Evolutionary Testing for Program Repair

Haifeng Ruan
National University of Singapore
Singapore
hruan@comp.nus.edu.sg

Hoang Lam Nguyen*
Humboldt-Universität zu Berlin
Germany
nguyehoa@informatik.hu-berlin.de

Ridwan Shariffdeen
National University of Singapore
Singapore
ridwan@comp.nus.edu.sg

Yannic Noller*
Singapore University of Technology and Design
Singapore
yannic.noller@acm.org

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

Abstract—**Automated program repair (APR) allows for autonomous software protection and improvement. Many proposed repair techniques rely on available test suites, since tests are available in real-world settings. Tests are incomplete specifications, however. As a result, repairs generated based on tests may suffer from the test overfitting problem. The patches produced by APR techniques may pass the given tests and thus be plausible, and yet be an incorrect patch. This hints towards more extensive test suites to guide program repair. Generating additional tests to improve the test suite quality is generally difficult because the oracle or expected observable behavior of the generated tests is unknown. In our work, we first construct additional oracles by instrumenting buggy programs from the DEFECTS4J benchmark with the knowledge obtained from the available bug reports. Then, we formulate a coevolution approach that generates tests and repairs in a unified workflow. The complete workflow is implemented as an extension of the well-known Java testing framework EVOSUITE. This includes re-purposing the search in EVOSUITE to search for repairs (instead of searching for tests) and enables an easy adoption for developers who are already familiar with EVOSUITE for test suite generation. The evaluation of our tool EVOREPAIR shows that coevolution positively impacts the quality of patches and tests. In the future, we hope that such coevolution can inspire new repair tools and techniques.**

Index Terms—**evolutionary testing, coevolution, automated program repair**

I. Introduction

Automated program repair (APR) [1] is a technology that aids developers in generating high-quality patches for software bugs. Various APR approaches have been proposed, such as search-based, semantic-based, and learning-based ones, amongst others. Since test cases represent a readily available specification of programs, most APR techniques use test cases as their main correctness criteria.

Although test cases are usually available in a software project, tests capture an incomplete specification of the program. Therefore, relying on tests can lead to the known issues of over-fitting to test data [2]. This leads to patches that are *plausible* because they pass the available test suite but still *incorrect* because they do not fix the bug in question. A straightforward intuition would be adding more test cases to improve the specification, but generating more test cases is non-trivial due to the test oracle problem (i.e., knowing the expected output for the generated tests). For example, while software fuzzing can generate many test inputs, it is restricted to fairly simple, pre-defined oracles like crash freedom or differential metrics like those used for regression testing. It is hard to formulate complex functional specifications without domain knowledge, and formal specifications are usually not available in practice.

*This work was done at the National University of Singapore.

We address the test oracle problem by referring to other sources of specifications: in practice, we often have developer-written bug reports [3], [4]. We can extract lightweight specifications from these bug reports and make them available as assertions by instrumenting the buggy program. In this work, we study how additional tests can improve patch generation and how test generation and patch generation can aid each other. Thereby, we formulate a *coevolutionary* generation of tests and patches to tackle the test overfitting problem and improve the quality of generated patches. Each coevolution cycle includes the generation of patches and tests while both generation processes guide each other. This leads to the incremental augmentation of the test suite with more tests and refines the patch space with high quality patches.

We note that Arcuri and Yao [5] explored coevolutionary program repair based on genetic programming. However, their strategy is based on the assumption that a formal specification of the intended program behavior (i.e., a formalization of the software requirements) is available. As it is widely known, formal specifications capturing software requirements are often unavailable in real-life software projects.

Our patch generation employs a coevolutionary search for patches *and* tests, initially driven by the developer-provided test suite. The resulting plausible patches are used as targets for the test generation so that the new tests should reach the corresponding fix locations and establish the incorrectness of these patches. Using the test execution results, we select seed patches for the next cycle of patch generation. Note that the fitness score of a patch can change over time, as new tests are generated. We do not discard patch candidates from the process as long as they pass the initial test suite. Our approach

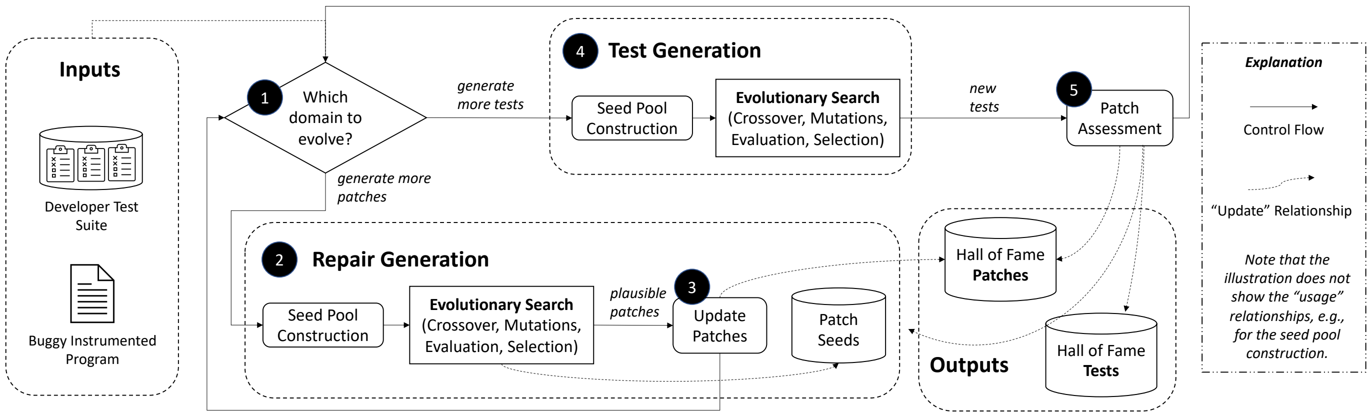


Fig. 1: Overview of EvoRepair's workflow.

however only reports patches that pass all available tests - including the newly generated tests.

