and its variants on 22 multi-location bugs listed in Table 9. The reason why we chose these bugs as the subject has been explained in Section 5.3. To ensure a fair comparison, all tested approaches use the same population size and termination criterion. For each bug, a tested approach performs 50 independent trials and the following two metrics are recorded:

(1) "Success": the number of trials that produce at least one plausible patch among 50 independent trials. This is used as the main metric for comparison.
(2) "#Evaluations": the average number of fitness evaluations required to find the first plausible patch in a successful trial.

To test the difference for statistical significance, we conduct 1-sided t-tests at a 5% significance level on the "Success" results obtained by two competing approaches. Note that patch correctness is not considered in this subsection, because in our repair system the search algorithm is only responsible for finding plausible patches in the search space and the selection of correct patches is left to a post-processing procedure.

To verify the effectiveness of the finer-grained fitness function, we implemented an ARJA-e variant denoted as ARJA-e_1, which just replaces the fitness function (see Eq. (7)) in ARJA-e with that (see Eq. (2)) in ARJA. The underlying GP in GenProg [45] is based on a canonical patch representation (see Fig. 1(a)) that takes each edit as a whole, so this GP can also be used over ARJA-e's search space. To show the superiority of our GP, we develop another ARJA-e variant denoted by ARJA-e_2, which employs GenProg's GP (including patch representation, genetic operators and selection) to traverse ARJA-e's search space. To avoid the influence of different fitness functions, ARJA-e_2 uses the finer-grained fitness function defined in Eq. (7).

Table 11 compares ARJA-e with ARJA-e_1 and ARJA-e_2, where $|T_{neg}|$ is the number of negative tests that trigger the bug. Compared to ARJA-e_1, ARJA-e performs significantly better on 12 bugs and is significantly worse only on a single bug (i.e., M4) in terms of "Success" metric. For bugs L1 and L14, ARJA-e_1 even fails in all 50 trials, whereas ARJA-e still achieves a good success rate. Since ARJA-e differs from ARJA-e_1 only in the fitness function, it can be concluded that the finer-grained fitness function provides better guidance for the evolutionary search to find a repair. In the future, it may be worthwhile to carefully analyze the fitness landscapes resulting from different fitness functions, which could provide deeper insights into the improvement of the search ability.

According to "Success" results, ARJA-e also shows obvious advantage over ARJA-e_2. More specifically, ARJA-e significantly outperforms ARJA-e_2 on 16 bugs and performs significantly worse than ARJA-e_2 only on bug C14. We also find that the performance difference between ARJA-e and ARJA-e_2 is usually very large. For example, for 5 bugs (i.e., C18, L41, L55, M24 and T15), ARJA-e achieves at least 20 more successful trials. Given that ARJA-e_2 uses GenProg's GP to explore the same search space as ARJA-e, we can conclude that our GP shows much stronger search ability than the underlying GP algorithm in GenProg.

Table 11. Comparison Between ARJA-e, ARJA-e_1 and ARJA-e_2

Bug Index	$\|T_{neg}\|$	Success			#Evaluations		
		ARJA-e	ARJA-e_1	ARJA-e_2	ARJA-e	ARJA-e_1	ARJA-e_2
C14	4	13	8	43*	1630.46	1584.00	1058.98
C18	4	46	25†	13†	985.00	981.80	766.77
L1	1	20	0†	22	1240.40	–	940.36
L14	1	18	0†	2†	808.06	–	1442.00
L15	2	7	3	1†	1383.00	1217.00	1940.00
L20	2	45	27†	37†	1053.98	1165.41	1063.86
L34	27	23	9†	0†	1293.61	1433.67	–
L41	2	36	9†	7†	1043.50	1219.22	1733.29
L46	1	21	22	11†	1066.29	747.50	673.36
L50	2	50	50	50	128.72	135.02	121.10
L55	1	50	50	28†	300.04	348.04	283.93
L61	2	39	45	23†	963.31	915.16	1120.61
M4	2	2	15*	2	1241.50	689.00	1197.00
M7	1	30	22	21†	582.27	563.18	762.19
M22	2	50	50	50	62.36	55.60	64.36
M24	1	36	24†	10†	1258.06	1123.21	870.50
M62	1	40	14†	5†	848.70	814.79	631.00
M64	2	23	14†	15	796.91	1024.21	1150.87
M67	1	23	13†	5†	1074.91	955.77	1042.00
M77	2	48	30†	17†	854.27	977.40	874.00
M98	2	49	42†	31†	778.49	1092.93	959.26
T15	1	23	19	3†	1125.09	905.11	783.67

"–" means the data is not available since there is no successful trial.
"†" means the result is significantly worse than that of ARJA-e.
"∗' means the result is significantly better than that of ARJA-e.

Note that in terms of "#Evaluations", ARJA-e does not show superiority over ARJA-e_1 and ARJA-e_2. It requires less evaluations on some bugs but more on others. However, this metric is secondary to "Success" and it can be meaningful compared only when the two competing approaches achieve comparable "Success" results.

RQ3.1: Can the finer-grained fitness function provide better guidance for search?
Answer to RQ3.1: Overall, ARJA-e clearly outperforms ARJA-e_1 in terms of "Success", indicating that the finer-grained fitness function can really provide better guidance of search.

RQ3.2: Is our GP in ARJA-e more effective than the underlying GP in GenProg?
Answer to RQ3.2: ARJA-e generally performs much better than ARJA-e_2 in terms of "Success", indicating that our GP is more effective than the underlying GP algorithm in GenProg.

