ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming

Yuan Yuan and Wolfgang Banzhaf

Abstract—Automated program repair is the problem of automatically fixing bugs in programs in order to significantly reduce the debugging costs and improve the software quality. To address this problem, test-suite based repair techniques regard a given test suite as an oracle and modify the input buggy program to make the whole test suite pass. GenProg is well recognized as a prominent repair approach of this kind, which uses genetic programming (GP) to rearrange the statements already extant in the buggy program. However, recent empirical studies show that the performance of GenProg is not satisfactory, particularly for Java. In this paper, we propose ARJA, a new GP based repair approach for automated repair of Java programs. To be specific, we present a novel lower-granularity patch representation that properly decouples the search subspaces of likely-buggy locations, operation types and potential fix ingredients, enabling GP to explore the search space more effectively. Based on this new representation, we formulate automated program repair as a multi-objective search problem and use NSGA-II to look for simpler repairs. To reduce the computational effort and search space, we introduce a test filtering procedure that can speed up the fitness evaluation of GP and three types of rules that can be applied to avoid unnecessary manipulations of the code. Moreover, we also propose a type matching strategy that can create new potential fix ingredients by exploiting the syntactic patterns of the existing statements. We conduct a large-scale empirical evaluation of ARJA along with its variants on both seeded bugs and real-world bugs in comparison with several state-of-the-art repair approaches. Our results verify the effectiveness and efficiency of the search mechanisms employed in ARJA and also show its superiority over the other approaches. In particular, compared to jGenProg (an implementation of GenProg for Java), an ARJA version fully following the redundancy assumption can generate a test-suite adequate patch for more than twice the number of bugs (from 27 to 59), and a correct patch for nearly four times of the number (from 5 to 18), on 224 real-world bugs considered in Defects4J. Furthermore, ARJA is able to correctly fix several real multi-location bugs that are hard to be repaired by most of the existing repair approaches.

Index Terms—Program repair, patch generation, genetic programming, multi-objective optimization, genetic improvement.

1 Introduction

Automated program repair [1]–[3] aims to automatically fix bugs in software. This research field has recently stirred great interest in the software engineering community since it tries to address a very practical and important problem. Automatic repair techniques generally depend on an oracle which can consist of a test suite [4], pre-post conditions [5] or an abstract behavioral model [6].

Our study focuses on the test-suite based program repair that considers a given test suite as an oracle. The test suite should contain at least one initially failing test case that exposes the bug to be repaired and a number of initially passing test cases that define the expected behavior of the program. In terms of test-suite based repair, a bug is said to be *fixed* or *repaired* if a repair approach generates a patch that makes its whole test suite pass. The patch obtained can be referred to as a *test-suite adequate patch* [7].

GenProg [4], [8], [9] is one of the most well-known repair approaches for test-suite based program repair. This general approach is based on the *redundancy assumption* (i.e., the fix ingredients for a bug already exist elsewhere in the buggy program); and it uses genetic programming (GP) [10], [11] to search for potential patches that can fulfill the test suite. Although GenProg has been well recognized as a state-of-the-art repair approach in the literature, it has aroused certain academic controversies among some researchers.

First, Qi et al. [12] studied to what extent GenProg can benefit from GP. Their results on GenProg benchmarks [9] indicate that just replacing GP in GenProg with random search can improve both repair effectiveness and efficiency, thereby questioning the necessity and effectiveness of GP in automated program repair. Second, an empirical study conducted by Qi et al. [13] pointed out that the overwhelming majority of patches reported by GenProg are incorrect and are equivalent to a single functionality deletion. Here we do not focus on the potential incorrectness of the patches that is mainly due to the weakness of the test suite rather than the repair approaches [14]. Our major concern is that GenProg usually generates nonsensical patches (e.g., a single deletion), which challenges the expressive power of GP to produce meaningful or semantically complex repairs. Lastly, a recent large-scale experiment [7] showed that an implementation of GenProg for Java (called jGenProg) can find a test-suite adequate patch for only 27 out of 224 real-world Java bugs, and only five of them were identified as correct. Obviously, the performance of GenProg for Java is currently far from satisfactory.

Considering these adverse reports about GenProg, it is necessary to revisit the most salient features of GenProg that qualify it as a well-established repair system. We think there are at least two. One is that GenProg can scale to

Y. Yuan and W. Banzhaf are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (e-mail: yyuan@msu.edu; banzhafw@msu.edu).
Manuscript received xxx; revised yyy.

large programs, mainly owing to its patch representation [9]. Another is that GenProg can potentially address various types of bugs because the expressive power of GP allows for diverse transformations of code. In particular, GP can change multiple locations of a program simultaneously, so that GenProg is likely to fix multi-location bugs that cannot be handled by most of the other repair approaches. The scalability of GenProg is visible since it has been widely applied to large real-world software [7], [9], making it distinguished from those approaches (e.g., SemFix [15] and SearchRepair [16]) that are largely limited to small programs. However, as mentioned before, the expressive power of GenProg inherited from GP has not been well supported and validated by recent experimental studies in the literature [7], [12], [13]. We think it is very important to shed more light on this issue and then address it, which would make GP really powerful in automated program repair.

Generally, a successful repair system consists of two key elements [13]: 1) a *search space* that contains correct patches; 2) a *search algorithm* that can navigate the search space effectively and efficiently. For the search space, GenProg uses the redundancy assumption, which has been largely validated by two independent empirical studies [17], [18]. This leaves the search algorithm as a bottleneck that might make GenProg unable to fulfill its potential for generating nontrivial patches. The reason could be that the search ability of the underlying GP algorithm in GenProg is not strong enough to really sustain its expressive power.

Given this analysis, our primary goal is to improve the effectiveness of the search via GP for program repair. To this end, we present a new GP based repair system for **a**utomated **r**epair of **Ja**va programs, called ARJA. ARJA is mainly characterized by a novel patch representation for GP, multi-objective search, a test filtering procedure and several strategies to reduce the search space. Our results indicate that an ARJA version that fully follows the redundancy assumption can generate a test-suite adequate patch for 59 real bugs in four projects of Defects4J [19] as opposed to only 27 reported by jGenProg [7]. By manual analysis, we find that this ARJA version can synthesize a correct patch for at least 18 bugs in Defects4J as opposed to only 5 by jGenProg. To our knowledge, some of the 18 correctly fixed bugs have never been repaired correctly by the other repair approaches. Furthermore, ARJA is able to correctly fix several multi-location bugs that are hard to be addressed by most of the existing repair approaches.

The main contributions of this paper are as follows:

1) The solution representation is a key factor that concerns the performance of GP. We propose a novel patch representation for GP based program repair, which properly decouples the search subspaces of likely-buggy locations, operation types and replacement/insertion code.
2) We propose to formulate automated program repair as a multi-objective optimization problem and employ NSGA-II [20] to search for potential repairs.
3) We present a procedure to filter out some test cases that are not influenced by GP from the given test suite, so as to speed up the fitness evaluation of GP significantly.
4) We propose to use not only variable scope but also method scope to restrict the number of potential replacement/insertion statements at the destination.
5) We introduce three types of rules which are integrated into three different phases of ARJA search (i.e., operation initialization, ingredient screening and solution decoding), in order to reduce the search space effectively.
6) Although our study mainly focuses on improving the search algorithm, we also make an effort to enrich the search space reasonably beyond reusing code already extant in the program. To that end, we propose a type matching strategy which can create promising new code for bug fixing by leveraging syntactic patterns of existing code.
7) We conduct a large-scale experimental study on 18 seeded bugs and 224 real-world bugs, from which some new findings and insights are obtained.
8) We develop a publicly-available program repair library for Java, which currently includes the implementation of our proposed approach (i.e., ARJA) and three previous repair approaches originally designed for C (i.e., GenProg [9], RSRepair [12] and Kali [13]). It is expected that the library can facilitate further replication and research on automated Java software repair.

The remainder of this paper is organized as follows. In Section 2, we provide the background knowledge and motivation for our study. Section 3 describes the proposed repair approach in detail. Section 4 presents the experimental design and Section 5 reports the experimental results obtained. Section 6 discusses the threats to validity. Section 7 lists the related work on test-suite based program repair. Finally, Section 8 concludes and outlines directions for future work.

2 Background and Motivation

In this section, we first provide background information of ARJA, including multi-objective genetic programming and the GenProg system. Then, we describe the goal and motivation of our study.

2.1 Multi-Objective Genetic Programming

Genetic programming (GP) is a stochastic search technique which uses an evolutionary algorithm (EA), often derived from a genetic algorithm (GA), to evolve computer programs towards particular functionality or quality goals. In GP, a computer program (i.e., phenotype) is encoded as a genome (i.e., genotype), which can be a syntax tree [10], an instruction sequence [21], or other linear and hierarchical data structures [22]; a fitness function is used to evaluate each genome in terms of how well the corresponding program works on the predefined task. GP starts with a population of genomes that is typically randomly produced and evolves over a series of generations progressively. In each generation, GP first selects a portion of the current population based on fitness, and then performs crossover and mutation operators on those selected to generate new genomes which would form the next population.

Traditionally, the aim of GP is to create a working program *from scratch*, in order to solve a problem encapsulated by a fitness function. Due to the limited size of successful programs that GP can generate, GP research and applications over the past few decades mainly focused on predictive modeling (e.g., medical data classification [23], energy

consumption forecasting [24] and scheduling rules design [25]), where a program is usually just a symbolic expression. It was not until recently that GP was used to evolve real-world software systems [4], [9], [26], [27]. Here, instead of starting from scratch, such GP applications generally take an *existing* program as a starting point, and then improve it by optimizing its functional properties (e.g., by fixing bugs) [4], [9] or non-functional properties (e.g., execution time and memory consumption) [26]–[29]. This paradigm of applying GP is formally called genetic improvement [30] in the literature.

Moreover, most previous usages of GP only consider a single objective. However, there usually exist several competing objectives that need to be optimized simultaneously in a real-world task, which can be formulated as a multi-objective optimization problem (MOP). Mathematically, a general MOP can be stated as

$$\begin{aligned} \min \mathbf{f}(\mathbf{x}) &= (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x}))^{\mathrm{T}} \\ \text{subject to } \mathbf{x} &\in \Omega \subseteq \mathbb{R}^n \end{aligned} \tag{1}$$

$\mathbf{x}$ is a n-dimensional decision vector in the decision space Ω, and $\mathbf{f} : \Omega \to \Theta \subseteq \mathbb{R}^m$, is an objective vector consisting of m objective functions, which maps the decision space Ω to the attainable objective space Θ. The objectives in Eq.(1) are often in conflict with each other (i.e., the decreasing of one objective may lead to the increasing of another), so there is typically no single solution that optimizes all objectives simultaneously. To solve a MOP, attention is paid to approximating the *Pareto front* (PF) that represents optimal trade-offs between objectives. The concept of a PF is formally defined as follows.[1]

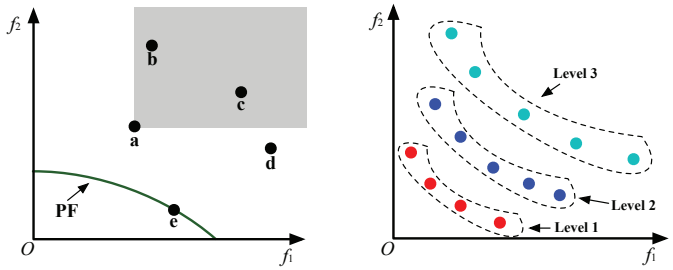
Definition 1 (Pareto Dominance). A vector $\mathbf{p} = (p_1, p_2, \ldots, p_m)^{\mathrm{T}}$ is said to *dominate* another vector $\mathbf{q} = (q_1, q_2, \ldots, q_m)^{\mathrm{T}}$, denoted by $\mathbf{p} \prec \mathbf{q}$, iff $\forall i \in \{1, 2, \ldots, m\} : p_i \leq q_i$ and $\exists j \in \{1, 2, \ldots, m\} : p_j < q_j$.

Definition 2 (Pareto Front). The Pareto front of a MOP is defined as $PF := \{\mathbf{f}(\mathbf{x}^*) \in \Theta \mid \nexists \mathbf{x} \in \Omega, \mathbf{f}(\mathbf{x}) \prec \mathbf{f}(\mathbf{x}^*)\}$.

From Definition 2, the PF is a subset of solutions which are not dominated by any other solution.

Due to the population-based nature of EAs, they are able to approximate the PF of a MOP in a single run by obtaining a set of non-dominated objective vectors, from which a decision maker can select one or more for their specific needs. These EAs are called multi-objective EAs (MOEAs). A comprehensive survey of MOEAs can be be found in ref. [31]. Considering a suitable multi-objective scenario, multi-objective GP evolves a population of candidate programs for multiple goals using a MOEA approach.

Fig. 1(a) illustrates Pareto dominance for a MOP with two objectives. According to Definition 1, all objective vectors within the grey rectangle (e.g., $\mathbf{b}$ and $\mathbf{c}$) are dominated by $\mathbf{a}$, and $\mathbf{a}$ and $\mathbf{d}$ are non-dominated by each other as $\mathbf{a}$ is better for f_1 while $\mathbf{d}$ is better for f_2. Because $\mathbf{e}$ is on the PF, no objective vectors in Θ can dominate it. To provide sufficient selection pressure toward the PF, many Pareto dominance-based MOEAs, e.g., NSGA-II [20], introduce elitism based on non-dominated sorting. Fig. 1(b) illustrates the non-dominated sorting procedure, where the union population (combination of current population and offsprings) is divided into different non-domination levels. The solutions on the first level are obtained by collecting every solution that is not dominated by any other one in the union population. To find the solutions on the j-th ($j \geq 2$) level, the solutions on the previous $j-1$ levels are first removed, and the same procedure is repeated again. The solutions on a lower level will have a higher priority to enter into the next population than those of a higher level.



(a) Pareto dominance (b) Non-dominated sorting

Fig. 1. Illustration of Pareto dominance and non-dominated sorting.

2.2 A Brief Introduction to GenProg

GenProg [4], [9] is a generic approach that uses GP to automatically find repairs of a buggy program. GenProg takes a buggy program as well as a test suite as input and generates one or more test-suite adequate patches for output. The test suite is required to contain initially passing tests to model the expected program functionality and at least one initially failing test to trigger the bug. To obtain a program variant that passes all the given tests, GenProg modifies the buggy program by using a combination of three kinds of statement-level edits (i.e., delete a destination statement, replace a destination statement with another statement, and insert another statement before a destination statement). In the early versions of GenProg [4], [8], [32], each genome in the underlying GP is an abstract syntax tree (AST) of the program combined with a weighted path through it. However, the AST based representation does not scale to large programs, since the memory consumed by a population of program ASTs is usually unaffordable. Inspired by ref. [33], Le Goues et al. [9] addressed the scalability problem of GenProg by using the *patch representation* instead of the AST representation. Specifically, each genome now is represented as a patch, which is stored as a sequence of edit operations parameterized by AST node numbers (e.g., Replace(7, 13), see Fig. 2(a)). The phenotype of a genome of this representation is a modified program obtained by applying the patch to the buggy input program.

Based on the patch representation, GenProg uses either a variant of uniform crossover or single-point crossover to generate offspring solutions. The experimental results in ref. [34] showed that single-point crossover is usually preferable because it can achieve a better comprise between success rate and repair effort. The single-point crossover randomly chooses a cut point each in two parent programs, and the genes beyond the cut points are swapped between the two parents to produce two offspring solutions. Fig.

1. In the following we shall assume that the goal is to minimize objectives.

2(b) illustrates the single-point crossover in GenProg. In crossover, we can only expect material in the two parents to be differently combined, but not newly generated.

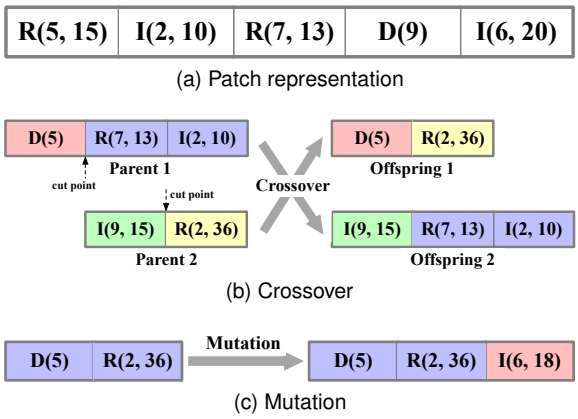


Fig. 2. Illustration of patch representation, crossover, and mutation in GenProg. For brevity, "D" denotes a delete operation; "R" a replace; and "I" an insert. The integers denote the AST node numbers of the corresponding statements. D(a) means that delete "a"; R(a, b) means that replace "a" with "b"; I(a,b) means that insert "b" before "a".

The mutation operator is therefore very important in GenProg, because it is responsible for introducing new edit operations into the population. To conduct the mutation on a solution, first each potential faulty statement is chosen as a destination statement with a probability of mutation rate weighted by its suspiciousness. Once a destination statement is determined, an operation type is randomly chosen from three types (i.e., "Delete", "Replace" and "Insert"). In case of "Replace" or "Insert", a second statement (i.e., replacement/insertion code) is randomly chosen from those statements which only reference variables in the *variable scope* at the destination and are visited by at least one test. Each edit operation created in this way is appended to a list of edits in the solution under mutation. Fig. 2(c) illustrates the mutation operator in GenProg.

The overall procedure of GenProg is summarized as follows. First, by running all tests, GenProg localizes potential buggy statements and gives each of them a weight measuring its suspiciousness. Then, GP searches to obtain an initial population by independently mutating N (the population size) copies of the empty patch. In each generation of GP, GenProg uses tournament selection based on fitness (i.e., the ability to pass the tests) to select $N/2$ parent solutions for mating from the current population, and conducts crossover (as in Fig. 2(b)) on the parents to generate $N/2$ offspring solutions. Afterwards, it conducts one mutation (as in Fig. 2(c)) on each parent and each offspring. The parents together with the offsprings will form the next population. The GP loop is terminated when a program variant passes all given tests or another termination criterion is reached.