We implement our proposed technique, EvoRepair, on top of the popular Java unit testing framework EvoSuite [6]. EvoSuite provides an extensible implementation of variety of evolutionary testing algorithms which we re-purpose for program repair. EvoSuite also supports mutation testing to assess different properties for test-suites, where the mutants can be re-purposed as patch candidates for program repair. We evaluate our proposed technique EvoRepair using 39 subjects derived from Defects4J [7]. We compare EvoRepair with several state-of-the-art tools including ARJA-e [8], TBar [9], and RewardRepair [10]. Our evaluation demonstrates that a *coevolutionary* process allows us to fix a larger set of bugs. The test cases generated by EvoRepair can prune the patch pool of EvoRepair by 69%, and also prune the patch pool of other tools by at least 33.3% in our experiments. Overall, we make the following core contributions:

- We propose a coevolutionary program repair approach integrated into the popular unit testing framework EvoSuite. We re-purpose EvoSuite's evolutionary testing search, to generate patches. Specifically, we re-purpose the evolutionary search for patch generation. The tests also serve as evidence of the trustworthiness of the patches reported.
- Using Defects4J subjects, for which ARJA [11] can generate plausible patches, we extracted 39 test oracles from the corresponding bug reports. These will be also useful for other researchers in testing and repair.
- We evaluate EvoRepair against evolutionary search based program repair tools. Experimental results demonstrate the increased efficacy of EvoRepair in reducing over-fitting patches from a pool of patch candidates.

II. Overview

Figure 1 shows the overall workflow of our coevolutionary generation of patches and tests. It takes in a buggy program and a developer provided test-suite. The developer provided test-suite contains at least one failing test. It outputs a set of candidate patches as well as a set of tests which provide evidence of veracity of the patches.

A. Workflow

In our workflow, the initial pool of patches is generated via mutations. Test generation with the goal of "killing" these patches is conducted. Thus, during the course of the repair algorithm - the test pool increases while the patch pool shrinks. Coevolutionary algorithms typically maintain a collection of individuals referred to as an *archive* or *hall of fame*, which is a collection of fittest individuals selected from multiple generations of coevolution [12]. Throughout the workflow, we maintain three of such archives:

$\mathbb{H}_{patches}$ the *Hall of Fame* for *patches* including all the patches that pass the currently considered tests.

$\mathbb{H}_{tests}$ the *Hall of Fame* for *tests* including the additionally generated test cases that were able to kill patches.

$\mathbb{V}_{patches}$ the *patch seeds*, i.e., the generated patches that pass the failing developer tests but fail on at least one other test (either initially passing developer tests or additionally generated tests). These patches are helpful because they fix the initial problem but fall short in other aspects. They serve as additional seeds for patch generation.

The central aspect of coevolutionary repair is the interaction between testing and repair, which is driven by the decision to evolve the test or patch domain (Step 1). In the beginning, since we assume there is a developer test suite, the coevolution starts from the patch domain and attempts to find plausible patches. Later, the workflow dynamically decides which domain to evolve. As long as the size of $\mathbb{H}_{patches}$ is below a pre-defined threshold, we will keep generating patches. If $\mathbb{H}_{patches}$ has reached this target size, we will keep generating tests to kill the patches in $\mathbb{H}_{patches}$. For practical usability, the size of $\mathbb{H}_{patches}$ should be small [13], so that a developer can examine the patches in $\mathbb{H}_{patches}$. On the other hand, $\mathbb{H}_{patches}$ should have enough patches to provide accurate feedback for test generation. In our experiment, we set the target size to 20.

The patch generation (Steps 2 and 3) starts by constructing the seed patch pool by combining the two sets $\mathbb{V}_{patches}$

```
Topic:
denominatorDegreeOfFreedom in FDistribution leads
to IllegalArgumentsException in
UnivariateRealSolverUtils.bracket

Description:
The problem is the 'initial' parameter to that
function, wich is POSITIVE_INFINITY and therefore not
within the boundaries. I already pinned down
the problem to the FDistributions
getInitialDomain()-method, wich goes like:

return getDenominatorDegreesOfFreedom() /
(getDenominatorDegreesOfFreedom() - 2.0);

Obviously, in case of  denominatorDegreesOfFreedom
is 2, this must lead to a division-by-zero, resulting
in POSTIVE_INFINITY.
```

(a) bug report for MATH-227

```
1  double getInitialDomain(double p, double d) {
2     double ret;
3     ret = d / (d - 2.0);
4     return ret;
5  }
```

(b) buggy code in MATH-227

```
1  T wrapper_method(Parameters p ...) {
2    if (instrumentation_enabled) {
3      T result = original_method(p);
4      if (<condition_for_buggy_behavior>) {
5        throw new RuntimeException("[
              Defects4J_BugReport_Violation]");
6      }
7        return result;
8    } else {
9        return original_method(p);
10   }
11 }
```

(c) general oracle instrumentation template

```
1  double getInitialDomain(double p, double d) {
2     if (Boolean.parseBoolean(System.getProperty("
          defects4j.instrumentation.enabled"))) {
3        double init = getInitialDomain_original(p, d);
4        double lowerBound = 0;
5        double upperBound = Double.MAX_VALUE;
6        if (init < lowerBound || init > upperBound) {
7           throw new RuntimeException("[
                Defects4J_BugReport_Violation]");
8        }
9        return init;
10    } else {
11       return getInitialDomain_original(p, d);
12    }
13 }
```