In ARJA-e, we use a new lower-granularity patch representation (see Fig. 11). This representation differs from ARJA's (see Fig. 2(a)) in that it decouples the statements for replacement and the statements for insertion. Note that we cannot compare the two different representations over ARJA-e's search space. This is because some ingredient statements used by ARJA-e come from repair templates, and each of these statements can be used either for replacement or for insertion and cannot serve both of these purposes, thereby rendering ARJA's representation not applicable to ARJA-e's search space. To show the benefits of the new representation, we implemented two other ARJA-e variants. One variant is denoted by ARJA-e_3, which just switches ARJA-e's search space to ARJA's (i.e., only based on statement-level redundancy assumption) and uses the same search algorithm with ARJA-e. The other variant is denoted by ARJA-e_4, which differs from ARJA-e_3 only

Table 12. Comparison Between ARJA-e_3 and ARJA-e_4

Bug Index	$\|T_f\|$	Success		#Evaluations	
		ARJA-e_3	ARJA-e_4	ARJA-e_3	ARJA-e_4
C14	4	50	47†	212.98	1423.00
C18	4	46	49	778.24	1125.41
L1	1	3	0†	1216.33	–
L14	1	0	0	–	–
L15	2	9	0†	1144.44	–
L20	2	50	50	309.56	962.46
L34	27	0	0	–	–
L41	2	40	16†	817.65	1196.13
L46	1	16	22	1178.31	1342.00
L50	2	50	50	123.56	949.48
L55	1	50	50	279.40	979.78
L61	2	41	40	902.29	1227.38
M4	2	0	0	–	–
M7	1	0	0	–	–
M22	2	50	50	164.34	1042.36
M24	1	23	11†	1318.52	1195.00
M62	1	0	0	–	–
M64	2	18	16	1023.50	1256.50
M67	1	42	0†	1037.81	–
M77	2	0	0	–	–
M98	2	50	50	335.48	1059.38
T15	1	48	41†	486.88	1043.17

"–" means the data is not available since there is no successful trial.
"†" means the result is significantly worse than that of ARJA-e_3 .
"$*$' means the result is significantly better than that of ARJA-e_3 .

in that it uses ARJA's representation rather than ARJA-e's. In ARJA-e_3 and ARJA-e_4, we use the same genetic operators.

Table 12 compares ARJA-e_3 with ARJA-e_4. As can be seen, ARJA-e_3 significantly outperforms ARJA-e_4 on 7 bugs and there are no bugs where ARJA-e_3 is significantly worse than ARJA-e_4. In particular, for bug M67, ARJA-e_4 fails in all trials whereas ARJA-e_3 achieves a very high success rate (i.e., 84%). Note that for 6 bugs (i.e., L14, L34, M4, M7, M62 and M77), both ARJA-e_3 and ARJA-e_4 achieve no successful trials. This is possibly because that the plausible patches for these bugs do not exist in ARJA's search space. Moreover, it is interesting to note that ARJA-e_3 generally needs much less evaluations to find a repair than ARJA-e_4. For example, both ARJA-e_3 and ARJA-e_4 achieve 100% success rate on bugs L20, L50, L55, M22 and M98, but ARJA-e_4 requires about 3–8 times the number of evaluations. The possible reason is that the new representation can make the promising replacement and insertion statements propagated more quickly between solutions. Since the difference between ARJA-e_3 and ARJA-e_4 only lies in the patch representation, we can conclude that the new patch representation used in ARJA-e can facilitate more effective and efficient search of plausible patches.

RQ3.3: What are the benefits of decoupling the statements for replacement and for insertion in the lower-granularity patch representation?
Answer to RQ3.3: Such a decoupling in the lower-granularity patch representation can make the evolutionary search more effective and efficient to find a test-adequate repair.

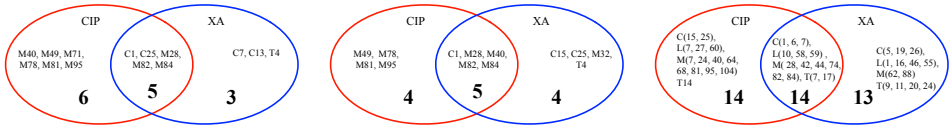
6.4 On Alleviating Patch Overfitting

6.4.1 Performance of the Overfit Detection Approach. We will first evaluate CIP described in Section 3.4.1. To demonstrate its effectiveness, we compare it with Xiong et al.'s approach (XA) [100], which is currently the state-of-the-art technique for overfit detection and shares certain similarities with our approach. To ensure a fair comparison, we use the version without test case generation for XA. According to [100], this simplified version has already achieved competitive performance compared to the version relying on new test cases. We run the two approaches on the dataset of plausible patches depicted in Table 6.

Table 13 shows the comparison results on the dataset per tool. From Table 13 we can see that for the patches of ARJA-e and jGenProg, CIP can filter out more incorrect patches than XA, while for the patches of jKali, the two approaches can identify the same number of incorrect patches. Moreover, CIP does not filter out any correct patch obtained by jGenProg and jKali, while XA filters out one correct patch (for bug M53) by jGenProg. Note that it was reported in [100] that XA does not exclude any correct patch by jGenProg. This inconsistency may be due to different computing environments. For the patches of ARJA-e, both approaches exclude correct patches by mistake, but CIP only excludes 3 out of 19 correct patches whereas XA excludes 7.

Table 13. Comparison Between CIP and XA (The Patches are Categorized by Repair Tools)

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



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

Fig. 23. Intersection of incorrect patches identified by CIP and XA.

To further understand the performance difference between CIP and XA, Fig. 23 shows the intersection of incorrect patches identified by the two approaches. It is interesting to see that CIP complements XA very well. Specifically, CIP can identify 6 incorrect patches by jGenProg, 4 incorrect patches by jKali and 14 incorrect patches by ARJA-e, respectively, which cannot be identified by XA. In addition, we note that none of the 8 correct patches excluded by XA is also excluded by CIP. Given this strong complementarity, it is very promising to further try to improve the accuracy of overfit detection by properly combining the strengths of the two approaches.