2.3 Goal and Motivation

Our overall goal in this study is to develop a more powerful GP based system for automated repair of Java programs. To this end, we conduct an analysis of the potential limitations of GenProg so as to guide the design of our new system. There are several deficiencies in GenProg that motivated us to pursue this goal, which are discussed as follows.

2.3.1 High-Granularity Patch Representation

In GenProg, each gene in the patch representation (see Fig. 2(a)) is a high-granularity edit operation in which the operation type, likely-buggy location (i.e., destination statement), and replacement/insertion code are *invisible* to the crossover (see Fig. 2(b)) and mutation (see Fig. 2(c)) operators. Manipulating such high-level units via GP would hinder the efficient recombination of genetic information between solutions. This is mainly because good partial information of an edit operation (e.g., a promising operation type, an accurate faulty location, and a useful replacement/insertion code) cannot be propagated from one solution to others. For illustration purposes, suppose there is a bug that requires two edit operations to be repaired: D(5), R(2, 10), and there are two candidate solutions in the population that are the same as "Parent 1" and "Parent 2" in Fig. 2(b) respectively. As can be seen, the two candidate solutions together contain all the material to compose the correct patch. The crossover in GenProg can easily propagate the desired edit D(5) in "Parent 1" to offspring solutions. However, because such crossover does not introduce any new edit, it cannot produce R(2,10), even though I(2, 10) in "Parent 1" and R(2, 36) in "Parent 2" are both one modification away and their desired partial information can be obtained from each other. The mutation in GenProg creates new edits from scratch, where the operation types and replacement/insertion code are just randomly chosen from all those available. Thus, there is only a slim chance for mutation to produce exactly R(2,10).

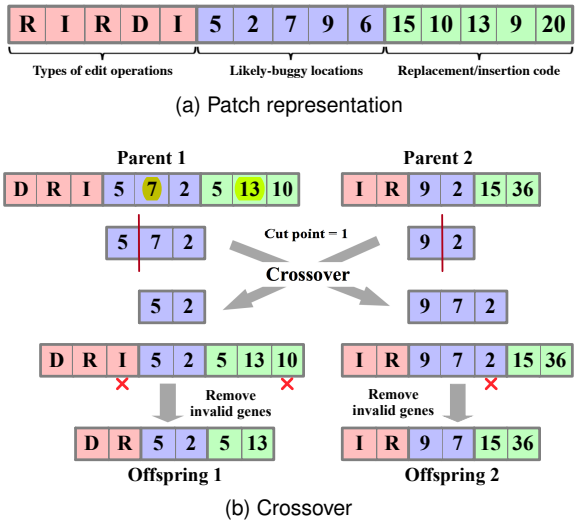


Fig. 3. Illustration of patch representation and crossover introduced by Oliveira et al. [35].

Recently, Oliveira et al. [35] noted the above limitation of GenProg and presented a lower-granularity patch representation, which decouples the three subspaces corresponding to three kinds of partial information in an edit operation. Using this representation, the patch represented in Fig. 2(a) can be reformulated as that shown in Fig. 3(a), where the representation is divided into three different parts: the first part is a list of operation types, the second a list of likely-buggy locations, and the third a list of replacement/insertion code. Such a representation makes the search explicitly explore three subspaces and thus overcomes GenProg's

drawback of over-constraining the search ability to some extent, but it can also lead to new problems.

One problem is that crossover becomes less elegant. Fig. 3(b) illustrates a kind of crossover suggested in ref. [35]. The crossover first randomly chooses one part in the representation, and conducts single-point crossover only on that part keeping the other two parts unchanged. However, due to different numbers of genes in the three parts after crossover, there exist some invalid genes that should be removed to obtain a final offspring solutions. The removal of invalid genes will potentially result in information loss. To relieve this issue, Oliveira et al. [35] introduced a memorization scheme to reuse invalid genes. However, this additional procedure will inevitably complicate the crossover and increase the computational burden.

Another problem is more essential. That is, crossover on such a representation exchanges information of operation types and replacement/insertion code between different likely-buggy locations very frequently. However, this situation is indeed not desirable, because every likely-buggy location has its own syntactic/semantic context and preferable operation types and replacement/insertion code can vary a lot. Moreover, due to scoping issues, just the available replacement/insertion code can be quite different at different likely-buggy locations, so exchanging replacement/insertion code between such locations usually even results in an uncompilable program variant. For example, in Fig. 3(b), the statement numbered 13 is originally the replacement code at the location 7, whereas after crossover, it becomes that at the location 2; however, certain variables in this statement can be out of the scope at the new location.

Our study aims to propose a novel lower-granularity patch representation that can address the limitations of GenProg's representation while avoiding the above two problems caused by the representation introduced in ref. [35].

2.3.2 High Complexity of Repairs

GP based repair methods (e.g., GenProg) can carry out complex transformations of original code via GP, so they have the potential to repair a buggy program that really requires many modifications. However, the great expressive power of GP can also pose the risk of finding a repair that is much more complex than necessary. In practice, a simpler patch is usually preferred, for two reasons: (1) According to Occam's razor, a simpler patch could have better generalization ability to unseen test cases, and (2) such a patch is generally more comprehensible and maintainable for humans. Hence, it is beneficial to direct a program repair tool to generate simple or small patches. A recent effort toward this goal can be seen in ref. [36], where a semantics-based repair method was proposed to find simple program repairs.

Given that existing GP based repair methods (e.g., GenProg) do not explicitly take into account the simplicity of a patch during search, our study aims to incorporate a consideration for the simplicity of repairs into the search process of GP, so as to introduce search bias towards small patches.

2.3.3 Expensive Fitness Evaluation

In GenProg, all given tests need to be run in order to evaluate the fitness of a solution accurately. However it is usually computationally expensive to run all the associated tests of real-world software. For example, there are more than 5200 JUnit tests in the current Commons Math project,[2] and it would take about 200 seconds to execute all of them for just one fitness evaluation on an Intel Core i5 2.9GHz processor with 16Gb of memory. Expensive fitness evaluations will limit the use of a reasonably large number of generations or population size in GP, thereby greatly limiting the potential of GP for program repair. To relieve this problem, Fast et al. [37] proposed to just use a random sample of given tests for each fitness evaluation. Although that strategy can increase efficiency, it will unavoidably reduce the precision of the search.

Our study argues that not all given tests are necessary for fitness evaluation. In fact, many can be omitted in order to speed up the fitness evaluation, but without affecting the precision of search.

2.3.4 Incomplete Scope

To follow the principle of high cohesion, modern Java software systems generally involve a considerable number of method invocations (e.g., up to 20,000 in the Commons Math project). However, when choosing replacement/insertion code, GenProg checks the validity of the code only according to the variable scope at destination, without considering the method scope. This practice may be effective for C programs considered in ref. [9], but is not good enough for real-world Java programs investigated in our study. For example, for a buggy version of the Commons Math project, we find that, among the available replacement/insertion statements (satisfying the variable scope only) at each destination, on average about 40% invoke invisible methods and are indeed invalid.

Hence, our study proposes to consider both variable scope and method scope when choosing valid replacement/insertion code for a likely-buggy location, so as to improve the success rate of compiling the modified programs.

2.3.5 Limited Utilization of Existing Code

Today, large Java projects are commonly developed by many programmers, each of whom is responsible for only one or several modules. Although the names of important APIs or even field variables can be determined in the software design phase, the names of local variables and private methods are generally chosen based on the preference of the responsible programmer, which leads to the fact that even variables or methods with similar functions can have different names in different Java files. Thus, for a likely-buggy location, it is sometimes possible that we can make an invalid statement become its hopeful replacement/insertion code by replacing the invisible variables or methods with similar ones in the scope. In other words, the underlying pattern in a statement other than the statement itself can also be exploited to acquire useful replacement/insertion code. GenProg does not create any new code, in which the replacement/insertion code is just taken from somewhere

2. Apache Commons Math, http://commons.apache.org/math

else in the buggy program without change. This practice may fail to make the most of the existing code.

Our study aims to present a strategy that can exploit the pattern of the existing code appropriately, so as to create some new replacement/insertion statements that are potentially useful.

2.3.6 Lack of Adequate Prior Knowledge

GenProg can conduct any deletion, replacement or insertion operations on the possibly faulty statements, provided that the replacement/insertion code meets the variable scope. However, from the view of an experienced programmer, some operations indeed make little sense, which is mainly due to the following two reasons.

One reason is that although a replacement/insertion statement conforms to the scope of variables and methods at a destination, it can still violate other Java language specifications when it is pasted to that place. Another reason is that certain operations either disrupt the program considered too much or have no effect at all. For example, in Fig. 4, if we delete the variable declaration statement (at line 1092), all the remaining statements will be invalidated immediately since they all reference the variable `cloned`. Moreover, even if a variable declaration statement should be deleted, leaving it as a redundant statement generally does not influence the correctness of the program. Thus the deletion operation here is not desired and should be disabled.

```
1092 final StrTokenizer cloned = (StrTokenizer)
         super.clone();
1093 if (cloned.chars != null) {
1094   cloned.chars = cloned.chars.clone();
1095 }
1096 cloned.reset();
1097 return cloned;
```

Fig. 4. The code snippet excerpted from the Commons Lang project.[3]

Our study aims to encode such prior knowledge of programmers as a number of rules, which can be integrated into the proposed repair method flexibly. We expect that the search space of GP can be reduced effectively with these rules. Note that our aim is very different from that of ref. [38]. The rules considered in our study disallow definitely unnecessary operations rather than likely unpromising ones, so they generally do not restrict the expressive power of GP.

3 Approach

This section presents our generic approach to automatically finding the test-suite adequate patches via multi-objective GP. This approach is implemented as a tool called ARJA that repairs Java code.

3. Apache Commons Lang, http://commons.apache.org/lang

3.1 Overview

In a nutshell, ARJA works as depicted in Fig. 5. ARJA takes a buggy program and a set of associated JUnit tests as the input. Among the tests, at least one *negative* (i.e., initially failing) test is required to be included, which exposes the bug to be fixed. All the remaining are *positive* (i.e., initially passing) tests, which describe the expected program behavior. The basic goal of ARJA is to modify the program so that all tests pass. Its process is composed of the following main steps.

Given the input, a fault localization technique is used to identify potentially buggy statements which are to be manipulated by GP. Meanwhile, coverage analysis is conducted to record every statement that is visited by any JUnit test. These statements collected in the coverage analysis (referred to as *seed statements* in ARJA) provide the source of the replacement/insertion code (referred to as *ingredient statements* in ARJA). Note that fault localization and coverage analysis both require the Eclipse AST parser to transform the line information of code to the corresponding Java statements.

Once the likely-buggy statements are identified, they will be put to use immediately in two ways. (1) positive tests unrelated to these statements are filtered out from the original JUnit test suite, so that a reduced set of tests can be obtained for further use. (2) the scope of variables and methods is determined for the location of each of these statements.

Then the ingredient statements for each likely-buggy statement considered are selected from the seed statements in view of the current variable and method scope. For convenience, a likely-buggy statement along with the scope at its location and its corresponding ingredient statements is called a *modification point* in short.

Before entering into the genetic search, the types of operations on potentially buggy statements should be defined in advance. Similar to GenProg [9], ARJA also uses three kinds of operations: *delete*, *replace*, and *insert*. More specifically, for each likely-buggy statement, ARJA either deletes it, replaces it with one of its ingredient statements, or inserts one of its ingredient statements before it. Note that although only three operation types are currently used, users can add other possible types [39] into ARJA easily due to its flexible design.

With a number of modification points, a subset of original JUnit tests, and the available operation types in place, ARJA encodes a program patch with a novel GP representation. Based on this new representation, a MOEA is employed to evolve the patches by simultaneously minimizing the failure rate on tests and patch size. Finally, the non-dominated solutions obtained with a failure rate of 0 are output as test-suite adequate patches.

Notably, ARJA is also characterized by a module that reduces the search space based on some specific rules. These rules can be divided into three different types which are specially designed for operation initialization, ingredient screening and decoding in multi-objective GP, respectively. Applying such rules allows the modified program to be compiled successfully with a higher probability, while focusing the search on more promising regions of the search space.

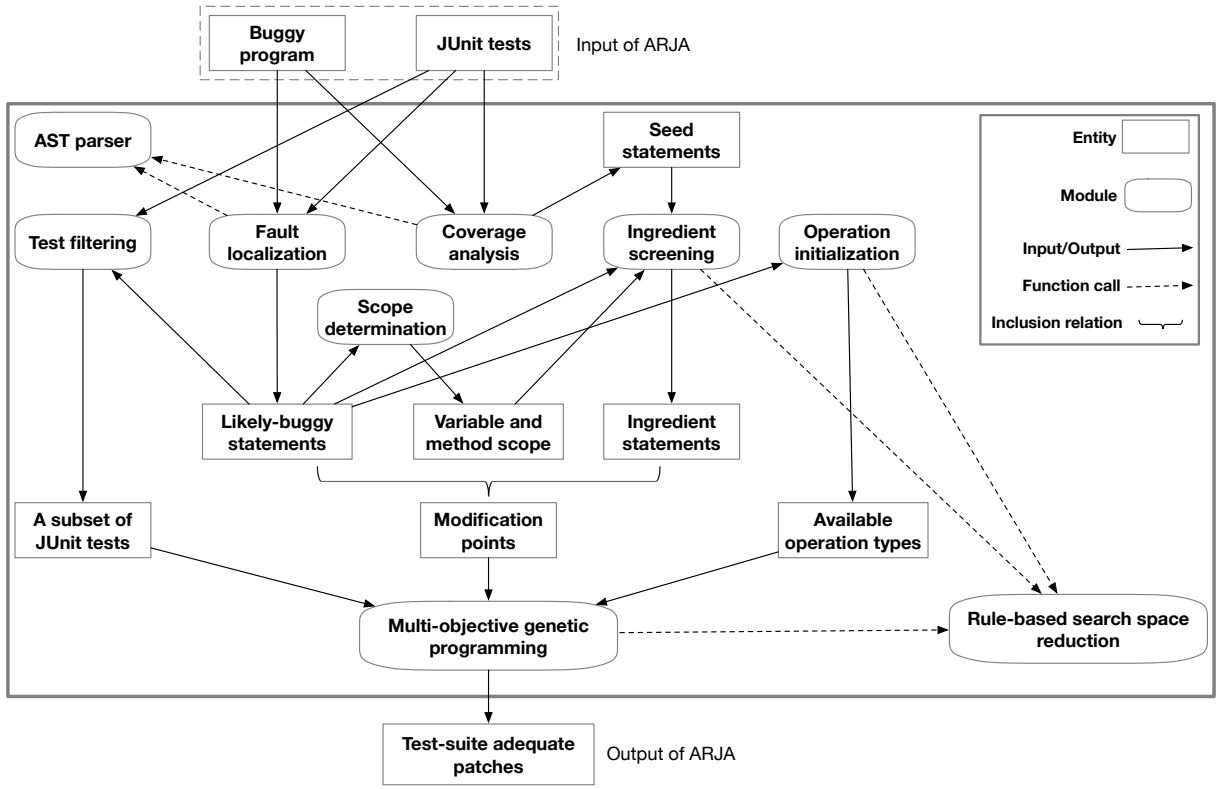


Fig. 5. Overview of the proposed automated program repair approach, i.e., ARJA.

3.2 Fault Localization and Coverage Analysis

For fault localization, ARJA uses an existing spectrum-based technique called Ochiai [40]. It computes a suspiciousness measure of a line of code (lc) as follows:

$$susp(lc) = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}} \quad (2)$$

where N_{CF} and N_{CS} are the number of negative tests and positive tests that visit the code lc, respectively, and N_F are the total number of negative tests. Fault localization analysis returns a number of potentially faulty lines, each represented as a tuple $(cls, lid, susp)$. cls and lid are the name of the Java class and the line number in this class, respectively, which are used to identify a line uniquely, and $susp \in [0, 1]$ is the corresponding suspiciousness score.

To look for seed statements, ARJA implements a strategy presented in ref. [9] to reduce the number of seed statements and to choose those more related to the given JUnit tests. That is, coverage analysis is conducted to find the lines of code that are visited by at least one test, each of which forms a tuple (cls, lid).

After the above phases, the Eclipse AST parser is used to parse the potentially faulty lines and seed lines into the likely-buggy statements and seed statements respectively. For duplicate seed statements, only one of them is recorded.

Note that in ARJA, we do not consider all potentially faulty statements. Instead, only a part of them are selected according to their suspiciousness in order to reduce the search space. The number of statements can be controlled in ARJA by either of two parameters denoted $\gamma_{\min}$ and $n_{\max}$. $\gamma_{\min}$ quantifies the minimum suspiciousness score for statements to be considered, while $n_{\max}$ determines that at most $n_{\max}$ likely-buggy statements with highest suspiciousness are chosen. If both $\gamma_{\min}$ and $n_{\max}$ are set, ARJA uses the smaller number determined by either of them.

3.3 Test Filtering

In ARJA, we propose to ignore those positive tests that are unrelated to the considered likely-buggy statements. To be specific, for each positive test, we record all the lines of code covered during its execution, and if these lines do not include any of the lines associated with the likely-buggy statements selected, we can safely filter out this positive test. This strategy usually enables us to obtain a much smaller test suite compared to the original one, which can significantly speed up the fitness evaluation in GP search without influencing its effectiveness.

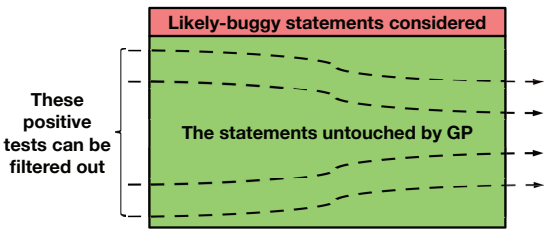


Fig. 6. Illustration of the execution path of the positive test that can be filtered out.

The rationale for test filtering can be explained as follows: since positive tests ignored by this procedure do not touch any code used by GP, every program variant

produced by GP will always pass these test cases once it can be compiled and run successfully. Thus, after discarding such positive tests, the relative superiority or inferiority in fitness between any two GP individuals will be kept the same, and the search behavior of GP will not be changed. Fig. 6 illustrates the principle of test filtering.