(d) instrumentation for MATH-227

Fig. 2: Illustration of the oracle generation from a developer provided bug-report and the buggy function, based on MATH-227 in Defects4J benchmark.

```
1  double getInitialDomain
2  (double p, double d) {
3    double ret = 0.0;
4    if (p > 0.5)
5      ret = d /(d - 2.0);
6    return ret;
7  }
```

(a) Patch p_1

```
1  double getInitialDomain
2  (double p, double d) {
3     double ret;
4     ret = d /(d - 1.0);
5     return ret;
6
7  }
```

(b) Patch p_2

```
1  double getInitialDomain
2  (double p, double d) {
3    double ret = 1.0;
4    if (d != 2.0)
5      ret = d /(d - 2.0);
6    return ret;
7  }
```

(c) Patch p_3

```
1  double getInitialDomain
2  (double p, double d) {
3    double ret = 1.0;
4    if (d > 2.0)
5      ret = d /(d - 2.0);
6    return ret;
7  }
```

(d) Patch p_4

Fig. 3: Evolution of patches captured in p_1, p_2, p_3 and p_4, generated through multiple iterations of coevolution.

and $\mathbb{H}_{patches}$. In the beginning, it is randomly initialized. Intuitively, we aim to refine the patches in $\mathbb{V}_{patches}$ by using the patches from $\mathbb{H}_{patches}$. During the evolutionary search, patches that pass the failing developer tests but fail any other test will be kept in $\mathbb{V}_{patches}$ as they can serve as helpful patch ingredients for future generations, while plausible patches will be kept in $\mathbb{H}_{patches}$. Patch generation ends when $\mathbb{H}_{patches}$ reaches a predefined upper-limit size, or when timeout occurs.

The test generation and patch assessment (Steps 4 and 5) aim at killing overfitting patches. It generates new test cases guided by the current patches in $\mathbb{H}_{patches}$. Patches that fail the additional tests are moved to $\mathbb{V}_{patches}$, while patches that pass all new tests remain in $\mathbb{H}_{patches}$.

The overall workflow ends after a predefined timeout or when the user stops the search. EVOREPAIR generates the sets $\mathbb{H}_{patches}$ (the most promising repair candidates that survived all generated test cases) and $\mathbb{H}_{tests}$ (the augmented test suite as an additional artifact for the user).

B. Illustrative Example

To further illustrate our proposed workflow, we show an example that is inspired by the bug report for MATH-227 [14], and the corresponding subject in Defects4J *math-95*. The original bug is in the class `FDistributionImpl` inside the method `getInitialDomain`, which accesses the initial domain value for an F-distribution. In our example, we kept the nature of the bug but simplified the context. Figure 2a depicts a snippet of the bug report for MATH-227, describing the flawed behavior and the expected behavior.

The buggy program (see Figure 2b) takes the desired probability p and the denominator degrees of freedom d. In particular, there are two incorrect behaviors: (1) for $d = 2$ there is a divide by zero error, because the divisor in line 2 is zero, and (2) for $d = 1$ the return value is negative. Both of these are unexpected behavior, which was reported by a user in the bug report (Figure 2a). Therefore, we create an oracle (see Figure 2d) in Section III, which makes the boundary check explicit and throws a custom exception, which is recognized by our test generation.

We assume that there is one failing developer test t_f with $p = 0.49$, $d = 2.0$, and an assertion that checks $ret < M$

for large constant M. In the execution of EVOREPAIR, Step 2 generates two plausible patches so that we have

$$\mathbb{H}_{patches} = \{p_1, p_2\}$$

Both p_1, p_2 pass the initially failing test case t_f. After returning to Step 1 and deciding that new tests should be generated, Step 6 produces a new test t_1 with $p = 0.6$ and $d = 2.0$ that *kills* p_1. Note that t_1 (as all additionally generated tests) does not define an assertion inside the test method but instead uses our custom oracle $ret < M$. After the patch assessment in Step 5, the patch sets are updated to

$$\mathbb{H}_{patches} = \{p_2\} \quad \mathbb{V}_{patches} = \{p_1\}$$

Inspired by the edits in p_1 and the updated $\mathbb{H}_{tests} = \{t_1\}$, a new iteration of repair generation produces the patch p_3 that passes all current tests. Another round of test generation produces t_2 with $p = 0.49$ and $d = 1.0$ *kills* p_2 and p_3 so that after patch assessment we have

$$\mathbb{V}_{patches} = \{p_1, p_2, p_3\} \quad \mathbb{H}_{patches} = \emptyset \quad \mathbb{H}_{tests} = \{t_1, t_2\}$$

Figure 3 shows all generated patches and the resulting final patch and test sets are:

$$\mathbb{V}_{patches} = \{p_1, p_2, p_3\} \quad \mathbb{H}_{patches} = \{p_4\} \quad \mathbb{H}_{tests} = \{t_1, t_2\}$$

We illustrate the concept of the coevolution of tests and patches, and that using bug-report-derived oracles, the test generation was able to kill overfitting patches. Overfitting patches are not completely discarded but re-purposed as seeds and ingredients for future repair generations, and finally, are evolved to the resulting correct patch.