In the following we conduct a case study to illustrate why CIP can succeed in many cases where XA cannot. In XA, patch correctness is determined by two distance metrics (i.e., A_p and A_f) and a threshold (i.e., $K_p = 0.25$), where A_p is the maximum distance between the executions of positive test cases on the buggy and patched programs, and A_f is the average distance between the

executions of negative test cases on the buggy and patched programs. A plausible patch is identified as being incorrect by XA if $A_p \geq K_p$ or $A_p \geq A_f$, otherwise it is classified as a correct patch. Fig. 24 shows an incorrect patch generated by jGenProg for M95. XA fails to identify this incorrect patch because after applying the patch, the execution paths of positive test cases do not change (i.e., $A_p = 0$) while the execution paths of negative test cases do change (i.e., $A_f > 0$), thereby leading to $A_p < K_p$ and $A_p < A_f$. In contrast to XA, CIP can successfully identify this incorrect patch. The reason is as follows: when running positive test cases on the buggy program, we can collect 11 input-output pairs for the method `getInitialDomain`; however, when we invoke `getInitialDomain` in the patched program with the 11 inputs, we obtain outputs that are totally different from those collected previously.

```
// FDistributionImpl.java
protected double getInitialDomain(double p) {
      double ret;
      double d = getDenominatorDegreesOfFreedom();
      ret = d / (d - 2.0);
-     return ret;
+     return numeratorDegreesOfFreedom;
}
```

Fig. 24. Incorrect patch generated by jGenProg for bug M95.

```
// MannWhitneyUTest.java
private double calculateAsymptoticPValue(final double Umin, final int n1,
   final int n2) throws ConvergenceException, MaxCountExceededException {
-    final int n1n2prod = n1 * n2;
+    final double n1n2prod = n1 * n2;
     final double EU = n1n2prod / 2.0;
   ...
}
```

Fig. 25. Correct patch generated by ARJA-e for bug M30.

As shown in Table 13, it is also possible for XA to exclude correct patches. One major reason is that when a correct patch contains new algorithmic procedures (e.g., the patch by ARJA-e for T15) or method invocations (e.g., the patch by ARJA-e for C12), the control flow can be significantly changed, even for positive test cases. Improper setting of threshold K_p can also lead to the failure of XA, whereas CIP is free of parameters. Here we provide an example to illustrate another reason for the failure of XA, which concerns the boundary condition. Fig. 25 shows the correct patch generated by ARJA-e for bug M30. We find that this patch will not change the execution path of any test case, implying $A_p = 0$ and $A_f = 0$. So condition $A_p \geq A_f$ and XA will misclassify this patch as an incorrect patch. Note that if the classification condition $A_p \geq A_f$ is changed to $A_p > A_f$, the failure of XA for this patch can be avoided. However, such a change may influence the successful identification of some other patches. Unlike XA, CIP will not exclude the correct patch shown in Fig. 25, because there is no difference between any method outputs Out_i and Out'_i (see Section 3.4.1).

Moreover, it is worth mentioning that the overfit detection results of CIP can be easily used by a human developer to augment the original test suite. For example, in Fig. 24, the input of `getInitialDomain` involves parameter `p` and field variable `denominatorDegreesOfFreedom`, while the output involves a return value with type `double`. By running our approach, one of the input-output pairs collected over the buggy program is: `p` is 0.1, `denominatorDegreesOfFreedom` is 6, and return value is 1.5; and the input therein will lead to a different output on the patched program. If correctness of the input-output pair can be confirmed by the human developer, a test case like that shown in Fig. 26 can be immediately added into the original test suite, which can eliminate overfitting patches like that in Fig. 24. Unlike CIP, it may be impossible to use the overfit detection

```
@Test
public void testInitialDomain() {
   FDistributionImpl fd = new FDistributionImpl(0, 6);
   //The second parameter specifies that denominatorDegreesOfFreedom is 6
   Assert.assertEquals(1.5, fd.getInitialDomain(0.1));
}
```

Fig. 26. A test case derived from the overfit detection result of our approach.

results of XA for test suite augmentation, since the changes of execution paths are abstract and are thereby difficult to be comprehended by a human developer.

Lastly, we find that CIP spends about 15 seconds on average to determine the correctness of a patch whereas XA spends about 19 minutes. So XA is much more computationally expensive than our approach. The possible reason is that XA needs to instrument a large number of program points in order to record the runtime trace, whereas CIP only needs to instrument the entry and the exit points for each modified method.

RQ4.1: What percentage of incorrect patches can be identified by CIP?
Answer to RQ4.1: CIP can identify 49.48% incorrect patches that are generated by three repair tools, versus 45.36% by XA.

RQ4.2: What are the advantages of CIP compared to the heuristic approach proposed by Xiong et al. [100]?
Answer to RQ4.2: Compared to XA, the advantages of CIP are summarized as follows: (i) CIP can filter out comparable number of incorrect patches with excluding less number of correct patches; (ii) CIP complements XA very well; (iii) the overfit detection results of CIP can assist a human developer in augmenting the original test suite, whereas the results of XA generally cannot; (iv) CIP requires much less computation time; (v) CIP is free of parameters, whereas XA relies on a threshold K_p.