3.4 Scope Determination

For each likely-buggy statement considered, most seed statements cannot become an ingredient statement. This is mainly because these seed statements access variables or methods that are invisible at the location of the likely-buggy statement. To identify as many of them as possible, we have to determine the scope (i.e., all the visible variables and methods) at that location.

Note that unlike GenProg [9], ARJA considers not only the *variable* scope but also the *method* scope, which can improve the chance of the modified Java program to being compiled successfully. The necessity of this practice for Java has been discussed in Section 2.3.4.

Suppose Cls and Med are the class and the method where a likely-buggy statement appears, respectively. According to the Java language specification, ARJA collects three kinds of variables to constitute the variable scope: the visible field variables in Med, the parameter variables of Med, and the local variables in Med defined before the location of the likely-buggy statement. Among them, the first kind of variables has three sources: the field variables declared in Cls, the field variables inherited from the parent classes of Cls, and the field variables declared in the outer classes (if they exist) of Cls. As for the method scope, ARJA collects the visible methods in Med, which have three sources similar to the visible field variables.

Note that besides the variable and method names, ARJA also records their corresponding type information and modifiers to make the scope more accurate. For a method, the type information includes both parameter types and the return type.

3.5 Ingredient Screening

This procedure aims to select the ingredient statements for each likely-buggy statement considered. In this phase, ARJA first adopts the *location awareness* strategy introduced in ref. [41]. This strategy defines three alternative ingredient modes (i.e., *File*, *Package*, *Application*), which are used to specify the places where ingredients are taken from. Suppose a likely-buggy statement is located in the file Fl that belongs to the package Pk, then the "File" and "Package" modes mean that this likely-buggy statement can only take its ingredient statements from Fl and Pk, respectively. The "Application" mode means that the ingredient statements can come from anywhere in the entire buggy program. Compared to the "Application" mode, the other two modes can significantly restrict the space of ingredients, which may help to find the repairs faster or find more of them.

With the location awareness strategy incorporated, ARJA provides the following two alternative approaches for ingredient screening, namely a direct approach and a type matching based approach.

3.5.1 Direct Approach

The direct approach works as follows. For each considered likely-buggy statement, all the seed statements are examined one by one. If a seed statement does not come from the place specified by the ingredient mode, it will just be ignored. Otherwise, we extract the variables and methods accessed by this seed statement. For example, for the following statement:

```
ret = 1.0 - getDistribution().beta +
    b.regularizedBeta(getProbability(),
    this.x + 1.0, getTrials() - this.x);
```

the extracted variables include `ret`, `b`, and `x`; and the extracted methods are `getDistribution`, `getProbability` and `getTrials`. Note that it is not necessary to consider the variable `beta` and the method `regularizedBeta`, because their accessibility generally only depends on the visibility of `getDistribution` and `b`, respectively.

For each extracted variable/method, we check whether the one with the same name and the compatible type exists in the variable/method scope (determined in Section 3.4). Only when all of them have the corresponding ones in the variable/method scope, this seed statement can become an ingredient statement of the likely-buggy statement.

3.5.2 Type Matching Based Approach

As mentioned in Section 2.3.5, it could be demanding for a seed statement to only access the variables/methods visible at the location of the likely-buggy statement. Indeed, the pattern of a seed statement can sometimes also be useful.

To exploit such patterns, the type matching based approach goes a step further compared to the direct approach. When certain variables or methods extracted from a seed statement cannot be found in the variable or method scope, the type matching based approach does not discard this seed statement immediately. Instead, it tries to map each variable or method out of scope to one with the compatible type in scope. To restrict the complexity, we follow two guidelines in type matching: 1) Different variables/methods in a seed statement must correspond to different ones in the scope; 2) If there is more than one variable/method with a compatible type, the one with the same type is preferred.

If the type matching is successful, the modified seed statement will become an ingredient statement. Fig. 7 and Fig. 8 illustrate how type matching works for variables and methods respectively, using toy programs.

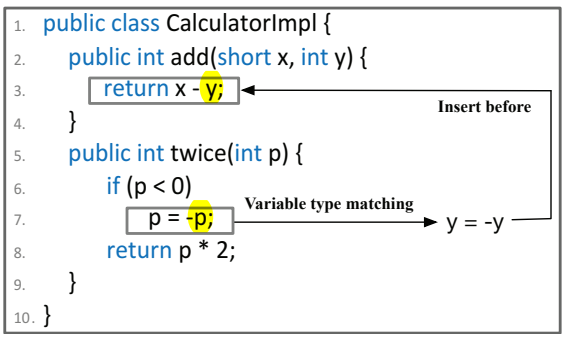


Fig. 7. Illustration of the type matching for variables.

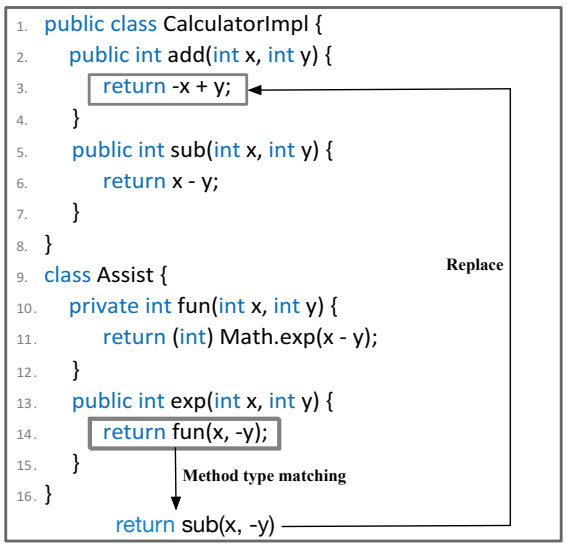


Fig. 8. Illustration of the type matching for methods.

In Fig. 7, the statement at line 3 is a faulty one, and the bug can be fixed by inserting `y=-y` before this statement. ARJA cannot repair this fault without type matching since no such fix ingredient exists in the current program. However, if type matching is enabled, `y=-y` can be generated via a seed statement `p=-p`, by mapping the variable `p` to `y`. Similarly, in Fig. 8, the method `fun` is mapped to `sub`, and an ingredient statement `sub(x,-y)` is generated via `fun(x,-y)`, which can be used to fix the bug at line 3.

3.6 Evolving Program Patches

Suppose that we have selected n likely-buggy statements using the procedure in Section 3.2, and each of them has a set of ingredient statements found by the method in Section 3.5. Thus, we have n modification points, each of which has a likely-buggy statement and its corresponding ingredient statements.

With the n modification points along with the available operation types and a reduced set of JUnit tests (obtained in Section 3.3), we can now encode a program patch as a genome and evolve a population of such tentative patches via multi-objective GP. The details are given as follows.

3.6.1 Solution Representation

To encode a patch, we first order the n modification points in some manner. For the j-th modification point, where $j = 1, 2, \ldots, n$, the corresponding set of ingredient statements is denoted by I_j and the statements in I_j are also ordered arbitrarily. Moreover, the set of operation types is denoted by O and the elements in O are numbered starting from 1. Note that the ID number for each modification point, each statement in I_j, or each operation type in O is fixed throughout the evolutionary process.

In ARJA, we propose a new patch representation that can decouple the search subspaces of likely-buggy locations, operation types and ingredients perfectly. Specifically, each solution can be represented as $\mathbf{x} = (\mathbf{b}, \mathbf{u}, \mathbf{v})$, which contains three different parts and each part is a vector with size n.

The first part, denoted by $\mathbf{b} = (b_1, b_2, \ldots, b_n)$, is a binary vector with n bits b_j ($b_j \in \{0, 1\}, j = 1, 2, \ldots, n$). b_j indicates whether or not the patch $\mathbf{x}$ chooses to edit the likely-buggy statement in the j-th modification point. Fig. 9 illustrates the representation of $\mathbf{b}$.

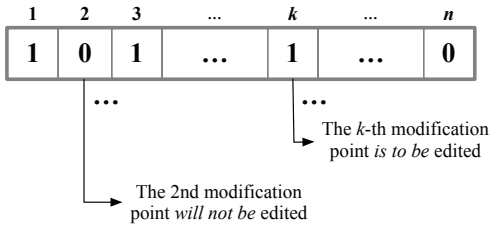


Fig. 9. Illustration of the first part (i.e., $\mathbf{b}$) of the representation.

The second part, denoted by $\mathbf{u} = (u_1, u_2, \ldots, u_n)$, is a vector with n integers, where $u_j \in [1, |O|], j = 1, 2, , \ldots, n$. u_j means that the patch $\mathbf{x}$ chooses the u_j-th operation type in the set O for the j-th modification point. In Fig. 10, we illustrate the representation of $\mathbf{u}$.

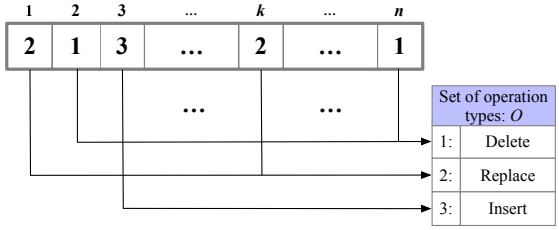


Fig. 10. Illustration of the second part (i.e., $\mathbf{u}$) of the representation.

Similar to $\mathbf{u}$, the third part (i.e., $\mathbf{v} = (v_1, v_2, \ldots, v_n)$) is also a vector with n integers, where $v_j \in [1, |I_j|], j = 1, 2, \ldots, n$. v_j indicates that the patch $\mathbf{x}$ chooses the v_j-th ingredient statement in the set I_j for the j-th modification point. Fig. 11 illustrates the representation of $\mathbf{v}$.

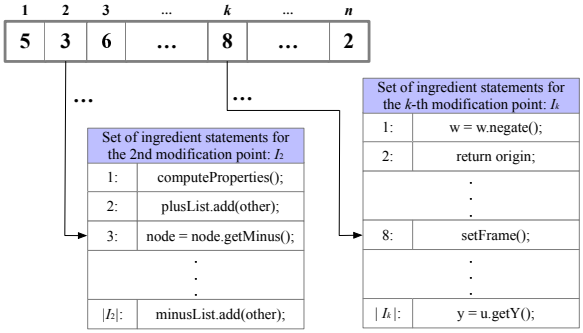


Fig. 11. Illustration of the third part (i.e., $\mathbf{v}$) of the representation.

As can be seen, b_j, u_j and v_j together determine what the patch $\mathbf{x}$ does to the j-th modification point. For example, for the patch represented in Figs. 9, 10 and 11, it replaces the likely-buggy statement in the k-th modification point with `setFrame()`. Suppose the operation types in O are numbered as in Fig. 10, the whole procedure to apply a patch $\mathbf{x}$ (i.e., the decoding procedure) is described in Algorithm 1.

Algorithm 1: The procedure to apply a patch $\mathbf{x}$

Input: n modification points; the set of operation types O; a patch $\mathbf{x} = (\mathbf{b}, \mathbf{u}, \mathbf{v})$.
Output: A modified program.

```
for j = 1 to n do
    if b_j = 1 then
        st ← the likely-buggy statement in the j-th modification point;
        if u_j = 1 then
            Delete st;
        else
            st* ← the v_j-th ingredient statement in I_j;
            if u_j = 2 then
                Replace st with st*;
            else if u_j = 3 then
                Insert st* before st;
```

3.6.2 Population Initialization

For a specific problem, it is usually better to use the initialization strategy based on prior knowledge instead of random initialization, which could help genetic search find desirable solutions more quickly and easily.

In ARJA, we initialize the first part (i.e., $\mathbf{b}$) of each solution by exploiting the output of fault localization. Suppose $susp_j$ is the suspiciousness of the likely buggy statement in the j-th modification point, then b_j is initialized to 1 with the probability $susp_j \times \mu$ and 0 with $1 - susp_j \times \mu$, where $\mu \in (0, 1)$ is a predefined parameter. The remaining two parts (i.e., $\mathbf{u}$ and $\mathbf{v}$) of each solution are just initialized randomly (i.e., u_j and v_j are initialized to an integer randomly chosen from $[1, |O|]$ and $[1, |I_j|]$ respectively).

3.6.3 Fitness Evaluation

In ARJA, we formulate automated program repair as a multi-objective search problem. To evaluate the fitness of a solution $\mathbf{x}$, we propose a multi-objective function to simultaneously minimize two objectives, namely *patch size* (denoted by $f_1(\mathbf{x})$) and *weighted failure rate* (denoted by $f_2(\mathbf{x})$).

The patch size is given by Eq. (3), which indeed refers to the number of edit operations contained in the patch.

$$f_1(\mathbf{x}) = \sum_{i=1}^{n} b_i \tag{3}$$

The weighted failure rate measures how well the modified program (obtained by applying the patch $\mathbf{x}$) passes the given tests. We can formulate it as follows:

$$f_2(\mathbf{x}) = \frac{|\{t \in T_f \mid \mathbf{x} \text{ fails } t\}|}{|T_f|} + w \times \frac{|\{t \in T_c \mid \mathbf{x} \text{ fails } t\}|}{|T_c|} \tag{4}$$

where T_f is the set of negative tests, T_c is the reduced set of positive tests obtained through test filtering, and $w \in (0, 1]$ is a global parameter which can introduce a bias toward negative tests. If $f_2(\mathbf{x}) = 0$, $\mathbf{x}$ does not fail any test and represents a test-adequate patch.

By simultaneously minimizing f_1 and f_2, we prefer test-adequate patches of smaller size. Note that if the modified program fails to compile or runs out of time when executing the tests, we set both of the objectives to $+\infty$. Moreover, $f_1 = 0$ would be meaningless for program repair since no modifications are made to the orginal program. So, once f_1 is equal to 0 for a solution $\mathbf{x}$, f_1 and f_2 are immediately reset to $+\infty$, forcing such solutions to disappear with elite selection.

ARJA also provides two optional strategies that are implemented for improving computational efficiency of f_1 and f_2 respectively. For f_1, the user can choose to set a threshold n_e to avoid disrupting the original program too much. Once n_e is set beforehand and f_1 is larger than n_e for a solution, f_1 is reset to n_e and only n_e edit operations are used on modification points with the largest suspiciousness to compute f_2. This strategy can also help to accelerate convergence of the genetic search.

As for f_2, the strategy is similar to that used by GenProg [9]. That is, if $|T_c|$ is still too large, a random sample of T_c (different at each time) can be used for each computation of f_2 instead of T_c. Only when the resulting f_2 is 0, all the tests in T_c are considered to recalculate f_2. The rationale for this strategyis that GP can also work well with a noisy fitness function [37].

3.6.4 Genetic Operators

The genetic operators (i.e., crossover and mutation) are executed to produce offspring solutions in GP. To inherit good traits from parents, crossover and mutation are applied to each part of the solution representation separately.

For the first part (i.e., $\mathbf{b}$), we use half uniform crossover (HUX) and bit-flip mutation. As for both of the remaining parts (i.e., $\mathbf{u}$ and $\mathbf{v}$), we adopt single-point crossover and uniform mutation, because of their integer encoding. Fig. 12 illustrates how crossover and mutation are executed on two parent solutions. For brevity, only one offspring is shown in this figure.

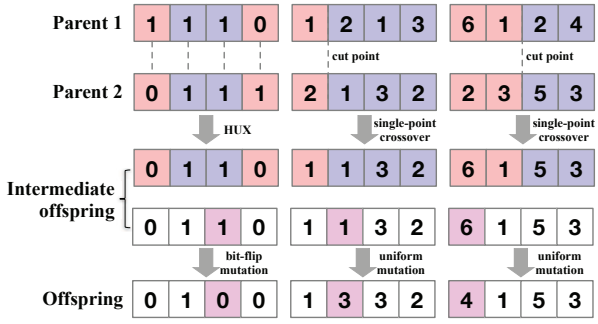


Fig. 12. Illustration of crossover and mutation in ARJA.

3.6.5 Using NSGA-II

Generally, based on the proposed solution representation, any MOEA can serve the purpose of evolving the patches for a buggy program. In ARJA, we employ NSGA-II [20] as the search algorithm, which is one of the most popular MOEA.

The NSGA-II based search procedure for finding test-adequate patches can be summarized as follows. First, an initial population with N (the population size) solutions is produced by using the initialization strategy presented in Section 3.6.2. Then the algorithm goes into a loop until the

TABLE 1
The rules integrated in ARJA for customizing the operation types for each modification point

No.	Rule	rationale
1	Do not delete a variable declaration statement (VDS).	Deleting a VDS is usually very disruptive to a program, and keeping a redundant VDS usually does not influence the correctness of a program.
2	Do not delete a `return`/`throw` statement which is the last statement of a method not declared `void`.	Avoid returning no value from a method that is not declared `void`.

TABLE 2
The rules integrated in ARJA for further filtering the ingredients for each modification point

No.	Rule	rationale
1	The `continue` statement can be used as the ingredient only for a likely-buggy statement in the loop.	The keyword `continue` cannot be used out of a loop (i.e., `for`, `while` or `do-while` loop).
2	The `break` statement can be used as the ingredient only for a likely-buggy statement in the loop or in the `switch` block.	The keyword `break` cannot be used out of a loop (i.e., `for`, `while` or `do-while` loop) or a `switch` block.
3	A `case` statement can be used as the ingredient only for a likely-buggy statement in a `switch` block having the same enumerated type.	The keyworkd `case` cannot be used out of a `switch` block, and the value for a `case` must be the same enumerated type as the variable in the `switch`.
4	A `return`/ `throw` statement can be used as the ingredient only for a likely-buggy statement in a method declaring the compatible return/throw type.	Avoid returning/throwing a value with non-compatible type from a method.
5	A `return`/ `throw` statement can be used as the ingredient only for a likely-buggy statement that is the last statement of a block.	Avoid the unreachable statements.
6	A VDS can be used as the ingredient only for another VDS having the compatible declared type and the same variable names.	Avoid using an edit operation with no effect on the program or disrupting the program too much.

maximum number of fitness evaluations is reached. In each generation g, binary tournament selection [20] and the genetic operators described in Section 3.6.4 are applied to the current population P_g to generate an offspring population Q_g. Then the N best solutions are selected from the union population $U_g = P_g \cup Q_g$ by using fast non-dominated sorting and crowding distance comparison (based on the two objectives formulated in Section 3.6.3). The resulting N best solutions constitute the next population P_{g+1}.