III. TEST ORACLES

One of the fundamental challenges in software testing is the *test oracle* problem [15], [16], [17]. It is concerned with determining the expected behavior of a program. In our case, we have to reason about the additionally generated tests, and hence, we also need an oracle. There are learning-based test oracle construction techniques like SEER [18], TOGA [19], and ODS [20], and other heuristics-based techniques [21], [22], [23] to identify likely overfitting or likely incorrect patches. However, all these techniques only provide an approximation. While they can be used for patch assessment and ranking, we cannot use them to confidently prune patches in our coevolution context. Therefore, we decided to use the natural language specifications available in the bug reports in the Defects4J benchmark to create *user-originated* oracles. Techniques like JDoctor [24] and MeMo [25] can be used to extract specifications from Javadoc comments, but they strongly depend on the quality and structure of code comments. Preliminary results showed that they do not perform well on the bug reports included in Defects4J, which is why we opted to *manually* extract the oracles from the bug reports. We envision that the bug reporters could also write such oracles. Therefore, we only use the information provided by the reporter rather than the follow-up discussion of the maintainers, which is often also included in the online bug reports. For creating these oracles, we focused on the 59 subjects for which the state-of-the-art search-based APR technique ARJA [11] can generate plausible patches. We instrument the original buggy program to check for the buggy behavior mentioned in the bug report. If the buggy behavior is detected, the instrumentation will throw a custom exception, which will cause the abortion of the original program execution detected by our framework. To some extent, this represents a hand-written, bug-report-derived, functionality based oracle.

A. Oracle Writing Methodology

Using the bug report's description, we first identify the program location (i.e., class and method) where the incorrect behavior is observed. Then, we extract the condition under which the incorrect behavior is described. The instrumentation usually involves writing a wrapper function around the original code, which checks for a specific condition and throws a custom exception if the condition is satisfied. In general, we applied the template shown in Figure 2c.

Some bug reports mentioned unexpected exceptions, so we need to add a try-catch-block around the original method call. Note that our instrumentation can be dynamically enabled (line 2 in Figure 2c), which we enable by checking whether the system property `"defects4j.instrumentation.enabled"` is set. Therefore, we can enable the oracle only for the execution of the newly added test methods. This becomes necessary because our additional oracles can cause initially passing test cases to fail. The reason is that our oracles make the buggy behavior explicit and throw a custom runtime exception, which might not be handled by the existing test cases. However, a correct patch should also pass our oracle. In our overall approach, we enable the oracles for newly generated tests only and disable them for the existing tests.

For example, let us have a look at the report for MATH-227 mentioned in Section II. In the title of the report, the reporter mentions an unexpected IllegalArgumentsException. Further in the description, the reporter specifies that the problem is caused by parameter values being outside of expected bounds. Therefore, our instrumentation checks for these bounds and throws a custom runtime exception as shown in Figure 2d.

Note that our instrumentation is applied at the location where the bug reporter expects a program failure and makes the incorrect behavior explicit by throwing an exception. In contrast to other Information Retrieval (IR)-based techniques [4], [26], this usage of the bug reports does not serve the fault or fix localization but solely has the purpose of guiding the test generation. Further, we believe that creating such oracles needs a minor additional effort so that a bug reporter could provide them. Such an oracle also could be simplified to conventional assertions.

B. Failed Attempts

From the 59 subjects, we successfully created 39 oracles. We could not formulate a meaningful oracle for the other 20 subjects because of missing information about the expected behavior (14/20) or because no bug report was available (6/20). In general, we require information about the condition under which we can observe the failure and the observation location. In most of the cases with insufficient information (7/14), the report only provided a failing test case that did not allow for generalization beyond the specific test. For six other reports, there was no precise description of the *condition* under which the program behaves incorrectly. For example, the report for MATH-949[1] describes that a method incorrectly always returns zero, but did not specify *when* this is incorrect. Further, one bug report only suggested a fix without mentioning the buggy behavior. More information on these bug reports can be found in our artifact.

C. Oracle Types

During the oracle creation, we observed a few patterns that can be categorized as follows. For 20/39 subjects, the bug report described that a particular exception, e.g., a null pointer exception, should not be observed, indicating incorrect behavior. For these subjects, we throw our custom exception when catching the described exception to make the error explicit. For 12/39 subjects, the reports mentioned a general condition on the output or an intermediate value. In particular, for 6/39 subjects, the reports described specific expected value bounds. Additionally, one bug report described a property that can be used for differential testing: a Java class was found to be buggy by checking its output against another class that implements the same functionality (see MATH-631[2]).

IV. Coevolutionary Repair

We propose a program repair framework based on coevolution. Coevolution is defined as the simultaneous evolution of multiple populations that belong to different domains; the respective fitness functions guide the evolution of the populations. In our case, we would evolve two populations: a population of *patches* and a population of *test cases*. Specifically, given a buggy program, an initial test suite, and an oracle, we first search for some plausible patches to form the initial patch population. The test suite and the patches are then evolved alternately in search of test cases that invalidate the current patches and of patches that pass the growing test suite.

The key observation behind the framework is that program repair can be seen as a competitive coevolution problem. Competitive coevolution is one where the fitness of individuals from different populations are inversely related; the populations thus compete against each other. In the context of program repair, a better patch can pass more test cases, while a better test case can fail more patches. This duality makes a natural competitive coevolution problem.

Benefit of the coevolutionary approach is also dual. On the one hand, evolving the population of test cases improves the specification of the buggy program and helps generate less overfitting patches [2]. On the other hand, evolving the population of patches is conducive to better test cases that prune the patch pool more effectively.