6.4.2 Contribution of the Metrics to Patch Ranking. As described in Section 3.4.2, we hierarchically use the metrics *Susp*, *Dist* and $Pref$ to rank plausible patches. Here we would like to investigate whether each of the three metrics contributes to the final ranking performance. To do this we compare the original ranking scheme (denoted by RS_0) with six alternatives (denoted by RS_1–RS_6). RS_1, RS_2 and RS_3 just use a single metric (i.e., *Susp*, *Dist* and $Pref$, respectively), while RS_4, RS_5 and RS_6 disable the using of $Pref$, *Dist* and *Susp* in RS_0, respectively. Table 14 shows the ranking performance of all these ranking schemes. It can be seen that the original ranking scheme RS_0 outperforms all the others in terms of the number of correct patches ranked first, indicating that all

the three metrics have an effect on the final ranking. *Susp* contributes most, while *Dist* and *Pref* make almost the same contribution.

Table 14. Performance of Ranking Schemes (*Susp*, *Dist* and *Pref* are abbreviated to *S*, *D* and *P*, respectively)

Ranking Scheme	RS_0 (*S*, *D*, *P*)	RS_1 (*S*)	RS_2 (*D*)	RS_3 (*P*)	RS_4 (*S*, *D*)	RS_5 (*S*, *P*)	RS_6 (*D*, *P*)
Number of Correct Patches Ranked First	39	32	29	29	35	35	31

Considering *Dist* is a special metric that is derived from our overfit detection technique, we will use an example to illustrate how it works in patch ranking. Fig. 27 shows two patches generated by ARJA-e for T7. The incorrect patch replaces the method invocation `year()` with `weekyear()`, while the correct patch changes the variable `instantLocal` to `instantMillis`. Because the two patches modify the same statement with template ER, they have the same *Susp* and *Pref* values. And because the incorrect patch is found earlier in our search, it will be ranked higher than the correct patch if we do not consider *Dist*. But once *Dist* is used for comparison, we can distinguish the two patches because *Dist* is 0.0125 for the incorrect patch and is 0 for the correct one.

```
1 // DateTimeFormatter.java
2 public int parseInto(ReadWritableInstant instant, String text, int position) { ...
3 -    int defaultYear = chrono.year().get(instantLocal);
4 +    int defaultYear = chrono.weekyear().get(instantLocal);    // incorrect patch
5 +    int defaultYear = chrono.year().get(instantMillis);       // correct patch
6 ...
7 }
```

Fig. 27. Incorrect and correct patches generated by ARJA-e for T7.

RQ4.3: Do all three metrics used in patch ranking contribute to the final ranking performance?
Answer to RQ4.3: As shown in Table 14, all three metrics make a meaningful contribution to the final ranking performance.

7 THREATS TO VALIDITY

In this section, we discuss the threats to the internal, external and construct validity of our empirical study.

Internal Validity. ARJA-e needs to be configured with a number of parameters as shown in Table 5. Different parameter values may affect the internal validity of this work. In our experiments, we basically followed the parameter setting in ARJA [108] and did not fine-tune these parameters. The results shown in Table 7 were obtained from five random trials of ARJA-e on each of the 224 bugs considered. These results might slightly change in replication experiments, influenced by the random nature of ARJA-e. In addition, the test filtering used in our repair approach may not be theoretically rigorous in view of the fact that Java is a high-level object-oriented language. To address this threat, we post-validated every plausible patch found by ARJA-e on the original test suite. We found that all patches also fulfill the original test suite, with no exception.

External Validity. We performed empirical evaluation on 224 bugs from Defects4J. These bugs may not fully represent the natural distribution of real-world bugs. So our results may not generalize

to other bugs. However, we note that Defects4J is the most widely used dataset for evaluating Java repair systems. In the future, it would be worthwhile to evaluate ARJA-e on recently proposed datasets [51, 59, 81] of Java bugs in order to further examine its repair performance.

Construct Validity. Although manually checking the correctness of patches has been a common practice in automatic program repair [15, 31, 61, 82, 96, 99, 108], it is indeed not scientifically sound. We may identify an incorrect patch as correct due to limited domain knowledge. To mitigate this threat, we put great effort into understanding program functionality and took a conservative approach to the manual study.

8 RELATED WORK

In this section, we first review evolutionary and non-evolutionary approaches for program repair. Then, we provide an overview of the techniques on patch overfitting. Lastly, we summarize briefly a number of notable studies on the empirical aspects of program repair. Our review devotes most attention to work closely related to our proposed technique. For a more comprehensive review, readers may want to consult two recent survey papers on program repair [28, 70].

8.1 Evolutionary Program Repair

Evolutionary repair approaches formulate program repair as a search problem and use evolutionary algorithms, GP in particular, to search for test-adequate repairs.

In 2008, Arcuri and Yao [6] presented the idea of using GP to co-evolve programs and test cases in order to automate the fixing of bugs, which was experimentally demonstrated on an instance of the bubble-sort algorithm. A similar idea using co-evolution was further investigated in [97], where a multi-objective approach was adopted associating each objective with an individual specification for the program. Such co-evolutionary approaches require a formal program specification to calculate fitness values, limiting their generality and scalability. GenProg [26, 48, 94] was the first evolutionary repair system that could be applied to real-world software. GenProg's scalability mainly comes from two innovations: (i) GenProg evaluates an individual's fitness based on a given test suite without the need of a formal specification; (ii) GenProg applies genetic operators only to the possibly faulty statements found by a fault localization technique, thereby reducing the search space significantly. Le Goues et al. [45] suggested a new version of GenProg, where the key insight is to represent candidate repairs as patches rather than as ASTs of the modified programs, given that the memory consumption of a population of ASTs for large programs is usually unaffordable. This patch representation can make GenProg scalable to millions of lines of code. Note that Ackling et al. [2] also introduced a kind of patch representation that encodes a patch as a variable length bit-string.