Finally, the obtained non-dominated solutions with $f_2 = 0$ are output as test-adequate patches found by ARJA. If no such solutions exist, ARJA fails to fix the bug.

3.7 Rule-Based Search Space Reduction

ARJA provides three types of rules that can be integrated into its three different procedures (i.e., operation initialization, ingredient screening and solution decoding) respectively. By taking advantage of these rules, we can not only increase the chance of the modified program to compile successfully , but also avoid some meaningless edit operations, thereby reducing the search space.

Note that when the rules are integrated into ARJA, the related procedures described in the previous subsections will be modified as discussed next.

3.7.1 Customizing the Operation Types

The first type of rules are used to customize the operation types for each modification point. Such rules are invoked in the operation initialization procedure since they only involve likely-buggy statements and the operation types.

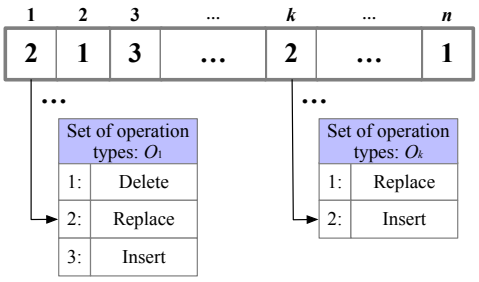


Fig. 13. Illustration of $\mathbf{u}$ for the purpose of customizing the operation types.

For a modification point, certain operation types in O may not be available according to the predefined rules. Currently in ARJA, we provide two rules of this type which are shown in Table 1. Suppose O_j is the set of available operation types for the j-th modification point, where $O_j \subseteq O, j = 1, 2, \ldots, n$. Fig. 13 illustrates the $\mathbf{u}$ vector for the purpose of customizing the operation types. Unlike in Fig 10, each modification point is associated with its own set of operation types (i.e., O_j), and u_j means that the patch chooses the u_j-th operation type in O_j (instead of O).

TABLE 3
The rules integrated in ARJA for disabling certain specific operations

No.	Rule	rationale
1	Do not replace a statement with the one having the same AST.	Avoid using an edit operation with no effect on the program.
2	Do not replace a VDS with the other kinds of statements.	Avoid disrupting the program too much.
3	Do not insert a VDS before a VDS.	The same with No. 1.
4	Do not insert a `return`/`throw` statement before any statement.	Avoid the unreachable statements.
5	Do not replace a `return` statement (with return value) that is the last statement of a method with the other kinds of statements.	Avoid returning no value from a method that is not declared void.
6	Do not insert an assignment statement before an assignment statement with the same left-hand side.	The same with No. 1.

3.7.2 Further Filtering the Ingredients

The second type of rules concerns likely-buggy statements and the ingredient statements, which are employed to further filter the ingredient statements in an ingredient screening procedure. Such rules can help to remove undesirable ingredients which pass the scope check of variables and methods. For example, a `continue` statement does not contain any variable or method invocation, but it can only be used in a loop. Table 2 lists the rules of this type integrated into ARJA, and also explains their rationale.

By applying these rules to I_j (obtained by the procedure in Section 3.5), we can generate a reduced set I'_j where $I'_j \subseteq I_j$. I'_j will become the set of ingredients for the j-th modification point instead of I_j. To illustrate $\mathbf{v}$ in this scenario, we can just replace I_j with I'_j in Fig. 11, with v_j then indicating that the patch chooses the v_j-th ingredient statement in I'_j.

3.7.3 Disabling Certain Specific Operations

The third type of rules involve at least the operation types and the ingredient statements. Such rules are used to ignore certain specific edit operations when decoding a solution $\mathbf{x}$ for fitness evaluation. Table 3 shows the rules of this type integrated in ARJA together with their rationale.

With these rules, if $b_j = 1$, the corresponding operation on the j-th modification point will not be conducted immediately as in Algorithm 1. Instead, we first check whether this operation conforms to one rule listed in Table 3. Once any of the rules is met, the operation will be disabled (equivalent to resetting b_j to 0).

4 Experimental Design

This section explains the design of our experimental study, including the research questions to be answered, the repair systems involved, the datasets of bugs used, and the evaluation protocol for comparing different repair approaches.

4.1 Research Questions

To conduct the general evaluation of ARJA, we seek to answer the following research questions in this study.

RQ1: How useful is the test filtering procedure in speeding up the fitness evaluation?

In this research question, we show what percentage of CPU time for fitness evaluation can be saved by test filtering.

RQ2: Does random search really outperform genetic search in automated program repair?

Previous work by Qi et al. [12] claimed that random search outperforms GP in terms of both repair effectiveness and efficiency. Their work targeted C programs and was based on GenProg framework. We are interested in revisiting this claim based on ARJA that targets Java programs.

RQ3: What are the benefits of formulating program repair as a multi-objective search problem?

We would expect that the multi-objective formulation described in Section 3.6.3 can help ARJA generate simpler patches compared to a single-objective formulation. Besides this, we investigate whether the multi-objective formulation can provide other benefits.

RQ4: Is ARJA better than state-of-the-art repair methods in terms of fixing multi-location bugs?

As stated in Section 1, one prominent feature of GP based repair approaches is that they have the potential to fix multi-location bugs. ARJA is a new GP based repair system, thus it is necessary to assess its superiority in this respect.

In RQs 2–4, our main concern is the *search ability* of ARJA (including its variants) and other related repair methods based on the redundancy assumption. So, here we only use ARJA *without* type matching for the simplicity and fair comparison.

RQ5: How useful is the type matching strategy when the fix ingredients do not exist in the current buggy program?

Type matching can reasonably create ingredient statements that do not appear in the buggy program. We investigate whether these newly generated ingredients can be exploited effectively by ARJA to fix some bugs.

RQ6: How well does ARJA perform in fixing real-world bugs compared to the existing repair approaches?

It is of major interest to address real-world bugs in program repair. We need to know whether ARJA can work on real-world bugs in large-scale Java software systems, beyond fixing seeded bugs.

RQ7: To what extent can ARJA synthesize semantically correct patches?

Since the empirical work by Qi et al. [13], it has been a hot question whether the patches generated by test-suite based repair approaches are correct beyond passing the test suite. We manually check the correctness of the patches synthesized by ARJA in our experiments.

RQ8: Why can't ARJA generate test-suite adequate patches for some bugs?

Sometimes, ARJA fails to find any test-suite adequate patch. We examine several reasons for failure.

RQ9: How long is the execution time for ARJA on one bug?

The computational cost of one repair is also an important concern for users. We want to see whether the repair cost of ARJA is acceptable for practical use.

4.2 Repair Systems Under Investigation

There are mainly five repair systems involved in our experiments, which are ARJA, GenProg [9], RSRepair [12], Kali [13] and Nopol [42].

Our repair system, ARJA, is implemented with Java 1.7 on top of jMetal 4.5.[4] jMetal [43] is a Java based framework that includes a number of state-of-the-art EAs, particularly MOEAs. It is used to provide computational search algorithms (e.g., NSGA-II) in our work. ARJA parses and manipulates Java source code using the Eclipse JDT Core[5] package. The fault localization in ARJA is implemented with Gzoltar 0.1.1.[6] Gzoltar [44] is a toolset which determines suspiciousness of faulty statements using spectrum-based fault localization algorithms. Both coverage analysis and test filtering in ARJA are implemented with JaCoCo 0.7.9,[7] which is a Java code coverage library. For the sake of reproducible research, the source code of ARJA is available at GitHub.[8] In addition, several ARJA variants have also been implemented to answer different research questions.

RSRepair is a repair method that always uses the population initialization procedure of GenProg to produce candidate patches. Kali generates a patch by just removing or skipping statements. Strictly speaking, Kali cannot be regarded as a "program repair" technique, but it is a very suitable technique for identifying weak test suites or under-specified bugs [7]. GenProg, RSRepair, and Kali were originally developed for C programs. According to the details given in the corresponding papers [9], [12], [42], we carefully reimplement the three systems for Java under the same infrastructure of ARJA. Our source code for the three systems is publicly released along with ARJA.[8] Note that a program repair library named Astor [41] also provides the implementation of GenProg and Kali for Java, which are called jGenProg and jKali respectively in the literature [7], [41]. But to conduct controlled experiments, we only use our own implementation, unless otherwise specified.

4. jMetal, http://jmetal.sourceforge.net
5. Eclipse JDK Core, https://www.eclipse.org/jdt/core/index.php
6. Gzoltar, http://www.gzoltar.com. The version 0.1.1 cannot localize the faults in a constructor. So when repairing some bugs in Defects4J that are located in a constructor, we switch to the version 1.6.2 although the new version appears much more computationally intensive
7. JaCoCo, http://www.eclemma.org/jacoco
8. ARJA, https://github.com/yyxhdy/arja

Nopol is a state-of-the-art semantic-based repair method for fixing conditional statement bugs in Java programs. The code of Nopol[9] has been released by the original authors.

4.3 Datasets of Bugs

In our experiments, we use both seeded bugs and real-world bugs to evaluate the performance of repair systems.

To answer RQ1 and RQs 6–9, we adopt a dataset consisting of four open-source Java projects (i.e., JFreeChart,[10] Joda-Time,[11] Commons Lang, and Commons Math) from Defects4J [19]. Defects4J[12] has been a popular database for evaluating Java program repair systems [7], [45]–[48], because it contains well-organized real-world Java bugs. Table 4 shows the basic information of the 224 real-world bugs considered in Defects4J, where the number of lines of code and the number of JUnit tests are extracted from the latest buggy version of each project. Note that Defects4J indeed contains another two projects, namely Closure Compiler[13] and Mockito[14]. Following the practice in refs. [7], [46], [47], we do not consider the two projects in the experiments. Closure Compiler is dropped since it uses the customized testing format rather than the standard JUnit tests; Mockito is ignored because it is a very recent project added into the Defects4J framework and its related artifacts are still in an unstable phase.

TABLE 4
The descriptive statistics of 224 bugs considered in Defects4J

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

To address RQs 2–4, we use a dataset of seeded bugs rather than Defects4J. We think that Defects4J is not well suited to the purpose of distinguishing clearly the search ability of the repair systems considered. The reasons are listed as follows:

1) Many bugs (e.g., C2, L4 and M12) in Defects4J cannot be localized by state-of-the-art fault localization tools (e.g., Gzoltar). In such a case, fault localization rather than the search is responsible for the failure.
2) Although existing empirical studies [17], [18] validated the redundancy assumption by examining a large number of commits (16,071 in ref. [17] and 15,723 in ref. [18]), Defects4J contains a relatively small number of bugs and it may not conform well to this general assumption. Indirect evidence is that jGenProg can find patches for only 27 out of 224 bugs in Defects4J, as reported by Martinez et al. [7]. In such a case, it is the inadequate search space that matters, rather than the search ability.

9. Nopol, https://github.com/SpoonLabs/nopol
10. JFreeChart, http://www.jfree.org/jfreechart
11. Joda-Time, http://www.joda.org/joda-time
12. Defects4J, https://github.com/rjust/defects4j
13. Closure Compiler, https://github.com/google/closure-compiler
14. Mockito, http://site.mockito.org

3) As indicated in ref. [7], among 27 bugs fixed by GenProg, 20 bugs can also be fixed by jKali. This means that for the overwhelming majority of these bugs, the search method can find a trivial patch (e.g., deleting a statement) that fulfills the test-suite by just focusing on a very limited search space. So, evaluation on such bugs cannot truly reflect the difference between redundancy-based repair methods in exploring a huge search space of potential fix ingredients.

The dataset of seeded bugs for RQs 2–4 are generated by the following procedures. First, we select the correct version of M85 as a target program, since it has a moderate number (i.e., 1983) of JUnit tests. Then, we randomly select k redundant[15] statements from two Java files (i.e., NormalDistributionImpl.java and PascalDistributionImpl.java). Last, we produce a buggy program by performing mutation to each of the k statements. Note that not every buggy program obtained in this way is a suitable test bed, we choose some of them according to the following principles:

1) The fault localization technique can identify all the faulty locations of the seeded bug. This rules out the influence of fault localization.
2) Any nonempty subset of the k mutations should make at least one test fail. Generally, this ensures that the seeded bug is really a multi-location bug when $k > 1$.
3) Kali cannot generate any test-suite adequate patch for the seeded bug. This challenges the search ability in finding nontrivial or complex repairs.

We vary k from 1 to 3 and finally collect a dataset consisting of 13 bugs of this kind, denoted by F1–F13. Among the 13 bugs, k is set to 1 for F1 and F2, 2 for F3–F9, and 3 for F10–F13. So, all bugs except F1 and F2 are multi-location bugs. Because the mutated statements are redundant, the redundancy-based repair systems (e.g., GenProg and RSRepair) can fix any bug in this dataset, assuming that their search ability is strong enough.

For RQ5, we use a similar method to generate a dataset of seeded bugs. The difference is that among the k statements to be mutated, at least one is non-redundant. So, it is expected that the redundancy assumption does not completely hold for such bugs, which can be used to verify the effectiveness of the type matching strategy. We set k to 2 and collect 5 bugs in this category, denoted by H1–H5.

To facilitate experimental reproducibility, we make the two datasets of seeded bugs available on GitHub as well.[16]

4.4 Evaluation Protocol

In our experiments, we always use "Package" as the ingredient mode in ARJA, GenProg, and RSRepair, although there exist two other alternatives as introduced in Section 3.5. To reduce the search space, we integrate all three types of rules (see Section 3.7) into ARJA. When investigating RQs 2–4, the direct approach (see Section 3.5.1) is used in ARJA for ingredient screening.[17] To save computational time, we employ the test filtering procedure (see Section 3.3) in all repair approaches implemented by ourselves (i.e., ARJA, GenProg, RSRepair, and Kali).

TABLE 5
The parameter setting for ARJA in the 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 modification points	40
μ	Refer to Section 3.6.2	0.06
w	Refer to Section 3.6.3	0.5
p_c	Crossover probability	1.0
p_m	Mutation probability	$1/n$

Table 5 presents the basic parameter setting for ARJA in the experimental study, where n is the number of modification points determined by $\gamma_{\min}$ and $n_{\max}$ together (see Section 3.2). To ensure a fair comparison, parameters N, G, $\gamma_{\min}$ and $n_{\max}$ in GenProg and RSRepair are set to the same values as shown in Table 5. The global mutation rate in GenProg and RSRepair is set to 0.06 since it is similar to parameter μ in ARJA. Corresponding to $w = 0.5$, negative tests are weighted twice as heavily as positive tests in fitness evaluation of GenProg and RSRepair. Each trial of ARJA, GenProg and RSRepair is terminated after the maximum number of generations is reached. Moreover, in Kali, we also use $\gamma_{\min}$ and $n_{\max}$ to restrict the number of modification points considered. Their values are set to 0.1 and 40 respectively.

ARJA, GenProg, and RSRepair are all stochastic search methods. To compare their search ability properly, each of these algorithms performs 30 independent trials for each seeded bug considered in RQs 2–5. There are several metrics involved for evaluating the search performance, which are explained as follows:

1) "Success": the number of trials that produce at least one test-suite adequate patch among 30 independent trials. This is regarded as the primary metric.
2) "#Evaluations" and "CPU (s)": the average number of evaluations and the average CPU time needed to find the first test-suite adequate patch in a successful trial.
3) "Patch Size": the average size of the smallest test-suite adequate patch obtained in a successful trial. Here "size" means the number of edits contained in a patch.
4) "#Patches": the average number of different test-suite adequate patches found in a successful trial. Note that we may obtain test-suite adequate patches with various sizes in a trial, this metric only counts the number of those with the smallest size. Moreover, the difference between patches here is judged in terms of syntactics rather than semantics.

Following the practice in ref. [7], we perform only one trial of ARJA, GenProg, and RSRepair for each real-world bug considered in RQ6, in order to keep experimental time acceptable. It is indeed very CPU intensive to conduct a large-scale experiment on the 224 bugs from Defects4J. For example, Martinez et al. [7] ran each of three algorithms (i.e., jGenProg, jKali and Nopol) on each of the 224 bugs only once, which took up to 17.6 days of computational

15. Here "redundant" means that the same statement can be found otherwhere in the current package.

16. Seeded bugs, http://github.com/yyxhdy/SeededBugs

17. Hereafter, if "ARJA" represents a specific algorithm, it refers to the version that uses the direct approach for ingredient screening. The ARJA variants using type matching will be differentiated by subscripts

time on Grid'5000. However, we note that multiple trials are needed to rigorously assess the performance of ARJA, GenProg, and RSRepair due to their stochastic nature. We discuss this important threat to validity in Section 6.

Our experiments were all performed in the MSU High Performance Computing Center.[18] We use 2.4 GHz Intel Xeon E5 processor machines with 20 GB memory.

5 Experimental Results and Analysis

This section presents the results of our experimental study in order to address RQs 1–9 set out in Section 4.1.

5.1 Value of Test Filtering (RQ1)

To answer RQ1, we select the latest buggy versions of the four projects considered in Defects4J (i.e., C1, T1, L1 and M1) as the subject programs.

For each buggy program, we vary $\gamma_{\min}$ (i.e., the threshold of suspiciousness) from 0 to 0.2. For each $\gamma_{\min}$ value chosen, we use the test filtering procedure to obtain a subset of original JUnit tests and then record two metrics associated with the reduced test suite: the number of tests and the execution time. Fig. 14 shows the influence of $\gamma_{\min}$ on the two metrics for each subject program. For comparison purposes, the two metrics have been normalized by dividing by the original number of JUnit tests and the CPU time consumed by the original test suite respectively. Note that the fluctuation of CPU time in Fig. 14 is due to the dynamic behavior of the computer system.