[1]https://issues.apache.org/jira/browse/MATH-949

[2]https://issues.apache.org/jira/browse/MATH-631

Algorithm 1: Coevolutionary Repair

```
Input: buggy program, developer test suite
Output: plausible patches $\mathbb{H}_{patches}$
Output: additional test cases $\mathbb{H}_{tests}$
1  $\mathbb{H}_{patches} \leftarrow \emptyset$;  $\mathbb{V}_{patches} \leftarrow \emptyset$;  $\mathbb{H}_{tests} \leftarrow \emptyset$
2  while within timeout do
3      if evolve patch domain then
4          $P \leftarrow \mathbb{H}_{patches} \cup \mathbb{V}_{patches} \cup randomPatches()$
5          $P_{surviving}, P_{partial} \leftarrow evolve(P, f_{\mathbb{H}_{tests}})$
6          $\mathbb{H}_{patches} \leftarrow \mathbb{H}_{patches} \cup P_{surviving}$
7          $\mathbb{V}_{patches} \leftarrow \mathbb{V}_{patches} \cup P_{partial}$
8      else
9          $T \leftarrow \mathbb{H}_{tests} \cup randomTests()$
10         $T_{killing} \leftarrow evolve(T, f_{\mathbb{H}_{patches}})$
11         $\mathbb{H}_{tests} \leftarrow \mathbb{H}_{tests} \cup T_{killing}$
12
13         $\mathbb{H}_{patches}, P_{killed} \leftarrow update(\mathbb{H}_{tests}, \mathbb{H}_{patches})$
14         $\mathbb{V}_{patches} \leftarrow \mathbb{V}_{patches} \cup P_{killed}$
15 return $\mathbb{H}_{patches}$, $\mathbb{H}_{tests}$
```

Evolution of either population is performed by means of an evolutionary search. An evolutionary search is guided by a fitness function f. In each generation, the fitness of each individual in a population is computed with f, based on which some individuals are selected as parents. The parents then undergo mutation and crossover to yield the next generation. Note that patches and tests have their respective mutation and crossover operators.

In our coevolutionary repair, the fitness of a patch is related to its execution result on a selected set of test cases that have failed some patch before. We denote this set with $\mathbb{H}_{tests}$ and denote the resulted fitness function with $f_{\mathbb{H}_{tests}}$. Likewise, we have $\mathbb{H}_{patches}$ and $f_{\mathbb{H}_{patches}}$ for guiding the evolution of test cases. We summarize the workflow of coevolutionary repair in Algorithm 1. The patch population is initialized with $\mathbb{H}_{patches}$, $\mathbb{V}_{patches}$ and some random patches (to escape possible local optima). The population then undergoes evolution under the guidance of $f_{\mathbb{H}_{tests}}$ to yield plausible patches $P_{surviving}$ that pass the developer test suite and the tests in $\mathbb{H}_{tests}$. We also keep *partial* patches that pass the developer test suite but fail on other tests in $\mathbb{H}_{tests}$; these are stored in $\mathbb{V}_{patches}$, while $P_{surviving}$ are added to $\mathbb{H}_{patches}$. Evolution of test cases is similar: the tests in $\mathbb{H}_{tests}$ and some random tests are evolved with the goal to find new tests ($T_{killing}$) that kill patches in $\mathbb{H}_{patches}$. These tests are added to $\mathbb{H}_{tests}$. Afterwards we update $\mathbb{H}_{patches}$ and $\mathbb{V}_{patches}$ accordingly. We provide greater details of our approach in the subsections below.

A. Search Space

We construct the search space of patch generation with ARJA-e, which uses the statements in the buggy program as well as a set of predefined templates as fix ingredients. The search space of test generation is composed of all possible sets of statements of sizes from 1 to N (i.e. $n \in [1, N]$) where each test case can have a size from 1 to L (i.e. $l \in [1, L]$).

The search space of EvoRepair is the combination of the program repair space and the program test space. Such a large search space requires an optimized navigation strategy [13]. Previous work have shown that a coevolutionary search is much more effective for navigating such a large search space than an evolutionary search [27]. In addition, such algorithms can adaptively focus on relevant areas in the search space due to the mutability of the fitness landscape.

B. Fitness Functions: Patch Generation

Compared to most coevolutionary algorithms, which use a single objective for the selection of the population, the two populations in our framework both require multiple objectives. There are several benefits of using multi-objective fitness functions to search for plausible patches in the context of program repair [11]. An additional helper function (i.e., minimizing the patch size) can alleviate the problem of convergence to local minima. In addition, generating a plausible patch is necessary but not sufficient for the developer. We leverage the same fitness functions as defined in previous work [8] that uses the *patch size* [11], [28], [8] and the *failure rate* on tests [11], [29], [8], and formulate the optimization problem as follows:

$$\begin{cases} min\, f_1(p) = \sum_{j=1}^{n} b_j \\ min\, f_2(p) = \frac{\sum_{t\in T_{pos}} h(p,t)}{|T_{pos}|} + w \times \left(\frac{\sum_{t\in T_{neg}} h(p,t)}{|T_{neg}|} \right) \end{cases} \quad (1)$$

where $f_1(p)$ represents the size of the patch p by counting the number of edit operators in the patch, and $f_2(x)$ combines the failure rate of the positive tests T_{pos} with the weighted failure rate of the negative tests T_{neg}. $w \in (0, 1]$ controls the bias towards negative test cases. $h(p, t)$ indicates how badly a test failed on the patch by computing how far an assertion failure was from the expected value (see details in [8]). We employ NSGA-II [30] to minimize both objective functions.

C. Fitness Functions: Test Generation

For test cases, the fitness is computed using many objectives: (1) the ability to cover the fix locations, (2) the ability to cover the oracle location, (3) the ability to cover a target location (i.e., fix or oracle location) through different contexts, and (4) the ability to trigger our oracle's custom exception during mutation testing. The third and fourth objectives are inspired by the intuition that a test case is more likely to invalidate a patch if it tests the patch in multiple different ways. The optimization problem can be formalized as follows:

$$\begin{cases} min\ g_1(t, p) = d(p_{loc}, t) \\ min\ g_2(t, p) = d(o_{loc}, t) \\ min\ g_3(t, p) = d_{context}(\{p_{loc}, o_{loc}\}, t) \\ min\ g_4(t, p) = d_{mutant}(p_{loc}, t) \end{cases} \quad (2)$$