Since the results achieved by GenProg were very promising [26, 45, 48, 94], follow-up studies on evolutionary program repair were mainly based on GenProg. To fix bugs in embedded systems, Schulte et al. [83, 84] extended previous work on GenProg to search for repairs at the assembly code level rather than the source code level. Fast et al. [25] proposed to use only a subset of positive tests, selected by sampling when evaluating the fitness, in order to improve the efficiency of the fitness function. In addition, they also tried to exploit the behavior measured by program invariants to improve the precision of the fitness function, but such a model was found to be very opaque and impossible to generalize to different bugs [46]. Le Goues et al. [49] investigated choices for the solution representations and genetic operators in GenProg and provided several concrete improvements according to experimental results. Since GenPog often generates nonsensical patches, Kim et al. [37] proposed PAR, which uses repair templates to generate program variants instead of random mutations, in order to produce patches more acceptable than those of GenProg. Tan et al. [89] suggested a set of generic forbidden transformations, called anti-patterns, which could be

enforced on top of GenProg or other evolutionary repair approaches. Most of these anti-patterns are related to the deletion operation which has been found to be involved in overfitting patches. Le et al. [43] presented a repair approach that uses the mined bug fix history to guide evolutionary search, where a candidate patch more similar to historical fixes will be given higher priority in the search process. Oliveira et al. [74] proposed a lower-granularity patch representation that decouples three subspaces (i.e., operator, fault and fix spaces) of the repair problem formulated in GenProg, and designed several new crossover operators specifically for this representation so as to explicitly traverse three subspaces. Their experiments showed that the resulting algorithm can provide considerable improvement over GenProg. However, the proposed lower-granularity representation can produce invalid genes during crossover, thereby leading to the loss of good partial information. In addition, the crossover in this representation exchanges information between different buggy locations very frequently, which may not be desirable. Yuan and Banzhaf [108] presented ARJA, a repair system for automated repair of Java programs. In ARJA, a different lower-granularity patch representation was described, inspired by the limitations of Oliveira et al.'s representation. Based on this representation, program repair is formulated as a multi-objective search problem, where the patch size is considered explicitly. Several auxiliary techniques, such as rule-based search space reduction and test filtering, were also included in ARJA. Souza et al. [20] presented a new fitness function, based on tracking of the values of numeric local variables participating in control-flow statements.

Discussion of Differences. In addition to GenProg's statement-level operation based on the redundancy assumption, ARJA-e incorporates repair templates generalized from those in PAR and recent non-evolutionary repair approaches [53, 55, 80, 82, 99], so as to conduct more targeted (e.g., null pointer check) or finer-grained (e.g., method name replacement) modifications. Consequently, compared to existing evolutionary repair approaches, ARJA-e considers a much larger search space that is more likely to contain correct patches. Unlike GenProg and ARJA, however, which reduce the search space only according to constraints enforced by the compiler, ARJA-e conducts a light-weight contextual analysis to filter out nonsensical replacement and insertion statements. Different from PAR which applies repair templates on-the-fly, ARJA-e executes all possible templates prior to search and converts the template-based edits into two kinds of statement-level edits. In terms of patch representation, ARJA-e uses a lower-granularity representation that is characterized by a decoupling of statements for replacement and for insertion. This representation is different from GenProg's coarse-grained representation which takes an edit as a whole, and is also different from ARJA's representation which mixes replacement and insertion statements in a single evolvable structure. Furthermore, compared to GenProg, PAR and ARJA, ARJA-e uses a finer-grained fitness function to better guide the evolutionary search. The fitness function in ARJA-e uses only reliable information encoded in the test suite, a further difference from those approaches that rely on information beyond the test suite, such as historical bug fixes [43] or extracted program state information [20, 25].

8.2 Non-Evolutionary Program Repair

Non-evolutionary repair approaches can be roughly divided into two classes: generate-and-validate approaches and semantics-based approaches. Generate-and-validate approaches are conceptually similar to evolutionary repair approaches, with the main difference that they use other search algorithms (e.g., random search or enumerative search) rather than evolutionary search to navigate their search space. Semantics-based approaches encode test cases as a number of constraints and synthesize a repair by solving the resulting constraint satisfaction problem.

8.2.1 Generate-and-Validate Approaches. Weimer et al. [93] proposed a deterministic repair approach called AE, which tries to avoid validating semantically equivalent program variants by using an approximate program equivalence relation and employs adaptive search strategies to control the order of examination of patches and test cases. Qi et al. [77] presented RSRepair that replaces GP in GenProg with random search. Long and Rinard [54, 55, 57] presented SPR, Prophet and Genesis. SPR [55] works with a set of parameterized transformation schemata and uses target value search and a condition synthesis algorithm to instantiate these transformation schemata. Prophet [57] builds a probabilistic model to predict the correctness of patches in SPR's search space, and validates candidate patches in order of predicted correctness. Genesis [54] aims to automatically infer program transformations for repair from human-written patches. Xiong et al. presented ACS [101], which can perform accurate condition synthesis aided by the analysis of Javadoc comments. Gupta et al. [29] developed a tool for common C errors, called DeepFix, which uses a deep neural network to predict possibly faulty locations along with fix statements. Chen et al. [15] presented JAID, which constructs detailed state abstractions by dynamic program analysis and then generates potential fix code to avoid reaching suspicious states. Saha et al. [82] proposed ELIXIR, which constructs richer repair expressions by considering the insertion of method invocations and builds a logistic regression model with four context related features to rank the candidate patches. Xin and Reiss [99] presented ssFix, which searches a code database to find the candidate fix ingredients that are syntax-related to the suspicious statement. Jiang et al. [33] presented SimFix. This approach uses similar code snippets from the current buggy program to fix a likely-buggy location, and only validates patches that conduct frequent AST modifications. Wen et al. [96] proposed CAPGEN, which works on AST node levels and prioritizes the candidate patches based on the AST nodes' context information. Liu and Zhong [53] presented SOFIX, which exploits repair templates mined from Stack Overflow. Hua et al. [31] proposed SKETCHFIX. This approach translates a buggy program to a partial program with holes using a number of AST node-level transformations and then employs backtracking search to lazily instantiate these holes during test validation, substantially pruning the search space thereby. Mechtaev et al. [67] proposed to partition candidate patches into test-equivalence classes on the fly, in order to significantly reduce the number of test executions.