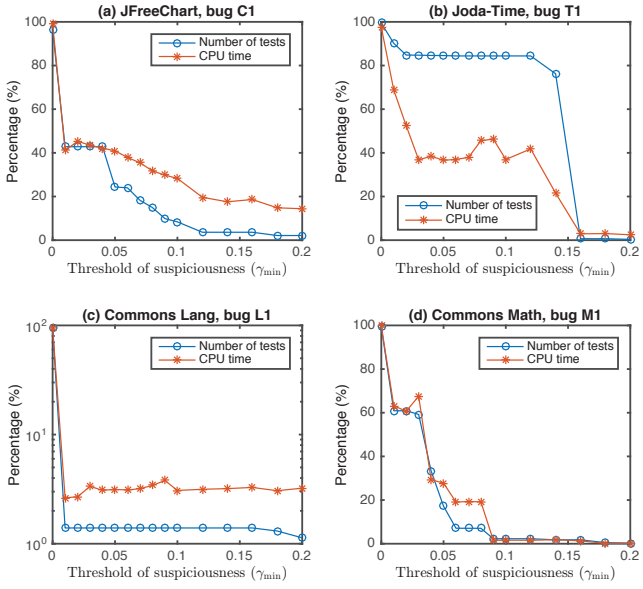


Fig. 14. Illustration of the value of the test filtering procedure. The base 10 logarithmic scale is used for the y axis in (c).

Judged from Fig. 14, test filtering can bring substantial benefits in terms of reduction of computational costs. With the increase in $\gamma_{\min}$ the number of tests that need to be considered and the corresponding CPU time consumed decrease significantly and quickly. Even if we set $\gamma_{\min}$ to a very small value (e.g., 0.01), test filtering can result in a considerable reduction of CPU time. Suppose $\gamma_{\min} = 0.01$, the percentage reduction is about 59% for C1, 31% for T1, 97% for L1 and 37% for M1. Generally, if we set $\gamma_{\min}$ to a larger value, we can consider a smaller test suite ifor fitness evaluation. However, it is not desirable to use too large a $\gamma_{\min}$ (e.g., 0.5), which would make repair approaches miss the actual faulty locations. In practice, we usually choose a moderate $\gamma_{\min}$ value (e.g., 0.1) to strike a compromise. Normally, test filtering can significantly speed up the fitness evaluation in such a case. For example, if we set $\gamma_{\min}$ to 0.1 for M1, the number of tests considered is reduced from 5,246 to 118, and the CPU time for one fitness evaluation is reduced from 210 seconds to 3.4 seconds. Suppose the termination criterion of ARJA is 2,000 evaluations, then we can save up to 115 hours for just a single repair trial.

For safety, we also conduct a post-run validation of the obtained patches on the original test suite. We find that if a patch can fulfill the reduced test suite, it can also fulfill the original one, with no exceptions.

5.2 Genetic Search vs. Random Search (RQ2)

To compare the performance of genetic search and random search under the ARJA framework, we implemented an ARJA variant denoted as ARJA$_r$. The only difference between ARJA and ARJA$_r$ lies in that ARJA$_r$ always uses the initialization procedure of ARJA to generate candidate solutions and does not use the genetic operators described in Section 3.6.4. So, ARJA$_r$ purely depends on the random search and there is no cumulative selection. The relationship between ARJA and ARJA$_r$ is similar to that between GenProg and RSRepair. For a fair comparison, ARJA$_r$ also uses the parameter setting shown in Table 5 (excluding p_c and p_m).

Table 6 compares ARJA and ARJA$_r$ on F1–F13 in terms of the metrics "Success" and "#Evaluations". In this table, the meaning of k can be referred to in Section 4.3 and $|T_f|$ is the number of negative tests that trigger the bug. For brevity, the two columns will be omitted later in Tables 7 and 8.

TABLE 6
Comparison between genetic search and random search within the ARJA framework. (Average over 30 runs)

Bug Index	k	$\lvert T_f \rvert$	Success		#Evaluations	
			ARJA	ARJA$_r$	ARJA	ARJA$_r$
F1	1	3	30	10	297.67	507.60
F2	1	4	17	19	492.71	392.32
F3	2	4	26	3	494.24	668.00
F4	2	6	13	0	746.54	–
F5	2	8	30	3	384.63	111.00
F6	2	4	30	1	624.80	1229.00
F7	2	3	25	0	698.52	–
F8	2	6	29	4	376.21	505.50
F9	2	2	6	0	1028.00	–
F10	3	6	18	0	936.11	–
F11	3	6	20	0	777.70	–
F12	3	8	28	0	742.04	–
F13	3	4	20	0	762.30	–

"–" means the data is not available.

As can be seen from Table 6, on all the bugs considered except F2 and F5, ARJA achieves a much higher success rate and also requires less number of evaluations to find

18. MSU HPCC, https://wiki.hpcc.msu.edu/display/hpccdocs

a repair patch compared to ARJA$_r$. Moreover, ARJA is much more effective than ARJA$_r$ in synthesizing multi-line patches. For example, on each of F10–F13 which need at least three edit operations, ARJA$_r$ cannot find any test-suite adequate patch in any of the 30 trials, whereas ARJA still maintains good performance and succeeds in the majority of trials. For the bug F5, ARJA$_r$ appears more efficient since it can find a repair more quickly, but its repair success rate is very low (3 out of 30) and it is therefore not reliable. In contrast to ARJA$_r$, ARJA can always succeed in fixing the bug F5. As for F2, ARJA$_r$ performs slightly better than ARJA. The possible reason is that the fix of F2 only requires one insertion operation and ARJA$_r$ focuses more on a search space containing such simple repairs.

In summary, ARJA significantly outperforms ARJA$_r$ in terms of both repair effectiveness and efficiency, particularly on multi-location bugs, which indicates that genetic search is indeed more powerful than random search in automated program repair.

Note that our conclusion here contradicts that drawn by Qi et al. [12]. This can be attributed to the fact that the two studies are based on different algorithmic frameworks and different subject programs. In ref. [12], GenProg and RSRepair were compared on 24 C bugs from the GenProg benchmark. As pointed out in ref. [13], almost all the patches reported by GenProg and RSRepair for these 24 bugs are equivalent to a single functionality deletion modification. When searching such trivial repairs, the crossover operator in GenProg will become ineffective. The main reason is that GenProg crossover works on the granularity of edits (as mentioned in Section 2.3.1) and produces a new patch just by combining the edits from two parent solutions without creating any new material. But it is clear that the recombination of existing edits will not be helpful to find a patch that contains only a single edit. In addition, because RSRepair always uses the initialization procedure of GenProg, it has a very high chance to generate patches with only one edit when using a small global mutation rate (e.g., 0.06). Whereas in GenProg, the new edits will be appended to the existing patch, which means that GenProg intends to explore larger patches during the search. Nevertheless, this search characteristic of GenProg may make it less efficient than RSRepair in finding trivial patches that are test-suite adequate for the bugs considered in ref. [12]. We speculate that GenProg will outperform RSRepair in terms of generating nontrivial or complex repairs.

5.3 Multi-Objective vs. Single-Objective (RQ3)

To show the benefits of the multi-objective formulation, we develop an ARJA variant denoted as ARJA$_s$, which only minimizes the weighted failure rate (see Eq. (4)). To serve the purpose of single-objective optimization, ARJA$_s$ uses a canonical single-objective GA instead of NSGA-II to evolve patches. To ensure a fair comparison, the single-objective GA also employs binary tournament selection and the genetic operators introduced in Section 3.6.4, and the parameter setting of ARJA$_s$ is the same with that of ARJA.

In Table 7, we present the comparative results between ARJA and ARJA$_s$ on bugs F1–F13, where the metrics "Success", "Patch Size" and "#Patches" are considered. As expected, ARJA can really generate test-suite adequate patches that contain smaller number of edits. The only exception is F5, where the patch sizes obtained by ARJA and ARJA$_s$ have no obvious difference. Moreover, it can be seen that the average patch size obtained by ARJA is usually very close to the corresponding k value (see Table 6) of the bug, demonstrating the effective minimization of f_1 (see Eq. (3)) by NSGA-II. According to "#Patches", for every bug, ARJA can find notably more different test-suite adequate patches than ARJA$_s$ in a successful trial, which is expected to provide more adequate choice for the programmer.

TABLE 7
Comparison between multi-objective and single-objective formulations within the ARJA framework. (Average of 30 runs)

Bug Index	Success		Patch Size		#Patches	
	ARJA	ARJA$_s$	ARJA	ARJA$_s$	ARJA	ARJA$_s$
F1	30	26	2.00	3.04	16.50	1.73
F2	17	13	1.35	2.85	10.94	1.54
F3	26	4	2.12	3.25	4.00	1.00
F4	13	10	2.23	4.50	5.92	1.40
F5	30	30	2.67	2.50	7.23	4.27
F6	30	28	2.80	3.86	9.77	1.61
F7	25	11	2.24	5.91	6.00	1.27
F8	29	22	2.14	4.23	7.76	1.50
F9	6	0	2.33	–	2.50	–
F10	18	11	3.00	4.45	3.22	1.09
F11	20	18	3.00	6.11	3.20	1.17
F12	28	15	3.07	4.07	6.61	1.80
F13	20	15	3.15	5.47	4.60	1.27

"–" means the data is not available.

More interestingly, in terms of "Success" metric, we find that ARJA also clearly outperforms ARJA$_s$. Considering that this metric only concerns the weighted failure rate (f_2 formulated in Eq. (4)), our results suggest that the simultaneous minimization of f_1 and f_2 promotes the minimization of f_2 significantly. So, in the sense of search or optimization, f_1 can be seen as a helper objective in our multi-objective formulation of program repair. A similar phenomenon was also observed by some previous studies [49]–[51] on other applications, which is formally termed as *multi-objectivization* [49] in the literature. One possible reason for this improvement is that a helper objective can guide the search toward solutions containing better building blocks and helps the search to escape local minima [50].

To sum up, the multi-objective formulation helps to find simpler repairs (containing smaller number of edits) and also helps to find more of them. Furthermore, the multi-objective formulation can facilitate more effective search of test-suite adequate patches compared to the single-objective formulation.

5.4 Strength in Fixing Multi-Location Bugs (RQ4)

Most existing program repair systems (e.g., Nopol) can only generate single point repairs. GenProg and RSRepair are two state-of-the-art repair approaches that can target multi-location bugs. To assess the strength of ARJA in multi-location repair, we compare it with GenProg and RSRepair on the bugs F3–F13. F1 and F2 are not considered here since they belong to single-location bugs.

Note that ARJA does not take advantage of GenProg and RSRepair when comparing them on F3–F13, because all

TABLE 8
Comparison of ARJA, GenProg, and RSRepair on multi-location bugs. (Average of 30 runs)

Bug Index	Success			#Evaluations			CPU (s)			Patch Size		
	ARJA	GenProg	RSRepair	ARJA	GenProg	RSRepair	ARJA	GenProg	RSRepair	ARJA	GenProg	RSRepair
F3	26	23	0	494.24	628.64	–	634.91	193.69	–	2.12	3.09	–
F4	13	2	0	746.54	1240.50	–	980.77	1129.29	–	2.23	6.50	–
F5	30	11	4	384.63	1235.30	1000.50	98.33	248.73	138.80	2.67	4.10	2.00
F6	30	11	0	624.80	894.91	–	393.25	127.18	–	2.80	7.09	–
F7	25	4	0	698.52	820.00	–	461.89	186.60	–	2.24	7.00	–
F8	29	24	3	376.21	915.21	1028.33	551.50	678.05	507.51	2.14	6.79	2.33
F9	6	1	0	1028.00	225.00	–	962.62	52.15	–	2.33	2.00	–
F10	18	2	0	936.11	1896.00	–	190.01	280.70	–	3.00	4.50	–
F11	20	9	0	777.70	1325.00	–	532.54	378.61	–	3.00	11.33	–
F12	28	15	0	742.04	973.73	–	566.57	273.59	–	3.07	9.67	–
F13	20	8	0	762.30	1383.00	–	485.07	357.65	–	3.15	14.88	–

"–" means the data is not available.

three approaches are based on the redundancy assumption and the fix ingredients of F3–F13 exist in their search space.

Table 8 shows the comparative results of ARJA, GenProg and RSRepair on F3–F13. As can be seen, ARJA outperforms both GenProg and RSRepair on all the considered bugs in terms of success rate. Indeed, on most of these bugs, ARJA achieves a much higher success rate than its counterparts. Compared to GenProg and RSRepair, ARJA also generally requires much smaller number of evaluations to find a repair. Although GenProg achieves better "#Evaluations" on F9, the metric is computed only based on a single successful trial. Given that ARJA does more than GenProg and RSRepair in one fitness evaluation, we also report the results of "CPU (s)" to provide another reference for comparing the efficiency of the approaches. It can be seen that the overall CPU time consumed by ARJA is comparable to that by GenProg. Since RSRepair only fixes two bugs here with very low success rate, we cannot rate its efficiency. In terms of "Patch Size", ARJA usually finds a much simpler repair than GenProg. This could be explained by different search mechanisms of GenProg and ARJA. GenProg considers larger patches in each generation by appending new edits to the existing patches. So once GenProg cannot find a repair in the first few generations, it will usually obtain patches that contain a relatively high number of edits. Different from GenProg, ARJA prefers smaller patches throughout the search process. In addition, we find that GenProg performs significantly better than RSRepair on the multi-location bugs considered, which corroborates our speculation from Section 5.2.

In summary, ARJA exhibits critical superiority over two prominent repair approaches (i.e., GenProg and RSRepair) in fixing multi-location bugs.

5.5 Effect of Type Matching (RQ5)

To show the effect of the type matching strategy, we introduce three additional ARJA variants denoted as ARJA$_v$, ARJA$_m$ and ARJA$_b$. They do not use the direct approach for ingredient screening. Instead, ARJA$_v$ uses the type matching strategy just for variables (illustrated in Fig. 7), ARJA$_m$ uses this strategy for just methods (illustrated in Fig. 8), and ARJA$_b$ conducts type matching for both variables and methods.

Table 9 compares ARJA (without type matching), ARJA$_v$, ARJA$_m$ and ARJA$_b$ on bugs H1–H5, where "Success" is used as the comparison metric. From Table 9 we can see that, ARJA cannot find any test-suite adequate patch for all the bugs except H4, whereas ARJA$_v$ or ARJA$_b$ have a good chance to fix these bugs. This indicates type matching is a promising strategy that can help ARJA to fix some bugs whose fix ingredients do not exist in the buggy program considered. However, ARJA$_m$ does not perform very well here, which may imply that type matching for methods struggles to generate the fix ingredients for bugs H1–H3 and H5. Note that ARJA can fix bug H4, which means that, in terms of semantics, the repair mode of H4 still follows the redundancy assumption.

TABLE 9
Illustration of the type matching strategy (the metric "Success" is reported in this table) (30 runs)

Bug Index	k	$\|T_f\|$	ARJA	ARJA$_v$	ARJA$_m$	ARJA$_b$
H1	2	6	0	2	0	0
H2	2	6	0	24	0	20
H3	2	5	0	4	0	3
H4	2	2	28	16	25	19
H5	2	7	0	17	0	13

Although ARJA$_v$ and ARJA$_b$ perform much better overall than ARJA on these bugs, we note that they fail to achieve a very high success rate, particularly on H1 and H3. Considering that ARJA$_v$ and ARJA$_b$ indeed search over a much larger space of ingredient statements than ARJA, a possible reason is that the larger search space poses a serious difficulty for the underlying genetic search. More CPU time may help ARJA$_v$ and ARJA$_b$ to overcome this difficulty.

To sum up, the type matching strategy shows good potential to generate useful ingredient statements. These new ingredients can be exploited by the genetic search to fix bugs for which the redundancy assumption does not hold. However, the much larger search space also challenges the search ability of GP in the ARJA framework.