- $g_1(t, p)$ computes the distance of test t from covering the fix location $p._{loc}$ using approach level and normalized branch distance [31]; similarly, $g_2(t, p)$ targets the oracle location o_{loc}.
- $g_3(t, p)$ computes the distance to a target location (i.e., fix or oracle location) *through* a particular context (i.e., call stack): We want to explore the different contexts where each control dependency of a target location is indirectly covered through invocation of other (public) methods. The set of possible contexts can be (statically) derived from the call graph of the class under test. Given a test that covers a particular context, the fitness is computed as the target location distance. If a test does not trigger this context, it is assigned maximum distance instead. This objective is a variation of the *direct branch coverage* criterion [32].
- $g_4(t, p)$ computes the infection and propagation distance [33] w.r.t. a fix location mutant: To assess the sensitivity of test cases to changes at the fix locations, we perform *strong mutation testing* [34], [33] by applying small mutants at all fix locations and guide test generation towards "killing" these mutants. A mutant is considered to be killed if it leads to a state infection that propagates to an assertable output difference. In patch testing, we are particularly interested in killing mutants by throwing the custom oracle exception. By targeting only the fix locations, we further tackle the scalability issue of traditional mutation testing and control the budget allocation.

We employ DynaMOSA [35] to optimize the selection of test cases and objectives for this many-objective problem over all patches $p \in \mathbb{H}_{patches}$. In particular, we compute structural dependencies between the individual objectives in the same manner as for the traditional test case generation setting (i.e., based on the control dependency hierarchy). Note that for the distance calculation we execute the original program.

V. Implementation

Our approach is implemented on top of EvoSuite [6], a testing framework based on multi-objective evolutionary algorithms. It has interfaces for numerous evolutionary algorithms such as MOSA [36], DynaMOSA [35], and NSGA-II [30], and allows easy addition of new fitness functions. Below, we explain our extensions to EvoSuite, which have allowed it to perform coevolution of patches and tests.

A. Test Generation

We base our test generation on the DynaMOSA algorithm [35]. Originally, DynaMOSA has been introduced in the context of the many-objective coverage problem for automated test case generation. Our goal is not to produce a general test suite but to generate test cases that can identify overfitting patches. Therefore, we adapt the DynaMOSA implementation in EvoSuite as follows:

- *Population Initialization:* Instead of *randomly* initializing the population, we use seeds from $\mathbb{H}_{tests}$.
- *Test Case Evaluation:* Our search employs several customized fitness functions as introduced in Section IV-C.

- *Extended Coverage Archive:* The original coverage archive in DynaMOSA contains test cases that reach a target location not reached by previous test cases. We have added to the archive test cases that kill program mutants at the fix locations as well as any test case that covers one of the target locations (i.e., fix or oracle location). These additional test cases are helpful because a single test reaching a location is usually not sufficient to detect program behavior changed by a patch.

B. Patch Generation

The evolution in the space of program edits is based on ARJA-e [8], which represents patches as chromosomes, defines crossover and mutation operators of the chromosomes, and evolves chromosomes by the NSGA-II algorithm. For better usability, we have ported the patch representation and the operators to EvoSuite, which are interfaced with EvoSuite's NSGA-II implementation. In addition, we make the following changes to EvoSuite and ARJA-e:

- *Avoiding Changes to the Oracles:* For generated test cases to rule out overfitting patches, an injected oracle should remain in the program after a patch is applied. Therefore, we identify the oracles from abstract syntax trees and prevent generating patches that alter or remove them.
- *Avoiding Using Oracles in Repair:* We prevent using statements from the oracles as fix ingredients, though these statements can serve as fix ingredients in principle. This is because we need to compare EvoRepair with other tools, which may generate patches that change the oracle and thus cannot handle a program with an oracle.
- *Taking Seed Patches:* Just as we provide seed test cases for test generation, we also provide seed patches for patch generation. A portion of the seeds are "best-so-far" patches coming from $\mathbb{H}_{patches}$. Another portion comes from $\mathbb{V}_{patches}$. By combining the two seed sources, we hope to breed offspring patches that pass more test cases while avoid falling into local optima.

VI. Evaluation

In this section, we present the results of our evaluation for the proposed coevolutionary repair process. We first show the general effectiveness of EvoRepair in repairing bugs. We then show that patch and test generation benefit each other, confirming the usefulness of the coevolutionary approach. Overall, we investigate the following research questions.

RQ1 How effective is EvoRepair in repairing bugs compared to the state of the art of Java program repair?
RQ2 What is the impact of coevolution on patch generation?
RQ3 What is the impact of coevolution on test generation?

A. Evaluation Subjects

For our evaluation we use a subset of subjects from the popular Defects4J [7] benchmark v1.0.1. We selected subjects for which we were able to generate a test oracle using the bug report as discussed in Section III. Namely, we consider four projects from Defects4J: `Chart`, `Lang`, `Math` and `Time`. In total, we include the 39 subjects for which we were able to generate a test oracle (see Section III). Given the large amount of CPU time required to run the experiment, we focus on this limited subject set that can be fixed by our baseline tool ARJA [11]. We do not consider the other 20 subjects that do not have a test oracle because we focus on the coevolution of tests and patches.

B. Evaluation Tools

We compare our proposed technique EvoRepair with several state-of-the-art tools for Java program repair. We evaluate the repairability for our subset of defects using ARJA [11], ARJA-e [8], jGenProg [37], Cardumen [38], jKali [39], jMutRepair [40], Nopol [41], TBar [9], and RewardRepair [10]. We use the Astor framework [37], which implements the techniques Cardumen, jKali, jMutRepair, and jGenProg. Similarly, we reuse the ARJA [11] framework for the implementations of ARJA and ARJA-e. Since RewardRepair assumes perfect fault localization, we compute a list of fix locations prior to invoking the model, using the Ochiai algorithm [42] via GZoltar [43] framework. Note that only EvoRepair runs on the instrumented version; all other tools are executed on the original buggy program. EvoRepair does not use the instrumentation for fix localization and instead relies on the built-in spectrum-based fault localization in ARJA-e.