Discussion of Differences. To traverse the search space of patches, RSRepair uses random search, while the other approaches in this category basically resort to the enumerative search. So in terms of the search algorithm, the novel multi-objective evolutionary search guided by finer-grained fitness function distinguishes ARJA-e from all of these repair approaches. Similar to our repair approach, SPR, ELIXIR, ssFix, SimFix, CAPGEN and SOFIX can also conduct a set of predefined changes finer-grained than statement level. However, one major limitation of these approaches is that they can only target a single likely-buggy location, so it is impossible for them to generate multi-location repairs. In contrast, ARJA-e can manipulate multiple likely-buggy locations simultaneously through GP, so there is high potential for ARJA-e to fix multi-location bugs. Similar to our approach, ELIXIR, ssFix, SimFix and CAPGEN perform contextual analysis to select potential fix ingredients. In this respect, ARJA-e mainly differs in that it distinguishes the replacement and insertion operations and selects promising replacement and insertion statements according to replacement similarity and insertion relevance respectively.

8.2.2 Semantics-Based Approaches. Nguyen et al. [73] proposed SemFix, a pioneering work on semantics-based approaches. SemFix first uses symbolic execution to generate repair constraints from the given test cases and then feeds these constraints to a program synthesis module to generate a repair. The other repair approaches that fall into this category include SearchRepair [36], DirectFix [68], Angelix [69], JFix [40], QLOSE [19], Nopol [103] and S3 [41].

Discussion of Differences. Similar to our repair approach, DirectFix also aims to find simple patches. However, DirectFix takes into account patch simplicity by integrating fault localization and patch generation into one step, whereas ARJA-e considers this by explicitly treating patch size as an optimization objective. Angelix and S3 can handle multi-location bugs by exploiting the light-weight repair constraint called angelic forest, whereas ARJA-e achieves this by leveraging the expressive power of GP. Nopol only aims at conditional bugs, whereas ARJA-e is more general.

8.3 Techniques Related to Patch Overfitting

For addressing patch overfitting, there are two classes of techniques in the literature: overfit detection and patch ranking.

8.3.1 Overfit Detection Techniques. Because patch overfitting is caused by a weak test suite, a natural idea for overfit detection is to use additional test cases. Xin et al. [98] proposed DiffTGen that identifies overfitting patches. This method first generates additional test inputs that can uncover the semantic differences between the buggy program and the patched program, then reports these differences to an oracle for judging correctness. The empirical evaluation showed that DiffTGen can identify 49.3% overfitting patches generated by four repair approaches. An obvious drawback of DiffTGen is that it requires an oracle that is usually not available in practice. Yang et al. [104] presented an overfitting patch detection framework called Opad. Opad first employs fuzzy testing to generate test inputs and then empowers these test inputs with two inherent oracles (i.e., crash and memory safety). Their evaluation on 45 C bugs showed that Opad can filter out 75.2% of the overfitting patches. However, it is still unclear which types of overfitting can be detected just by inherent oracles, and it has been shown that Opad is indeed not effective in identifying overfitting patches in Java [100]. Yu et al. [107] presented two approaches (called MinImpact and UnsatGuided) based on test case generation for alleviating the overfitting issue. Their approaches generate new test cases using the buggy program as an oracle. The idea of MinImpact is that if a patched program fails in more newly generated test cases, then the corresponding patch is more likely to suffer from overfitting. UnsatGuided is tailored for semantics-based approaches, and exploits a new repair constraint enforced by each generated test case if this constraint does not contradict existing ones. However, their empirical results indicate that the two approaches are not helpful for generating correct patches. Recently, Xiong et al. [100] proposed an approach that can heuristically identify the correctness of plausible patches. Their approach is based on the assumption that in terms of execution paths, positive tests on the buggy and patched programs should behave similarly while negative tests on the buggy and patched programs should behave differently. In addition, they used newly generated test inputs to enhance the original test suites. Their empirical evaluation was conducted on 139 patches obtained by existing repair approaches and the results showed that their approach can identify 56.3% incorrect patches without excluding any correct patches.

Discussion of Differences. Our approach CIP does not rely on any new test cases or inherent oracles, making it quite different from DiffTGen, Opad, MinImpact and UnsatGuided. CIP is somewhat similar to Xiong et al.'s approach (XA). Both of these approaches can work based on the original test suite, and exploit the differences in the execution of positive test cases introduced by the patch. However, the basic assumption of CIP is quite different from that of XA. XA assumes that the execution paths of positive test cases between the buggy and patched programs should not changed a lot, whereas CIP assumes that the buggy program can still function correctly on the test inputs encoded in the positive test cases so that with these inputs, the patched program should obtain the same outputs as the buggy program. In theory, XA may fail when the correct patch introduces complex changes such as the addition of new method invocations, whereas CIP is not sensitive to the complexity of patches.

8.3.2 Patch Ranking Techniques. Patch ranking is usually employed internally in repair approaches, giving priority to patches that are more likely to be correct. Most of these techniques leverage syntactic [15, 19, 31, 33, 41, 43, 55, 96, 99, 101] or semantic differences [19, 41] between the buggy and patched programs to heuristically compare the possibility of being correct. How to quantify such differences varies between different repair approaches and generally depends on their repair models, such as which repair actions are used. Unlike these techniques, Prophet [57] and ELIXIR [82] learn probabilistic models to predict the probability of a patch being correct.