5.6 Evaluation on Real-World Bugs (RQ6)

In this subsection, we conduct a large-scale experiment on the Defects4J dataset, in order to show the superiority of the

TABLE 10
Results for 224 bugs considered in Defects4J with ten repair approaches. For each approach, we list the bugs for which the test-suite adequate patches are found

Project	ARJA	ARJA$_v$	ARJA$_m$	ARJA$_b$	GenProg
JFreeChart	C1, C3, C5, C7, C12, C13, C15, C19, C25	C1, C3, C5, C7, C12, C13, C15, C25	C1, C3, C5, C7, C12, C13, C15, C18, C19, C25	C1, C3, C7, C12, C13, C15, C19, C25	C1, C3, C7, C12, C13, C18, C25
	$\sum = 9$	$\sum = 8$	$\sum = 10$	$\sum = 8$	$\sum = 7$
JodaTime	T4, T11, T14, T15	T1, T4, T11	T4, T11, T15, T24	T1, T4, T11, T14, T17	T4, T11, T24
	$\sum = 4$	$\sum = 3$	$\sum = 4$	$\sum = 5$	$\sum = 3$
Commons Lang	L7, L16, L20, L22, L35, L39, L41, L43, L45, L46, L50, L51, L55, L59, L60, L61, L63	L7, L13, L14, L22, L35, L39, L43, L45, L51, L55, L59, L60, L63	L7, L20, L22, L27, L39, L43, L45, L50, L51, L55, L58, L59, L60, L61, L63	L7, L13, L14, L21, L22, L35, L39, L45, L46, L51, L55, L59, L60, L61, L63	L7, L22, L35, L39, L43, L51, L55, L59, L63
	$\sum = 17$	$\sum = 13$	$\sum = 15$	$\sum = 15$	$\sum = 9$
Commons Math	M2, M5, M6, M8, M20, M22, M28, M31, M39, M49, M50, M53, M56, M58, M60, M68, M70, M71, M73, M74, M80, M81, M82, M84, M85, M86, M95, M98, M103	M2, M6, M8, M20, M28, M31, M39, M49, M50, M53, M56, M70, M71, M73, M79, M80, M81, M82, M85, M95	M2, M5, M6, M8, M20, M22, M28, M31, M39, M40, M44, M49, M50, M53, M58, M70, M71, M73, M74, M79, M80, M81, M82, M84, M85, M86, M95, M98, M103	M2, M5, M6, M7, M8, M20, M28, M49, M50, M53, M58, M60, M71, M73, M80, M81, M82, M84, M85, M95, M103	M2, M6, M8, M20, M28, M31, M39, M40, M49, M50, M71, M73, M80, M81, M82, M85, M95
	$\sum = 29$	$\sum = 20$	$\sum = 29$	$\sum = 21$	$\sum = 17$
Total	59 (26.3%)	44 (19.6%)	58 (25.9%)	49 (21.9%)	36 (16.1%)

Project	RSRepair	Kali	jGenProg[1]	jKali[1]	Nopol[1]
JFreeChart	C1, C5, C7, C12, C13, C15, C25	C1, C5, C12, C13, C15, C25, C26	C1, C3, C5, C7, C13, C15, C25	C1, C5, C13, C15, C25, C26	C3, C5, C13, C21, C25, C26
	$\sum = 7$	$\sum = 7$	$\sum = 7$	$\sum = 6$	$\sum = 6$
JodaTime	T4, T11, T24	T4, T11, T24	T4, T11	T4, T11	T11
	$\sum = 3$	$\sum = 3$	$\sum = 2$	$\sum = 2$	$\sum = 1$
Commons Lang	L7, L22, L27, L39, L43, L51, L59, L60, L63	L7, L22, L27, L39, L44, L51, L55, L58, L63			L39, L44, L46, L51, L53, L55, L58
	$\sum = 9$	$\sum = 9$	$\sum = 0$	$\sum = 0$	$\sum = 7$
Commons Math	M2, M5, M6, M8, M18, M20, M28, M31, M39, M49, M50, M53, M58, M68, M70, M71, M73, M74, M80, M81, M82, M84, M85, M95, M103	M2, M8, M20, M28, M31, M32, M49, M50, M80, M81, M82, M84, M85, M95	M2, M5, M8, M28, M40, M49, M50, M53, M70, M71, M73, M78, M80, M81, M82, M84, M85, M95	M2, M8, M28, M32, M40, M49, M50, M78, M80, M81, M82, M84, M85, M95	M32, M33, M40, M42, M49, M50, M57, M58, M69, M71, M73, M78, M80, M81, M82, M85, M87, M88, M97, M104, M105
	$\sum = 25$	$\sum = 14$	$\sum = 18$	$\sum = 14$	$\sum = 21$
Total	44 (19.6%)	33 (14.7%)	27 (12.1%)	22 (9.8%)	35 (15.6%)

[1] The results are organized according to those reported in ref. [7].

proposed repair approaches in fixing real-world bugs. Our experiment here is similar to that by Martinez et al. [7] but involves a larger number of repair approaches. Specifically, we consider ARJA along with its three variants (i.e., ARJA$_v$, ARJA$_m$ and ARJA$_b$), GenProg, RSRepair and Kali, which are all implemented by ourselves under the same infrastructure. Moreover, for our comparison purposes, we also include the results of jGenProg, jKali and Nopol reported in ref. [7] on the same dataset. Note that the time-out for all three approaches was set to three hours per repair attempt. According to the experiments by Martinez et al. [7], a larger time-out would not improve their effectiveness. The implemented approaches generally need less than 1.5 hours to find the first test-suite adequate patch (with very few exceptions) and we use a CPU environment similar to the one used in ref. [7], so the comparison here is unlikely to favor our implemented approaches.

Table 10 summarizes the results of the ten repair approaches on 224 bugs considered in Defects4J. For each approach, we list every bug for which at least one test-suite adequate patch is found. From Table 10, ARJA is able to fix the highest number of bugs among all the ten approaches, with a total of 59, which accounts for 26.3% of all the bugs considered. To our knowledge, none of the existing

```
1  // DateTimeZone.java
2  public long adjustOffset(long instant,
       boolean earlierOrLater) {
3  ...
4  +  instantAfter =
5  +     FieldUtils.safeAdd(instantAfter,
6  +     ((long) hashCode()) * ((long)
7  +        DateTimeConstants.MINUTES_PER_DAY));
8     return convertLocalToUTC(local, false,
9        earlierOrLater ? instantAfter :
10          instantBefore);
11 }
```

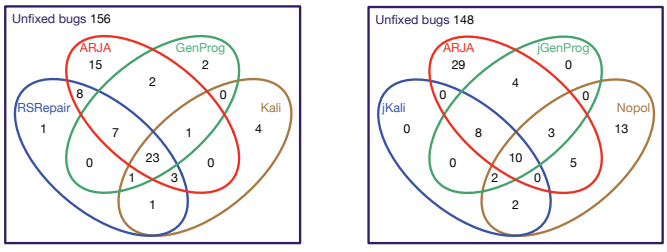
Fig. 15. Test-suite adequate patch generated by ARJA$_b$ for the bug T17.

repair approaches in the literature can synthesize test-suite adequate patches for so many bugs on the same dataset.

Although each of the three ARJA variants (using type matching) searches over a superset of ARJA's ingredient space, they do not repair a higher number of bugs compared to ARJA. The possible reason is that the search ability of GP is still not strong enough to handle such a large search space determined by type matching, which has also been mentioned in Section 5.5. However, we note that the three ARJA variants can fix a few bugs that cannot be patched by any redundancy-based repair approach (including ARJA) in comparison. These bugs are T1, L13, L14 and M79 by ARJA$_v$; L58, M44 and M79 by ARJA$_m$; and T1, T17, L13, L14, L21 and M7 by ARJA$_b$. This demonstrates the effectiveness of type matching on some real-world bugs. For instance, only ARJA$_b$ synthesizes a test-suite adequate patch for T17, which is shown in Fig. 15. The statement inserted (lines 4–7 in Fig. 15) indeed does not exist in the buggy program considered, whereas the following statement does

```
minutes = FieldUtils.safeAdd(minutes,
    ((long) getDays()) * ((long)
    DateTimeConstants.MINUTES_PER_DAY));
```

But the variable `minutes` and the method `getDays()` in this statement are both outside the scope of the faulty location. ARJA$_b$ maps the variable `minutes` to `instantAfter` and the method invocation `getDays()` to `hashCode()` through type matching, thereby inventing a new statement. ARJA$_b$ exploits GP to insert this new statement before line 8, which allows the patched program to fulfill the given test suite.



(a) ARJA, GenProg, RSRepair and Kali

(b) ARJA, jGenProg, jKali and Nopol

Fig. 16. Venn diagram of bugs for which test-suite adequate patches are found.

Another three repair approaches implemented by ourselves (i.e., GenProg, RSRepair and Kali) fix 36, 44 and 33 bugs respectively. To show their performance difference with ARJA more clearly, Fig. 16(a) presents a Venn diagram that shows the intersections of the fixed bugs among the four approaches. As seen from Fig. 16(a), the overwhelming majority of bugs handled by GenProg, RSRepair and Kali can also be handled by ARJA; ARJA is able to fix 15 bugs that neither GenProg, RSRepair nor Kali could fix; for 23 bugs, all the four repair approaches can generate a test-suite adequate patch. Note that on the Defects4J dataset considered, RSRepair can synthesize test-suite adequate patches for more bugs than GenProg. Further, we find that RSRepair generates a patch containing only a single edit for 41 out of 44 fixed bugs. Recall that the search mechanism of RSRepair is more suitable to find such very simple repairs than that of GenProg, as discussed in Section 5.2. So, it is not surprising that RSRepair shows a certain advantage over GenProg on the bugs considered here.

jGenProg, jKali and Nopol can synthesize test-suite adequate patches for 27, 22 and 35 bugs respectively. Fig. 16(b) compares ARJA with the three approaches using the Venn diagram. ARJA is clearly superior to jGenProg and jKali. Almost all the bugs repaired by jGenProg and jKali can also be repaired by ARJA. ARJA also fixes a greater number of bugs than Nopol, but their performance shows good complementarity: ARJA and Nopol can handle 41 and 17 bugs, respectively, that cannot be handled by the peer.

In addition, our implemented GenProg and Kali show better overall performance than jGenProg or jKali. The reason may be due to different parameter settings and implementation methods.

To summarize, ARJA and its three variants have an obvious advantage over the other state-of-the-art repair approaches in handling real-world bugs. We find that the ten repair approaches under consideration in our experiment can synthesize a test-suite adequate patch for 88 out of total 224 bugs. To provide a reference for future research, the patches generated by our approaches are publicly available at GitHub.[19]

5.7 Patch Correctness (RQ7)

Because of a weak test suite used as oracle, a test-suite adequate patch for certain bugs may be incorrect though passing the given test suite, a condition called patch overfitting [13], [52]. In this subsection, we manually evaluate the correctness of the patches generated by ARJA.[20] We regard a test-suite adequate patch correct if it is exactly the same as or semantically equivalent to a human-written patch. Due to a lack of domain knowledge, we cannot confirm the correctness of the test-suite adequate patches for some bugs, and do not include them in our manual assessment.

After careful analysis, we find that ARJA can synthesize correct patches for at least 18 bugs in Defects4J, which are shown in Table 11. These results are very encouraging. This is because ARJA referred to here is also based on the redundancy assumption like jGenProg, but jGenProg can only correctly fix 5 bugs (as reported in [7]) which are also correctly repaired by ARJA. Among the remaining 13

19. Defects4J patches, http://github.com/yyxhdy/defects4j-patches
20. Due to limited manual effort, we currently do not consider the correctness of the patches found by the other approaches in comparison

TABLE 11
The bugs for which the correct patches are synthesized by ARJA

Project	Bug ID
JFreeChart	C3, C5, C12
Joda-Time	T15
Commons Lang	L20, L35, L43, L45
Commons Math	M5, M22, M39, M50, M53, M58, M70, M73, M86, M98
Total	18

bugs, jGenProg cannot even find a test-suite adequate patch for 11 of them. This again demonstrates the effectiveness of the improved GP search in ARJA. Furthermore, to our knowledge, only ARJA can generate a correct patch for bugs C12, L20, M22, M39, M58, M86 and M98, whereas the other repair systems in the literature cannot. Another highlight of ARJA is that it can correctly fix some multi-location bugs in Defects4J, which are hard to repair by the other repair methods.

To illustrate the expressive power, we conduct the case studies of the bugs that are correctly repaired by ARJA. We find that some of these repairs appear to be very interesting.

5.7.1 Case Study of Single-Location Bugs that are Correctly Repaired by ARJA

Among the bugs correctly fixed by ARJA, 13 can be categorized as single-location bugs since ARJA is able to repair them with only a single edit. Here we only take M58 and M86 as examples.

Fig. 17 shows the correct patch generated by ARJA for M58. It is syntactically different from the human-written patch that replaces the faulty statement (lines 6–7) with `return fit(guess);`. Nevertheless, the method parameter in the statement inserted by ARJA (lines 8–10) is indeed equivalent to the variable `guess` according to the variable declaration statement (lines 3–5), and we have confirmed that the method invocations `ParameterGuesser`, `getObservations` and `guess` do not change anything outside the faulty method `fit()`. Thus the patch shown in Fig. 17 is semantically equivalent to the human-written patch.

```
1  // GaussianFitter.java
2  public double[] fit() {
3    final double[] guess = (new
4      ParameterGuesser(getObservations()))
5        .guess();
6  - return fit(new Gaussian.Parametric(),
7  -      guess);
8  + return fit((new
9  +    ParameterGuesser(getObservations()))
10 +      .guess());
11 }
```

Fig. 17. Correct patch generated by ARJA for bug M58.

In Fig. 18, we show the correct patch found by ARJA for the bug M86. This bug occurs because the buggy program fails to correctly check whether a symmetric matrix is positive definite (the Cholesky decomposition only applies to

```
1  // CholeskyDecompositionImpl.java
2  public CholeskyDecompositionImpl(...) {
3    for (int i = 0; i < order; ++i) {
4      final double[] lI = lTData[i];
5      if (lTData[i][i] <
6        absolutePositivityThreshold) {
7        throw new
8          NotPositiveDefiniteMatrixException();
9      }
10     ...
11   }
12   for (int i = 0; i < order; ++i) {
13     final double[] ltI = lTData[i];
14 +   if (lTData[i][i] <
15 +     absolutePositivityThreshold) {
16 +     throw new
17 +       NotPositiveDefiniteMatrixException();
18 +   }
19     ...
20   }
21  ...
22 }
```

Fig. 18. Correct patch generated by ARJA for the bug M86.

the positive-definite matrix). The buggy program does such a check using the `if` statement at line 5 in Fig. 18, which examines whether all diagonal elements are positive. However this is only a necessary, though not sufficient, condition for the positive definite matrix. The human-written patch first deletes the `if` statement at line 5 and then inserts almost the same `if` statement (`lTData[i][i]` is equivalently changed to `ltI[i]`) before line 19, so that the validation of the positive definitiveness is conducted correctly during the decomposition process. Unlike the human-written patch, the correct patch by ARJA does not delete the `if` statement at line 5. Because this `if` statement states a necessary condition, just keeping it intact would not influence the correctness of the patched program.

5.7.2 Case Study of Multi-Location Bugs that are Correctly Repaired by ARJA

The bugs L20, L35, T15, M22 and M98 are classified as multi-location bugs, since ARJA fixes each of them correctly using more than one edit. For M22 and M98, ARJA can generate a correct patch that is exactly the same as the human-written patch. As for the remaining three, ARJA synthesizes semantically equivalent patches, which are analyzed as follows. In Fig. 19, we present a correct patch synthesized by ARJA for bug L20. The reason leading to this bug is that even if `array[startIndex]` is not equal to `null`, `array[startIndex].toString()` can still be `null`, and `array[startIndex].toString().length()` would thereby cause the `NullPointerException`. The human developer fixes this bug by replacing two faulty statements (lines 4–7 and lines 14–18 in Fig. 19) with the same statement shown as follows:

```
StringBuilder buf = new
    StringBuilder(noOfItems * 16);
```

We find that the initial capacity (e.g., `noOfItems * 16` or 256) of `StringBuilder` has nothing to do with the correct-

```
// StringUtils.java
public static String join(Object[] array,
    char separator, int startIndex, int
    endIndex) {
  ...
- StringBuilder buf = new
-     StringBuilder((array[startIndex] ==
-       null ? 16 : array[startIndex]
-         .toString().length()) + 1);
+ StringBuilder buf = new
+     StringBuilder(256);
  ...
}
public static String join(Object[] array,
    String separator, int startIndex, int
    endIndex) {
  ...
- StringBuilder buf = new
-     StringBuilder((array[startIndex] ==
-       null ? 16 : array[startIndex]
-         .toString().length()) +
-          separator.length());
+ StringBuilder buf = new
+     StringBuilder(256);
  ...
}
```

Fig. 19. Correct patch generated by ARJA for bug L20.

ness but with the performance. If the the initial capacity is too large, much unnecessary memory will be allocated; if it is too small, `StringBuilder` will frequently expand its capacity when accommodating additions, requiring more computational time. But in terms of making the buggy program functionally correct, the patch generated by ARJA has no difference with the human-written patch.

Fig. 20 shows the correct patch found by ARJA for bug L35. The buggy program fails to satisfy a desired behavior: the method `add(T[] array, T element)` should throw `IllegalArgumentException` when both parameters are `null`. The only difference between the patch by ARJA and the human-written patch lies in the detailed message of the exception. However, this difference will not affect the ability of the patched program by ARJA to meet the specified functionality successfully.

ARJA fixes bug T15 correctly in an interesting way as shown in Fig. 21. This bug occurs when `val1` and `val2` are equal to `Long.MIN_VALUE` and -1 respectively. In this scenario, the product should be `-Long.MIN_VALUE`. But `-Long.MIN_VALUE` exceeds the maximum allowed value (i.e., `Long.MAX_VALUE`) and the buggy program indeed returns an incorrect value due to overflow. To fix this bug, the human developer just inserts the following `if` statement before line 5 in Fig. 21:

```
if (val1 == Long.MIN_VALUE) {
  throw new
      ArithmeticException("...overflows");
}
```

So in term of the human-written patch, this bug can also be regarded as a single-location bug. However, ARJA cannot generate such a patch since the above `if` statement does

```
// ArrayUtils.java
public static <T> T[] add(T[] array, T
    element) {
...
-  type = Object.class;
+  throw new IllegalArgumentException
+    ("The Integer did not match any ...");
...
}
public static <T> T[] add(T[] array, int
    index, T element) {
...
-  return (T[]) new Object[] { null };
+  throw new IllegalArgumentException
+    ("The Integer did not match any ...");
...
}
```

Fig. 20. Correct patch generated by ARJA for bug L35.

not exist anywhere in the buggy program. As shown in Fig. 21, the patch by ARJA first replaces line 5 with a `break` statement to avoid returning an incorrect value there, then it detects the overflow that triggers the bug in the new `if` statement (lines 15–20) with the expression `val1 == Long.MIN_VALUE && val2 == -1`. Note that the boolean expression `val2 == Long.MIN_VALUE && val1 == -1` is always false since `val2` is an `int` value, so it has no effect and can be ignored. As can be seen, the patch by ARJA just does the same thing as the human-written patch in a different way, and thus it is correct.

```
// FieldUtils.java
public static long safeMultiply(long val1,
    int val2) {
  switch (val2) {
  case -1:
- return -val1;
+ break;
  case 0: return 0L;
  case 1: return val1;
  }
  long total = val1 * val2;
- if (total / val2 != val1) {
-   throw new
-     ArithmeticException("...overflows");
- }
+ if (total / val2 != val1 || val1 ==
+   Long.MIN_VALUE && val2 == -1 || val2 ==
+    Long.MIN_VALUE && val1 == -1) {
+   throw new
+    ArithmeticException("...overflows");
+ }
  return total;
}
```

Fig. 21. Correct patch generated by ARJA for bug T15.

5.7.3 Other Findings

Our manual study also provides other findings besides the correct patches for some bugs.

We find that although the test-suite adequate patches for a number of bugs (e.g., C1, C19, L7 and L16) may not be correct, they present some similarities with corresponding human-written patches. So these test-suite adequate patches would still be useful in assisting the human developer to create correct patches.