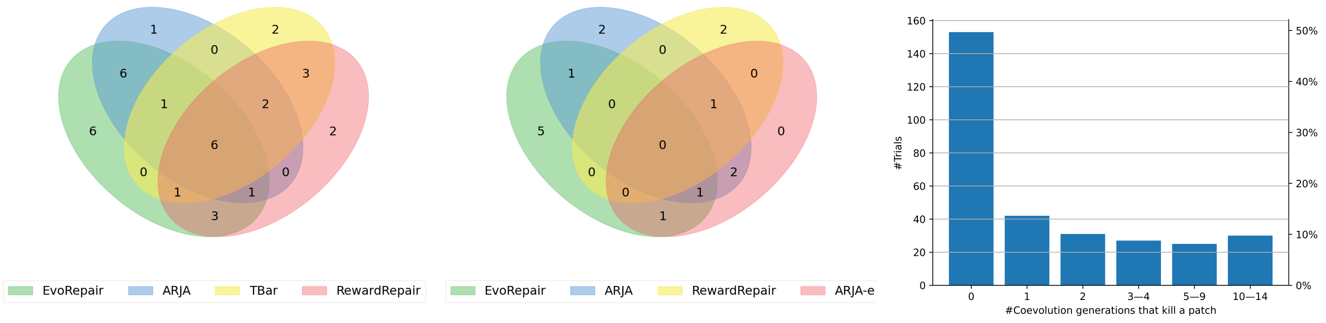
C. Experimental Setup

All our experiments were conducted on servers with 32 vCPU and 64GB memory using Docker containers. Additionally we provide an NVIDIA GeForce RTX 4090 GPU for RewardRepair. We set a 2h timeout for all our experiments per repair task following previous empirical studies [44]. Based on previous test runs, we set the internal timeout for patch generation to 10 minutes and for test generation to 1 minute in each coevolution iteration. The experiment on each subject is repeated for 10 times. For each tool, we report the number of bugs for which it can consistently generate a plausible patch in all 10 runs. We also report the cumulated total number of bugs fixed in at least one of the 10 runs. For the number of correctly fixed bugs (i.e., semantically equivalent to the developer fix), we only report the total over 10 runs in this paper since correct patches are sparse. However, we include all experimental results in our artifact.

Seed Selection Parameters: EvoRepair seeds patch generation with patches from $\mathbb{H}_{patches}$ and $\mathbb{V}_{patches}$. In the experiment, we form $\frac{1}{2}$ of the initial population from $\mathbb{H}_{patches}$ and $\frac{1}{4}$ from $\mathbb{V}_{patches}$, and the remaining $\frac{1}{4}$ is randomly initialized. When $\mathbb{H}_{patches}$ and $\mathbb{V}_{patches}$ do not have enough patches, random patches are used instead. For test generation, $\frac{1}{2}$ of the initial population is from $\mathbb{H}_{tests}$, and the other $\frac{1}{2}$ is randomly initialized.

D. (RQ1) Efficacy of EvoRepair

Figure 4a depicts the number of bugs each tool could plausibly fix in every one of the 10 runs. For brevity, we



(a) Plausible Patches (b) Correct Patches (c) Distribution of Patch Killing

Fig. 4: (a) Venn diagram of bugs for which repair tools found a plausible patch in every repetition. Omitted from the figure are JGENPROG and ARJA-e. They consistently fixed 13 and 6 bugs, with 8 and 4 bugs overlapping with EVOREPAIR. The other tools only fixed a subset of the bugs fixed by JGENPROG. (b) Venn diagram of bugs for which repair tools found a correct patch in at least one repetition. (c) Distribution of experimental trials by the number of generations where an overfitting patch is removed. The left axis shows the absolute number of trials while the right shows the relative number.

omit numbers from CARDUMEN, JKALI, and JMUTREPAIR as they are a subset of JGENPROG. For 24 bugs, EVOREPAIR finds a plausible patch in every repetition, and there are 33 bugs for which EVOREPAIR was able to find a plausible patch in at least one repetition. In comparison, REWARDREPAIR consistently fixes 18 bugs; ARJA finds a plausible patch for 17 bugs in every repetition and for 21 bugs in at least one repetition; TBAR consistently fixes 15 bugs, and ARJA-e finds a plausible patch for 6 bugs in every repetition and for 19 bugs in at least one iteration. None of the other tools have fixed more than 15 bugs.

EVOREPAIR also produced more correct patches. As shown in Figure 4b, over the 10 repetitions, EVOREPAIR correctly fixed 8 of the 39 bugs. This is followed by ARJA, ARJA-e, and REWARDREPAIR, which correctly fixed 7, 5, and 3 bugs. The other tools correctly fixed no more than 2 bugs each. Note that, although ARJA and ARJA-e had similar performance with EVOREPAIR, they generated more than 200 plausible patches on average for each bug. In contrast, EVOREPAIR maintains no more than 20 patches in the hall of fame. We also notice from Figure 4b that EVOREPAIR, ARJA, and REWARDREPAIR have several uniquely fixed bugs, due to the different search strategies of the tools.

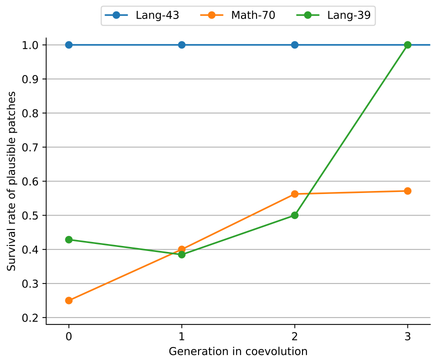
E. (RQ2) Impact of Coevolution on Patch Generation

Intuitively, patch quality is improved because the additional test cases can remove overfitting patches during the search for a correct patch. As will be shown below, this intuition is supported by our experiments.