Discussion of Differences. Different from most of patch ranking techniques [15, 31, 33, 43, 55, 96, 99, 101] that only exploit the changes to the program syntax, our technique combines both program syntax and program semantics to rank plausible patches. The semantics-based approaches QLOSE [19] and S3 [41] also exploit both types of information. However, their ranking procedures are used during the repair process in order to prioritize the validation of patches, whereas our ranking procedure is used as a post-processing step to rank the plausible patches found by an evolutionary repair approach. Moreover, in QLOSE and S3, the average or sum of the syntactic and semantic distances is used to score patches, whereas our ranking procedure considers syntactic and semantic differences in a hierarchical way. Most importantly, two new metrics $Dist$ and $Pref$ are introduced in our ranking technique, which can be easily used in other repair systems for the purpose of ranking plausible patches.

8.4 Empirical Aspects of Repair

Another line of research focuses on the empirical aspects of program repair, including the performance evaluation of different repair systems [24, 38, 42, 61, 105], the investigation of patch quality [44, 56, 79, 85, 106], the analysis of real-world bug fixes [13, 52, 62, 86, 87, 111], the validation of the redundancy assumption [9, 65], the study of the influence of fault localization [7, 78, 90, 95], the creation of datasets of bugs [10, 34, 47, 51, 59, 81, 88], as well as several novel aspects that have been examined in recent empirical studies [35, 71, 91].

It is worth noting that other researchers have evaluated our original ARJA system [108] on other benchmark datasets besides Defects4J. Ye et al. [105] empirically evaluated ARJA along with Astor [63, 64], Nopol, NPEfix [23] and RSRepair on a dataset called QuixBugs [51], providing several new findings about program repair. To have a better understanding of the performance of repair tools across various benchmarks, Durieux et al. [24] conducted a large-scale experiment on five benchmark datasets using 11 Java repair tools, where four tools (i.e., ARJA, GenProg, Kali and RSRepair) were implemented in the ARJA framework.

9 CONCLUSION

In this paper, we have proposed an integrated approach, called ARJA-e, for better evolutionary program repair. By combining two sources of fix ingredients (i.e., statement-level redundancy assumption and repair templates), ARJA-e can conduct complex statement-level transformations like GenProg and ARJA, targeted code changes (e.g., adding a null pointer checker), and code changes at a finer-granularity than statement level (e.g., replacing a method name), which empowers ARJA-e to fix various kinds of bugs. To reduce the search space and avoid nonsensical patches, ARJA-e uses a strategy based on a light-weight contextual analysis, which can filter out unpromising replacement and insertion statements, respectively. In order to harness the potential repair power of the search space, ARJA-e first unifies the edits at different granularities into statement-level edits, so as to encode patches in the search space with a lower-granularity patch representation that is characterized by the decoupling of statements for replacement and insertion. With this new patch representation, ARJA-e employs multi-objective GP to navigate the search space. To better guide the search of GP, ARJA-e uses a finer-grained fitness function that can make full use of semantic

information provided by existing test cases. Moreover, ARJA-e includes a post-processing tool for alleviating patch overfitting. This tool can serve two purposes: (i) it can identify incorrect patches by using a new overfit detection approach that is not sensitive to patch complexity; (ii) it can rank the plausible patches using a heuristic patch ranking approach that can be easily generalized to other evolutionary repair systems.

We have conducted an extensive empirical study on 224 real bugs in Defects4J. The evaluation results show that ARJA-e outperforms 7 existing repair approaches by a considerable margin in terms of the number of bugs correctly fixed. Specifically, ARJA-e can correctly fix 39 bugs in terms of the patches ranked first. Moreover, ARJA-e also shows increased strength in multi-location repair compared to its predecessor ARJA. With respect to the search space, the results demonstrate that both the statement-level redundancy assumption and repair templates contribute substantially to the overall performance of ARJA-e, and the search space reduction strategy can help substantially to correctly fix bugs. Regarding the search algorithm, the results show that the finer-grained fitness function can significantly improve the success rate of repair, and the decoupling of replacement and insertion statements in the lower-granularity patch representation can lead to more effective and efficient search. Regarding patch overfitting, the results indicate that our overfit detection technique shows several advantages over a state-of-the-art approach [100], and all three metrics used in patch ranking make a meaningful contribution.

In the future, we plan to incorporate additional sources of fix ingredients (e.g., source code repositories [36, 99]) into our repair framework, which should increase the potential for fixing more bugs. Moreover, we would like to investigate new mating and survival selection methods [30, 72, 110] in GP, so as to further improve the evolutionary search algorithm for bug repair.

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] Thomas Ackling, Bradley Alexander, and Ian Grunert. 2011. Evolving patches for software repair. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. ACM, 1427–1434.

[3] Andrea Arcuri. 2010. It does matter how you normalise the branch distance in search based software testing. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*. IEEE, 205–214.

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

[5] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*. IEEE, 1–10.

[6] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the 2008 IEEE Congress on Evolutionary Computation*. IEEE, 162–168.

[7] Fatmah Yousef Assiri and James M Bieman. 2017. Fault localization for automated program repair: Effectiveness, performance, repair correctness. *Software Quality Journal* 25, 1 (2017), 171–199.

[8] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. 1998. *Genetic programming: An introduction*. Vol. 1. Morgan Kaufmann San Francisco.

[9] 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.

[10] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 117–128.

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

[12] Eric Bruneton. 2011. ASM 4.0 A Java bytecode engineering library. (2011). https://asm.ow2.io/asm4-guide.pdf

[13] Eduardo C Campos and Marcelo A Maia. 2017. Common bug-fix patterns: A large-scale observational study. In *Proceedings of the 11th International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 404–413.