With stronger search ability, ARJA can identify more under-specified bugs than previous repair approaches (e.g., jGenProg and jKali). For example, Martinez et al. [7] claimed that the specification of the bug L55 by the test suite is good enough to drive the generation of a correct patch, considering Nopol can repair this bug whereas jGenProg and jKali cannot. However we find that L55 is also an under-specified bug and an overfitting patch (shown in Fig. 22) that simply deletes two statements can fulfill its test suite.

```
1 // StopWatch.java
2 public void stop() {
3 ...
4 -  stopTime = System.currentTimeMillis();
5 -  this.runningState = STATE_STOPPED;
6 }
```

Fig. 22. Test-suite adequate but incorect patch for bug L55.

Moreover, we have also checked the correctness of the patches by ARJA for seeded bugs. We find that most of these test-suite adequate patches are correct. Recalling that our implemented Kali cannot uncover the weakness of the test suite for any seeded bug, this implies that a stronger test suite would render ARJA more able to generate correct patches. Several recent studies [47], [48], [53] have started to explore the potential of test case augmentation for program repair.

5.7.4 Summary

In summary, through careful manual assessment, we find that ARJA can synthesize a correct patch for at least 18 bugs in Defects4J. To the best of our knowledge, some of the 18 bugs have never been fixed correctly by existing repair systems in the literature. Furthermore, ARJA is able to generate correct patches for several multi-location bugs, which is impossible for most of the other repair approaches.

Note that we do not focus on the number of correctly repaired bugs on the Defects4J dataset when comparing ARJA with other approaches that are *not* based on the redundancy assumption (e.g., Nopol). Nowadays, it is common knowledge that different kinds of repair techniques can be better at addressing different classes of bugs. For example, although Nopol that targets conditional statement bugs can only fix 5 bugs correctly on the same dataset [7], 3 of them cannot be repaired correctly by ARJA. So the number of correct fixes by different categories of repair techniques would strongly depend on how the dataset tested was built [14]. Also, we cannot expect that the 224 bugs in Defects4J can truly reflect the natural distribution of real-world bugs. For instance, Defects4J indeed contains a considerable number of null pointer bugs, so it may favor those approaches that can explicitly conduct null pointer detection with fix templates (e.g., PAR [54] and ACS [46]). Compared to such approaches, ARJA is a more generic repair approach.

In contrast, ARJA and jGenProg can be compared meaningfully on the Defects4J dataset in terms of the number of bugs fixed or correctly fixed, because both of them typically belong to redundancy-based repair techniques and use GP to explore the search space. ARJA performs much better than jGenProg, which clearly validates the improvement of GP search in ARJA.

To facilitate re-examination by the other researchers, we provide a detailed explanation of the correctness for each correct patch generated by ARJA, publicly available at GitHub.[21]

5.8 Reasons for Failure (RQ8)

As seen from the experimental results, ARJA and its variants sometimes fail to find a test-suite adequate patch for some bugs. We find that there are three possible reasons for failure, which are discussed in the following.

The first reason is that fix ingredients for the bug do not exist in the search space of the proposed repair approach. In this case, no matter how powerful the underlying genetic search is, ARJA (or its variants) will definitely fail to fix the bug. The failure of ARJA on H1–H3 and H5 (see Table 9) can be attributed to this reason.

The second reason is that although test-suite adequate (or even correct) patches exist in the search space, the search ability of GP is still not strong enough to find it within the required number of generations. An example is that ARJA fails on bug F9 in 24 out of 30 trials. Another example is the failure of ARJA on the real-world bug L53. Fig. 23 shows a human-written patch for this bug. We find that ARJA takes into account lines 11 and 19 as potential faulty lines by fault localization. So, a correct patch within the search space of ARJA consists of the following two edits: 1) insert the `if` statement located at line 7 before line 11; 2) insert the `if` statement located at line 15 before line 19. This patch is semantically equivalent to the human-written patch shown in Fig. 23. However, we note ARJA fails to find it under the parameter settings of our experiment. This may be because the genetic search is easily trapped in local optima when navigating the search space corresponding to L53.

The last reason is that ARJA fails to consider the faulty lines that trigger the bug in to computational search, which can be further divided into the following three categories:

1) The fault localization technique adopted in ARJA fails to identify all faulty lines related to the bug of interest. This applies to bugs C2, T9, L4, M12, M104, and so on.
2) Due to the inadequate accuracy of fault localization, the faulty lines are given relatively low suspiciousness. For example, Fig. 24 shows the human-written patch for L10. We find that the suspiciousness for all these faulty lines is less than 0.2, but the number of lines with suspiciousness larger than 0.2 is more than 40 (i.e., $n_{\max}$ value in our experiments). Hence ARJA leaves out faulty lines and fails to fix this bug.
3) The test suite is not adequate for the fault localization. Bug M46 provides an example of such a scenario, whose human written patch is shown in Fig. 25. We find that the whole method starting from line 10 is not

21. Correctness, http://github.com/yyxhdy/defects4j-correctness

```
// DateUtils.java
private static void modify(Calendar val,
    int field, boolean round) {
  ...
  if (!round || millisecs < 500) {
    time = time - millisecs;
+ }
    if (field == Calendar.SECOND) {
      done = true;
    }
- }
  int seconds = val.get(Calendar.SECOND);
  if (!done && (!round || seconds < 30)) {
    time = time - (seconds * 1000L);
+ }
    if (field == Calendar.MINUTE) {
      done = true;
    }
- }
  int minutes = val.get(Calendar.MINUTE);
  ...
}
```

Fig. 23. Human-written patch for bug L53.

covered by any negative test. So based on the current test suite of M46, the coverage-based fault localization technique, no matter how powerful, cannot identify line 13 as a potential faulty line.

```
// FastDateParser.java
private static StringBuilder
    escapeRegex(StringBuilder regex,...) {
- boolean wasWhite = false;
    ...
-   if(Character.isWhitespace(c)) {
-     if(!wasWhite) {
-       wasWhite = true;
-       regex.append("\\s*+");
-     }
-     continue;
-   }
-   wasWhite = false;
    ...
}
```

Fig. 24. Human-written patch for bug L10.

Note that we can use simple strategies to alleviate the issues mentioned in the second and the third categories. For example, for the bug L10, we just reset the parameter $n_{\max}$ to 80 and then run ARJA again. As a result, ARJA can now find a test-suite adequate patch for L10 which simply deletes the `if` statement at line 5 in Fig. 24. This patch is semantically equivalent to the human written patch and is thus correct. As for M46, we modify the JUnit test named `testScalarDivideZero` as shown in Fig. 26. This JUnit test (before modification) is originally a positive test, because `x.divide(Complex.ZERO)` and `x.divide(0)` return the same incorrect value (i.e., `INF`) due to the faults in lines 5 and 13 in Fig. 25, and because this test only checks the equality rather than the individual values. We add a statement (lines 4–5 in Fig. 26) that can expose the fault

```
// Complex.java
public Complex divide(Complex divisor)
    throws NullArgumentException {
  ...
  if (divisor.isZero) {
-   return isZero ? NaN : INF;
+   return NaN;
  }
  ...
}
public Complex divide(double divisor) {
  ...
  if (divisor == 0d) {
-   return isZero ? NaN : INF;
+   return NaN;
  }
  ...
}
```

Fig. 25. Human-written patch for bug M46.

at line 13 in Fig. 25, and run ARJA again on M46 with the modified test suite. As a result, ARJA can now fix this multi-location bug and the patch obtained is exactly the same as the human-written patch.

```
// ComplexTest.java
public void testScalarDivideZero() {
  Complex x = new Complex(1,1);
+ TestUtils.assertEquals(x.divide(0),
+     Complex.NaN, 0);
  TestUtils.assertEquals(x.divide(
    Complex.ZERO), x.divide(0), 0);
}
```

Fig. 26. The modification of an associated JUnit test of M46.

5.9 Repair Efficiency (RQ9)

For industrial applicability of automated program repair, an approach should fix a bug within a reasonable amount of time. Table 12 presents time (in minutes) of generating the first repair for the Defects4J bugs. Note that although jGenProg, jKali and Nopol were executed by Martinez et al. [7] on a machine different from ours, we can roughly compare their time cost with that of our implementation since we use a similar CPU environment.

As seen from Table 12, the median and average time for a successful repair by ARJA and its three variants runs for around 5 minutes and 10 minutes, respectively. However, maximum CPU time can reach more than one hour. GenProg, RSRepair and Nopol are less efficient than ARJA and its variants, which further verifies the superiority of the ARJA framework. Kali is the most efficient on average but it only considers trivial patches. In addition, our implementation of GenProg and Kali show clear advantage over jGenProg and jKali in terms of time cost possibly attributable to the test filtering procedure or different implementation methods.

In summary, the repair efficiency of ARJA and its variants compares favorably with that of existing notable auto-

TABLE 12
Time cost of patch generation on Defects4J dataset

Repair Approach	CPU Time (in minutes)			
	Min	Median	Max	Average
Arja[1]	0.73	4.70	63.73	10.02
Arja_v[1]	0.86	4.63	80.63	10.32
Arja_m[1]	0.86	4.67	37.85	10.49
Arja_b[1]	0.89	5.27	95.12	11.48
GenProg[1]	0.61	8.43	83.06	16.20
RSRepair[1]	0.87	6.23	238.93	17.88
Kali[1]	0.89	2.38	48.05	6.58
jGenProg[2]	0.67	61	76	55.83
jKali[2]	0.6	18.75	87	23.55
Nopol[2]	0.52	22.5	114	30.88

[1] The CPU time on an Intel Xeon E5-2680 V4 2.4 GHz processor with 20 GB memory.
[2] The CPU time on an Intel Xeon X3440 2.53 GHz processor with 15 GB memory. The results are excerpted from [7].

matic repair approaches. Considering that ARJA consumes about 10 minutes on average and one hour at most for a repair in our experiments, we think that this efficiency is generally acceptable for industrial use in light of the bug-fixing time required by human programmers [55], [56].

6 Threats to Validity

In this section, we discuss three basic types of threats that can affect the validity of our empirical studies.

6.1 Internal Validity

To support a more reasonable comparison, we reimplemented GenProg, RSRepair and Kali for Java under the same infrastructure as ARJA. Although we faithfully and carefully followed the details described in the corresponding research papers during reimplementation, our implementation may still not perform as well as the original systems. There may even be bugs in the implemented systems that we have not found yet. To mitigate this threat, we make the code of the three systems available on GitHub for peer-review. Note that although jGenProg has been widely used as the the implementation of GenProg for Java [45]–[48], it was not implemented by the original authors of GenProg, thereby also potentially suffering from reimplementation issues. According to our results, our implemented GenProg indeed shows advantages over jGenProg.

In our experiments, we use the same parameter setting for all bugs considered. There is a risk that the parameter setting is poor for handling some bugs. Section 5.8 has shown an example where resetting of $n_{\max}$ can allow ARJA find a correct patch for bug L10. However it is not realistic to select an ideal parameter setting for every repair approach on every bug. We here use a uniform parameter setting among tghe implemented repair approaches to ensure a fair comparison.

Another internal threat to validity concerns the stochastic nature of ARJA (including its variants), GenProg and RSRepair. It is possible that different runs of these repair approaches would obtain somewhat different results. We run each of them only once on each of 224 bugs in Defects4J, which may lead to an overestimation or underestimation of their repair effectiveness on this dataset. However, our experiments on the Defects4J dataset have already been much larger in scale than those conducted by Martinez et al. [7], since it involved a larger number of repair methods (i.e., 7 methods) and repair trials (i.e., 1,568 trials).

6.2 External Validity

Our experimental results are based on both seeded bugs and real-world bugs.

Although the seeded bugs F1–F15 are randomly produced, there still exists the possibility that the fitness landscapes corresponding to these bugs favor a certain kind of search mechanism. So the evaluation on these bugs may not reflect the actual difference in search ability between different search strategies (i.e., multi-objective GP, single-objective GP and random search).

For real-world bugs, we used 224 bugs of four Java projects from Defects4J. However, it is not possible to expect that such a number of bugs can fully represent the actual distribution of defect classes and difficulties in the real world. So the evaluation may not be adequate enough to reflect the actual effectiveness of our repair techniques on real-world bugs, and our results may also not generalize to other datasets. However, Defects4J is known to be the most comprehensive dataset of real Java bugs currently available, and it has been extensively used as the benchmark for evaluating Java program repair systems [7], [45]–[48].

6.3 Construct Validity

Here we manually analyze the correctness of the test-suite adequate patches generated by ARJA. Such a manual study is not scientifically sound, although it has been an accepted practice [7], [13] in automated program repair. It may happen that the analysts classify an incorrect patch as correct due to a limited understanding of the buggy program. For example, Martinez et al. [7] claimed that Nopol can synthesize correct patches for bugs C5 and M50, but a recent study [47] indicated that the generated patches are not really correct. To reduce this threat, we carefully rechecked the correctness of the identified correct patches and made the explanation,. why we believe they are correct, available online.

ARJA finishes a repair trial for a bug only when the maximum number of generations is reached. So ARJA may finally return multiple test-suite adequate patches (with the same patch size) for a bug. In such a case, we examine the correctness of all the patches obtained, and we deem ARJA to be able to fix this bug correctly if at least one of these patches is identified as correct. We confirm that ARJA outputs both correct and incorrect (i.e., only test-suite adequate) patches for bugs C12, L43 and M22. So a strict criterion for correctness would pose a threat to the validity of the number of correct repairs by ARJA. However, we think it is unrealistic to completely avoid the generation of incorrect but test-suite adequate patches when program repair is just based on a weak test suite. Possibly machine learning techniques [57] can be used to estimate the probability of a

patch being correct, but this falls outside the scope of this paper. Further, sometimes it is indeed very difficult to differentiate the incorrect patches from correct ones. For example, Fig. 27 shows a test-suite adequate patch found by ARJA for bug L43. This patch is indeed incorrect but it is very similar to the human-written patch that inserts `next(pos);` before line 6 rather than line 5. Even an experienced human programmer cannot easily recognize it as incorrect without a deep understanding of the program specification. We still think test-suite augmentation is a fundamental solution to the patch overfitting problem.

```
1 // ExtendedMessageFormat.java
2  private StringBuffer
       appendQuotedString(...) {
3 ...
4 + next(pos);
5    if (escapingOn && c[start] == QUOTE) {
6      return appendTo == null ? null :
7          appendTo.append(QUOTE);
8    }
9 }
```

Fig. 27. Test-suite adequate patch found by ARJA for bug L43.

Different from ARJA, jGenProg is terminated once the first test-suite adequate patch is found, and the analysis of correctness only targets this patch. This may lead to a bias toward ARJA when comparing its number of correct fixes with that of jGenProg. jGenProg may still have had the chance to find a correct patch for certain bugs, had it not been terminated immediately. However, we think such bias is minimal: Except for 5 bugs correctly fixed by both ARJA and jGenProg, jGenProg could not produce any patch for 11 out of the remaining 13 bugs that are correctly repaired by ARJA.

7 Related Work

The test-suite based program repair techniques can be roughly divided into two main branches: search-based repair approaches and semantics-based repair approaches. In this section, we first list related studies about these two categories of approaches, then review the research on empirical aspects of test-suite based repair.

7.1 Search-Based Repair Approaches

Search-based repair approaches generally determine a search space that potentially contains correct repairs, and then apply computational search techniques, such as a GA or random search, to navigate the search space, in order to find test-suite adequate patches.

JAFF. Arcuri and Yao [58] proposed the idea of using GP to co-evolve the programs and unit tests, in order to fix a bug automatically. Subsequently, Arcuri [59] developed a research prototype, called JAFF, which models bug fixing as a search problem and uses EAs to solve it. JAFF can only handle a subset of Java and was evaluated on toy programs.

GenProg. GenProg is a prominent GP based program repair system which was developed jointly by several researchers [4], [8], [9], [32]. Le Goues et al. [9] presented the latest GenProg implementation, where the patch representation is used instead of the AST based representation [4], [8], [32] in order to make GenProg scalable to large-scale programs. It was reported in [9] that GenProg can automatically repair 55 out of 105 bugs from 8 open-source programs. Since our study is based on GenProg, a more detailed description was given in Section 2.2.

Around the GenProg framework, a number of related studies have been conducted in the literature. Fast et al. [37] investigated two approaches (i.e., test suite sampling and dynamic predicates) to enhancing fitness functions in GenProg. Schulte et al. [60] applied GenPog to fix bugs in x86 assembly and Java bytecode programs. Le Goues et al. [34] investigated the choices of solution representation and genetic operators for the underlying GP in GenProg. Oliveira et al. [35] presented a low-granularity patch representation and developed several crossover operators associated with this representation. Tan et al. [38] suggested a set of anti-patterns to inhibit GenProg or the other search-based methods from generating nonsensical patches.

Mutation-based repair. Debroy and Wong [61] proposed to combine standard mutation operators (from the mutation testing literature) and fault localization to fix bugs. This method considers each possibly faulty location one by one according to the suspiciousness metric, and mutates the statement at the current location to produce potential fixes.

PAR. Kim et al. [54] proposed PAR, which leverages fix patterns manually learned from human written patches. Similar to GenProg, PAR also implements an evolutionary computing process. But instead of using crossover and mutation operators as in GenProg, PAR uses fix templates derived from common fix patterns to produce new program variants in each generation. Experiments on six Java projects and a user study confirm that the patches generated by PAR are often more meaningful than those by GenProg.

AE. Weimer et al. [62] proposed a deterministic repair algorithm based on program equivalence, called AE. This algorithm uses adaptive search strategies to control the order in which candidate repairs and test cases are considered. Empirical evaluations showed that, AE can reduce the search space by an order of magnitude when compared to GenProg.

RSRepair. Qi et al. [12] presented RSRepair, which replaces the evolutionary search in GenProg with random search. Their experiments on 24 bugs of the GenProg benchmark suite indicate that random search performs more effectively and efficiently than GP in program repair.

SPR. Long and Rinard [63] reported SPR, which adopts a staged program repair strategy to navigate a rich search space of candidate patches efficiently. SPR defines a set of transformation schemas beforehand and uses the target value search or condition synthesis algorithm to determine the parameter values of the selected transformation schema. Experimental results on 69 bugs from 8 open source applications indicate that SPR can generate more correct patches than previous repair systems.

Prophet. Based on SPR, Long and Rinard [57] further designed Prophet, a repair system using a probabilistic model to rank candidate patches in the search space of SPR. Given a training set of successful human patches, Prophet learns model parameters via maximum likelihood estima-

tion. Experimental results indicate that a learned model in Prophet can significantly improve its ability to generate correct patches.

HistoricalFix. Le et al. [45] introduced a repair method which evolves patches based on bug fix patterns mined from the history of many projects. This method uses 12 kinds of existing mutation operators to generate candidate patches and determines the fitness of a patch by assessing its similarity to the mined bug-fixing patches. Experimental results on 90 bugs from Defects4J show that the proposed method can produce good-quality fixes for more bugs compared to GenProg and PAR.

ACS. Xiong et al. [46] reported ACS, a repair system targeting `if` condition bugs. ACS decomposes the condition synthesis into two steps: variable selection and predicate selection. Based on the decomposition, it uses dependency-based ordering along with the information of javadoc comments to rank the variables, and uses predicate mining to rank predicates. With a synthesized condition, ACS leverages three fix templates to generate a patch. Experiments show that ACS can successfully repair 18 bugs from four projects of Defects4J.

Genesis. Long et al. [64] presented a repair system, called Genesis, which can automatically infer code transforms and search spaces for patch generation. Genesis was tested on two classes of errors (i.e., null pointer errors and out of bounds errors) in real-world Java programs.

7.2 Semantics-Based Repair Approaches

Typically, the semantics-based repair approaches infer semantic constraints from the given test cases, and then generate the test-suite adequate patch through solving the resulting constraint satisfaction problem, particularly the SMT problem.

SemFix. Nguyen et al. [15] proposed SemFix, a pioneering tool for semantic-based program repair. SemFix first employs statistical fault localization to identify likely-buggy statements. Then, for each identified statement, it generates repair constraints through symbolic execution of the given test suite and solves the resulting constraints by an SMT solver. SemFix targets faulty locations that are either a right hand side of an assignment or a branch predicate, and was compared to GP based methods on programs with seeded as well as real bugs.

SearchRepair. Ke et al. [16] developed a repair method based on semantic code search, called SearchRepair. This method encodes a database of human-written code fragments as SMT constraints on the input-output behavior and searches the database for potential fixes with an input-output specification. Experiments on small C programs written by students showed that SearchRepair can generate higher-quality repairs than GenProg, AE and RSRepair.

DirectFix. Mechtaev et al. [36] implemented a prototype system, called DirectFix, for automatic program repair. To consider the simplicity of repairs, DirectFix integrates fault localization and patch generation into a single step by leveraging partial maximum SMT constraint solving and component-based program synthesis. Experimental comparison indicates that the patches found by DirectFix are simpler and safer than those by SemFix.

QLOSE. D'Antoni et al. [65] formulated the quantitative program repair problem, where the optimal repair is obtained by minimizing an objective function combing several syntactic and semantic distances to the buggy program. The problem is an instance of maximum SMT problem and is solved by an existing program synthesizer. The technique was implemented in a prototype tool called QLOSE and was evaluated on programs taken from educational tools.

Angelix, JFix. Mechtaev et al. [66] presented Angelix, a semantic-based repair method that is more scalable than SemFix and DirectFix. The scalability of Angelix attributes to the new lightweight repair constraint (called an angelic forest), which is independent of the size of the program under repair. Experimental studies on the GenProg benchmark indicate that Angelix has the ability to fix bugs from large-scale software and multi-location bugs. Angelix was originally designed for C programs. Recently, Le et al. [67] developed JFix which is an extension of Angelix for Java.

Nopol. Xuan et al. [42] proposed an approach, called Nopol, for automatic repair of buggy conditional statements in Java programs. Nopol employs angelic fix localization to identify potential fix locations and expected values of `if` conditions. For each identified location, it encodes the test execution traces as a SMT problem and converts the solution to this SMT into a patch for the buggy program. Nopol was evaluated on 22 bugs in real-world programs.

S3. Le et al. [68] presented S3 which leverages the programming-by-examples methodology to synthesize high-quality patches. S3 was evaluated on 52 bugs in small programs and 100 bugs in real-world large programs, and experimental results show that it can generate more high-quality repairs than several existing semantic-based repair methods.

7.3 Empirical Aspects of Test-Suite Based Repair

Besides proposing new program repair methods, there is another line of research that focuses on the empirical aspects of test-suite based repair.

Patch maintainability. Fry et al. [69] presented a human study of patch maintainability involving 150 participants and 32 real-world defects. Their results indicate that machine-generated patches are slightly less maintainable than human-written ones. Tao et al. [70] investigated an application scenario of automatic program repair where the auto-generated patches are used to aid the debugging process by humans.

Redundancy assumption. Martinez et al. [17] investigated all commits of 6 open-source Java projects experimentally, and found that a large portion of commits can be composed of what has already existed in previous commits, thereby validating the fundamental redundancy assumption of GenProg. In the same year, Barr et al. [18] inquired whether the redundancy assumption (or plastic surgery hypothesis) holds by examining 15,723 commits from 12 large Java projects. Their results show that 43% changes can be reconstituted from existing code, thus promising success to the repair methods that search for fix ingredients in the buggy program considered.

Patch overfitting. Qi et al. [13] analyzed the patches reported by three existing patch generation systems (i.e.,

GenProg, RSRepair, and AE), and found that most of these are not correct and are equivalent to a single functionality deletion, due to either the use of weak proxies or weak test suites. Based on this observation, they presented Kali which generates patches only by deleting functionality. Their experiments show that Kali can find at least as many plausible patches than three prior systems on the GenProg benchmark. Smith et al. [52] conducted a controlled empirical study of GenProg and RSRepair on the IntroClass benchmark. By using two test suites for each program (one for patch generation and another for evaluation), their experiments identified the circumstances under which patch overfitting happens. Long and Rinard. [71] analyzed the search spaces for patch generation systems. Their analysis indicates that correct patches occur sparsely within the search spaces and that plausible patches are relatively abundant compared to correct patches. They suggest using information other than the test suite to isolate correct patches. Yu et al. [47] investigated the feasibility and effectiveness of test case generation in addressing the overfitting problem. Their results indicate that test case generation is ineffective at promoting the generation of correct patches, thus calling for research on test case generation techniques tailored to program repair systems. Xin et al. [48] proposed a tool named DiffTGen, which could identify overfitting patches through test case generation. They also showed that a repair method configured with DiffTGen could avoid obtaining overfitting patches and potentially generate correct ones. Yang et al. [53] presented an overfitting patch detection framework which can filter out overfitting patches by enhancing existing test cases.

Analysis of real-world bug fixes. Martinez and Monperrus [39] proposed to mine repair actions from software repositories. Based on a fine-grain AST differencing tool, their work analyzed 62,179 versioning transactions extracted from 14 repositories of open-source Java software, in order to obtain the probability distributions over different repair actions. It was expected that such distributions can guide the search of repair methods. Zhong and Su [72] conducted a large-scale empirical investigation on over 9,000 bug fixes from 6 popular Java projects, then distilled several findings and insights that can help improve state-of-the-art repair methods. Soto et al. [73] presented a large-scale empirical study of bug fix commits in Java projects. Their work provided several insights about broad characteristics, fix patterns, and statement-level mutations in real-world bug fixes, motivating additional study of repair for Java.

Performance evaluation. Kong et al. [74] compared four program repair techniques on 153 bugs from 9 small to medium sized programs, and investigated the impacts of different programs and test suites on effectiveness and efficiency of the techniques in comparison. Martinez et al. [7] conducted a large-scale empirical evaluation of program repair methods on 224 bugs in Defects4J. Their experimental results showed that three considered methods (i.e., GenProg, Kali, and Nopol) can generate test-adequate patches for 47 bugs, among which 9 bugs were confirmed to be repaired correctly by manual. Le et al. [75] presented an empirical comparison of different synthesis engines for semantics-based repair approaches on IntroClass benchmark. Durieux et al. [76] reported the test-adequate patches obtained by Nopol on the bugs of Defects4J version 1.1.0.

Influence of fault localization. Qi et al. [77] evaluated the effectiveness of 15 popular fault localization techniques when plugged into GenPog. Their work claims that automated fault localization techniques need to be studied from the viewpoint of fully automated debugging. Assiri and Bieman [78] experimentally evaluated the impact of 10 fault location techniques on the effectiveness, performance, and repair correctness of a brute-force repair method. Wen et al. [79] conducted controlled experiments using the Defects4J dataset to investigate the influence of the fault space on a typical search-based repair approach (i.e., GenProg).

Datasets. Just et al. [19] presented Defects4J which is a bug database containing 357 real bugs from 5 real-world open-source Java projects. Le Goues et al. [80] designed two datasets (i.e., ManyBugs and IntroClass), which consist of 1,183 bugs in 15 C programs and support the comparative evaluation of repair algorithms for various of experimental questions. Tan et al. [81] presented a dataset, called Codeflaws, where all 3,902 defects contained from 7,436 C programs are classified into 39 defect classes. Berlin [82] collected a dataset called DBGBench, which consists of 27 real bugs in widely-used C programs and can serve as reality check for debugging and repair approaches.

8 Conclusion and Future Work

In this paper, we have proposed ARJA, a new GP based program repair approach for Java. Specifically, we present a lower-granularity patch representation which properly decouples the search subspaces of likely-buggy locations, operation types and ingredient statements, thereby enabling GP to traverse the search space more effectively. Based on this new representation, we propose to view automated program repair as a multi-objective optimization problem of minimizing the weighted failure rate and patch size simultaneously, and then use a multi-objective GA (i.e., NSGA-II) to search for simpler repairs. To speed up the fitness evaluation of GP, we present a test filtering procedure that can ignore a number of tests that are unrelated to the GP manipulations. Considering the characteristic of Java, we propose to restrict the replacement/insertion code to that which are not only in the variable scope but also in the method scope at the destination, so that the modified program is more likely to be compiled successfully. To further reduce the search space, we design three types of rules that are seamlessly integrated into three different phases (i.e., operation initialization, ingredient screening and solution decoding) of ARJA. In addition, we present a type matching strategy that can exploit the syntactic patterns of the statements that are out of scope at the destination so as to invent some new ingredient statements that are potentially useful. The type matching strategy can be optionally integrated into ARJA.

We conduct a large-scale experimental study on both seeded bugs and real-world bugs. The evaluation on seeded bugs clearly demonstrates the necessity and effectiveness of multi-objective GP used in ARJA, and also illustrates the strength of the type matching strategy. Furthermore, we evaluate ARJA and its three variants using type matching on 224 real-world bugs from Defects4J, in comparison with several state-of-the-art repair approaches. The comparison

results show that ARJA can generate a test-suite adequate patch for the highest number (i.e., 59) of real bugs, as opposed to only 27 by jGenProg and 35 by Nopol. The three ARJA variants can fix some bugs that cannot be fixed by ARJA, showing the potential of type matching on real-world bugs. Manual analysis confirms that ARJA can correctly fix 18 bugs at least in Defects4J, as opposed to 5 by jGenProg. To our knowledge, there are 7 among the 18 bugs that are repaired automatically and correctly for the first time. Another highlight is that ARJA can correctly repair several multi-location bugs that are widely recognized as hard to be repaired. Our study strongly suggests that the power of GP for program repair was far from being fully exploited in the past, and that GP can be expected to perform much better on this important and challenging task.

ARJA is publicly available at GitHub to facilitate further reproducible research on automated Java program repair: http://github.com/yyxhdy/arja.

In the future, we plan to incorporate a number of repair templates [54] into our ARJA framework so as to further enhance its performance on real-world bugs. Moreover, considering the mutational robustness [83] in software, we would like to combine the infrastructure of ARJA and the advanced many-objective GAs [84], [85] to improve the non-functional properties [26], [28] of Java software.

REFERENCES

[1] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[2] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys*, in press.

[3] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, in press.

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

[5] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[6] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 550–554.

[7] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[8] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 364–374.

[9] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 3–13.

[10] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[11] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic programming: an introduction*. Morgan Kaufmann San Francisco, 1998, vol. 1.

[12] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[13] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[14] M. Monperrus, "A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 234–242.

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

[16] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 295–306.

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

[18] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, 2014, pp. 306–317.

[19] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[21] M. F. Brameier and W. Banzhaf, *Linear genetic programming*. Springer Science & Business Media, 2007.

[22] W. B. Langdon, *Genetic programming and data structures: genetic programming+ data structures= automatic programming!* Springer Science & Business Media, 2012, vol. 1.

[23] M. Brameier and W. Banzhaf, "A comparison of linear genetic programming and neural networks in medical data mining," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 1, pp. 17–26, 2001.

[24] Y.-S. Lee and L.-I. Tong, "Forecasting energy consumption using a grey model improved by incorporating genetic programming," *Energy Conversion and Management*, vol. 52, no. 1, pp. 147–152, 2011.

[25] S. Nguyen, Y. Mei, and M. Zhang, "Genetic programming for production scheduling: a survey with a unified framework," *Complex & Intelligent Systems*, vol. 3, no. 1, pp. 41–66, 2017.

[26] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, 2015.

[27] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Specialising software for different downstream applications using genetic improvement and code transplantation," *IEEE Transactions on Software Engineering*, in press.

[28] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 4, pp. 515–538, 2011.

[29] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015, pp. 1375–1382.

[30] J. Petke, S. Haraldsson, M. Harman, D. White, J. Woodward *et al.*, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, in press.

[31] A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P. N. Suganthan, and Q. Zhang, "Multiobjective evolutionary algorithms: A survey of the state of the art," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 32–49, 2011.

[32] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation*, 2009, pp. 947–954.

[33] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, 2011, pp. 1427–1434.

[34] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, 2012, pp. 959–966.

[35] V. P. L. Oliveira, E. F. Souza, C. Le Goues, and C. G. Camilo-Junior, "Improved crossover operators for genetic programming for pro-

gram repair," in *Proceedings of the 8th International Symposium on Search Based Software Engineering*, 2016, pp. 112–127.

[36] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 448–458.

[37] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, 2010, pp. 965–972.

[38] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 24th International Symposium on Foundations of Software Engineering*, 2016, pp. 727–738.

[39] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Engineering*, vol. 20, no. 1, pp. 176–205, 2015.

[40] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, 2006, pp. 39–46.

[41] M. Martinez and M. Monperrus, "Astor: a program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[42] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[43] J. J. Durillo and A. J. Nebro, "jMetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760–771, 2011.

[44] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 378–381.

[45] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 213–224.

[46] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 416–426.

[47] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Test case generation for program repair: A study of feasibility and effectiveness," *arXiv preprint arXiv:1703.00198*, 2017.

[48] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th International Symposium on Software Testing and Analysis*, 2017, pp. 226–236.

[49] J. D. Knowles, R. A. Watson, and D. W. Corne, "Reducing local optima in single-objective problems by multi-objectivization," in *Proceedings of International Conference on Evolutionary Multi-Criterion Optimization*, 2001, pp. 269–283.

[50] M. T. Jensen, "Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation," *Journal of Mathematical Modelling and Algorithms*, vol. 3, no. 4, pp. 323–347, 2004.

[51] Y. Yuan and H. Xu, "Multiobjective flexible job shop scheduling using memetic algorithms," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 1, pp. 336–353, 2015.

[52] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.

[53] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 831–841.

[54] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 802–811.

[55] S. Kim and E. J. Whitehead Jr, "How long did it take to fix bugs?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, 2006, pp. 173–174.

[56] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the 4th International Workshop on Mining Software Repositories*, 2007, pp. 1–8.

[57] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

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

[59] A. Arcuri, "Evolutionary repair of faulty software," *Applied Soft Computing*, vol. 11, no. 4, pp. 3494–3514, 2011.

[60] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 313–316.

[61] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, 2010, pp. 65–74.

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

[63] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.

[64] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.

[65] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *Proceedings of the 28th International Conference on Computer Aided Verification*, 2016, pp. 383–401.

[66] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multi-line program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 691–701.

[67] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "Jfix: semantics-based repair of java programs via symbolic pathfinder," in *Proceedings of the 26th International Symposium on Software Testing and Analysis*, 2017, pp. 376–379.

[68] ——, "S3: syntax-and semantic-guided repair synthesis via programming by examples," *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 593–604.

[69] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 177–187.

[70] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, 2014, pp. 64–74.

[71] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 702–713.

[72] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 913–923.

[73] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo, "A deeper look into bug fixes: Patterns, replacements, deletions, and additions," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 512–515.

[74] X. Kong, L. Zhang, W. E. Wong, and B. Li, "Experience report: How do techniques, programs, and tests impact automated program repair?" in *Proceedings of the 26th International Symposium on Software Reliability Engineering*, 2015, pp. 194–204.

[75] X.-B. D. Le, D. Lo, and C. Le Goues, "Empirical study on synthesis engines for semantics-based program repair," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, 2016, pp. 423–427.

[76] T. Durieux, B. Danglot, Z. Zu, M. Martinez, and M. Monperrus, "The patches of the nopol automatic repair system on the bugs of defects4j version 1.1.0," Technical Report, Université Lille 1-Sciences et Technologies, 2017.

[77] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 191–201.

[78] F. Y. Assiri and J. M. Bieman, "Fault localization for automated program repair: effectiveness, performance, repair correctness," *Software Quality Journal*, vol. 25, no. 1, pp. 171–199, 2017.

[79] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "An empirical analysis of the influence of fault space on search-based automated program repair," *arXiv preprint arXiv:1707.05172*, 2017.
[80] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.
[81] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *Companion Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 180–182.
[82] S. Berlin, "Where is the bug and how is it fixed? an experiment with practitioners," *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 117–128.
[83] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genetic Programming and Evolvable Machines*, vol. 15, no. 3, pp. 281–312, 2014.
[84] Y. Yuan, H. Xu, B. Wang, and X. Yao, "A new dominance relation-based evolutionary algorithm for many-objective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 16–37, 2016.
[85] Y. Yuan, H. Xu, B. Wang, B. Zhang, and X. Yao, "Balancing convergence and diversity in decomposition-based many-objective optimizers," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 2, pp. 180–198, 2016.