[14] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: An eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM,

378–381.
[15] 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.
[16] JUnit Contributors. 2004. A programmer-oriented testing framework for Java. https://github.com/junit-team/junit4
[17] Kryo Contributors. 2013. Java binary serialization and cloning: Fast, efficient, automatic. https://github.com/EsotericSoftware/kryo
[18] MSU HPCC Contributors. 2019. High Performance Computing at iCER. (2019). https://wiki.hpcc.msu.edu
[19] 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.
[20] Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. 2018. A novel fitness function for automated program repair based on source code checkpoints. In *Proceedings of the 20th Annual Conference on Genetic and Evolutionary Computation*. ACM, 1443–1450.
[21] 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.
[22] 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.
[23] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 349–358.
[24] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://arxiv.org/abs/1905.11973
[25] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2010. Designing better fitness functions for automated program repair. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. ACM, 965–972.
[26] 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.
[27] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.
[28] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.
[29] 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.
[30] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (2015), 630–643.
[31] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 12–23.
[32] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678.
[33] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar Code. In *Proceedings of the 27th International Symposium on Software Testing and Analysis*. ACM, 298–309.
[34] 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.
[35] René Just, Chris Parnin, Ian Drosos, and Michael D Ernst. 2018. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In *Proceedings of the 27th International Symposium on Software Testing and Analysis*. ACM, 287–297.
[36] 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.
[37] 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.
[38] Xianglong Kong, Lingming Zhang, W Eric Wong, and Bixin Li. 2015. Experience report: How do techniques, programs, and tests impact automated program repair?. In *Proceedings of the 26th International Symposium on Software Reliability*

Engineering. IEEE, 194–204.
[39] John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.
[40] 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.
[41] 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.
[42] Xuan-Bach D Le, David Lo, and Claire Le Goues. 2016. Empirical Study on Synthesis Engines for Semantics-Based Program Repair. In *Proceedings of the 2016 International Conference on Software Maintenance and Evolution*. IEEE, 423–427.
[43] 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.
[44] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033.
[45] 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.
[46] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software Quality Journal* 21, 3 (2013), 421–443.
[47] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.
[48] 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.
[49] Claire Le Goues, Westley Weimer, and Stephanie Forrest. 2012. Representations and operators for improving evolutionary software repair. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. ACM, 959–966.
[50] Yanchuan Li and Gordon Fraser. 2011. Bytecode testability transformation. In *Proceedings of the 2011 International Symposium on Search Based Software Engineering*. Springer, 237–251.
[51] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 55–56.
[52] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 275–286.
[53] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *Proceedings of 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 118–129.
[54] 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.
[55] 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.
[56] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 702–713.
[57] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices* 51, 1 (2016), 298–312.
[58] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 227–238.
[59] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An extensible Java bug benchmark for automatic program repair studies. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 468–478.
[60] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated end-to-end repair at scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 269–278.
[61] 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.
[62] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205.
[63] 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.
[64] Matias Martinez and Martin Monperrus. 2019. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *Journal of Systems and Software* 151 (2019), 65–80.
[65] 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.
[66] Phil McMinn. 2004. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.
[67] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-equivalence analysis for automatic patch generation. *ACM Transactions on Software Engineering and Methodology* 27, 4 (2018), 15.
[68] 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.
[69] 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.
[70] Martin Monperrus. 2018. Automatic software repair: A bibliography. *Comput. Surveys* 51, 1 (2018), 17.
[71] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering* 23, 5 (2018), 2901–2947.
[72] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909* (2015).
[73] 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.
[74] 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.
[75] Oracle. 2010. Java object serialization specification: System architecture. https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serial-arch.html#7182
[76] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (2009), 286–315.
[77] 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.
[78] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. 2013. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 191–201.
[79] 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.
[80] 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.
[81] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*. IEEE, 10–13.
[82] 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.
[83] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. 2013. Automated repair of binary and assembly programs for cooperating embedded devices. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 317–328.
[84] Eric Schulte, Stephanie Forrest, and Westley Weimer. 2010. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, 313–316.
[85] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543.

[86] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 130–140.

[87] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 512–515.

[88] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 180–182.

[89] 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.

[90] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. 2017. An investigation into the use of mutation analysis for automated program repair. In *Proceedings of the 2017 International Symposium on Search Based Software Engineering*. Springer, 99–114.

[91] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 2018. How to design a program repair bot?: Insights from the repairnator project. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 95–104.

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

[93] 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.

[94] 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.

[95] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2017. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv preprint arXiv:1707.05172* (2017).

[96] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1–11.

[97] Josh L Wilkerson, Daniel R Tauritz, and James M Bridges. 2012. Multi-objective coevolutionary automated software correction. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. ACM, 1229–1236.

[98] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th International Symposium on Software Testing and Analysis*. ACM, 226–236.

[99] 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.

[100] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 789–799.

[101] 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.

[102] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 512–523.

[103] 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.

[104] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 831–841.

[105] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A comprehensive study of automatic program repair on the QuixBugs benchmark. In *Proceedings of the IEEE 1st International Workshop on Intelligent Bug Fixing*. IEEE, 1–10.

[106] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering* 23, 5, 2948–2979.

[107] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2017. Test case generation for program repair: A study of feasibility and effectiveness. *arXiv preprint arXiv:1703.00198* (2017).

[108] 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

[109] Yuan Yuan and Wolfgang Banzhaf. 2019. ARJA-e: A System for Better Evolutionary Program Repair. https://github.com/yyxhdy/arja/tree/arja-e

[110] Yuan Yuan, Hua Xu, Bo Wang, and Xin Yao. 2016. A new dominance relation-based evolutionary algorithm for many-objective optimization. *IEEE Transactions on Evolutionary Computation* 20, 1 (2016), 16–37.

[111] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 913–923.