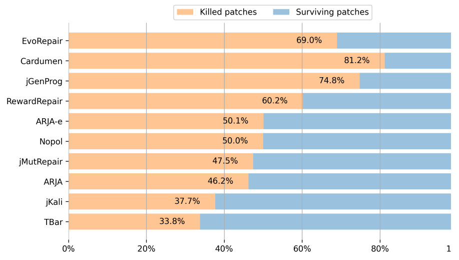
We have collected the number of plausible patches generated and the number of patches killed in our experiments. Overall, about 14,500 patches were generated, of which over 10,000 were killed, making a signification reduction of 69.0% to the patch pool. We further investigate how these reductions are distributed among experimental trials and among the coevolotion generations within the trials. Of our 390 trials (39 subjects $\times$ 10 repetitions), 308 have generated plausible patches. In these trials, it is desirable if overfitting patches have been detected in different generations, which would mean that the test cases were constantly pruning the patch pools. We plot in Figure 4c the distribution of the 308 trials by the number of coevolution generations where an overfitting patch was detected. A total of over 35% of the trials have plausible patches killed in more than one generation, and over 25% of the trials have plausible patches killed in three or more generations. This means that in a reasonable number of trials, the test cases are steadily removing overfitting patches.

The steady pruning of patch pools has led to a gradual improvement in the quality of the generated patches. The quality of patches found in a certain generation can be reflected by *survival rate*, which is the fraction of these patches that survive until the end of coevolution. From our 390 trials, we have collected the survival rate of patches for the 162 trials that have both plausible patches found in multiple generations and at least one surviving patch. These trials can be further divided into three categories, each represented by a line in Figure 5a. In 71 (43.8%) trials, survival rates were always one, represented by the Lang-43 line in Figure 5a, which means no test case was generated that could kill a patch. In 62 (38.3%) trials, survival rates have monotonically increased throughout the coevolutions, represented by the Math-70 line. Only in 29 (17.9%) trials, the survival rate dropped at some point; however, the survival rate may still go back up (see Lang-39).

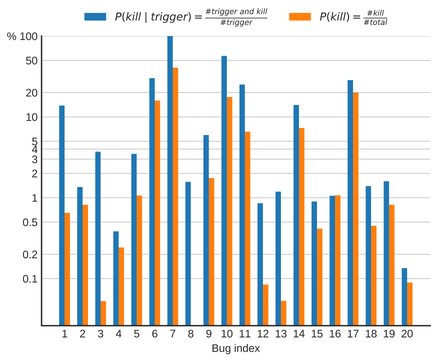
In order to see whether coevolution can help improve patch quality of other repair tools, we have also pruned patch pools of the other tools with our test cases. Figure 5b shows the fraction of overfitting patches by each repair tool detected by the additional test cases generated by EVOREPAIR. While the test cases had not been generated to target these patches, more than 60% of the plausible patches generated by REWARDREPAIR, JGENPROG, and CARDUMEN are removed, and minimum of 33% of the plausible patches generated by the other tools are also removed. The high proportions of detected



(a) Survival Rate



(b) Killed Patches



(c) Probability of Killing a Patch

Fig. 5: (a) Change in the survival rate of plausible patches over coevolution generations in three representative trials. (b) Percentage of patches of each repair tool killed by additional test cases. (c) Probability of a triggering test and an average test to kill patches on 20 bugs (y-axis on a log scale).

overfitting patches show the great potential of coevolution in improving patch quality of different repair tools.

F. (RQ3) Impact of Coevolution on Test Generation

This section demonstrates that our coevolutionary workflow improves the quality of generated test suites. In particular, we illustrate the effectiveness of our fitness functions by showing the effectiveness of our core intuition: triggering the custom exception on the *original* program is a good predictor of whether it can kill a patch, i.e., trigger the exception on a *patched* program. This intuition directly motivates the fitness function g_2 and indirectly also g_3 and g_4 (see Section IV-C).

To illustrate this, we have compared in Figure 5c how likely it is for a triggering test and for an average test to kill at least one patch. The comparison is performed on 20 bugs for which there are both triggering tests and killing tests. Note that the y-axis is on a log scale. A test that triggers the oracle on the original program has a much higher probability (10.9x on average and 70.4x maximum) of killing a plausible patch afterwards. In other words, the fitness function performs well in selecting potential killing tests.

G. Threats to Validity

External Validity: To mitigate the selection bias for our comparison with the state-of-the-art APR tools, we selected the seven best-performing tools reported in previously studied large-scale empirical evaluations [44] and REWARDREPAIR as the more recent state of the art learning based tool [10]. Since our implementation is based on ARJA, we focused the benchmark selection on the subjects in Defects4J, for which ARJA can find plausible patches. In the future, we plan to evaluate the efficacy of EVOREPAIR on a more diverse set of subjects in recently proposed data-sets [45], [46] of Java bugs.

Internal Validity: EVOREPAIR and the state-of-the-art Java repair tools in our comparison have non-deterministic components, and hence, can produce different results for different runs. To alleviate the threat of random observations, we ran each of the repair tasks 10 times and report the average values. Our artifact includes all experimental results. An additional threat to internal validity is that the manually written test oracles may not be fully correct if the referenced bug reports are incomplete or incorrect. Two of the authors independently double-checked the correctness of written oracles with respect to the available bug reports. Finally, our experiments used a two-hour timeout, and it is possible that larger timeouts can lead to other observations.

Construct Validity: Our analysis focuses primarily on identifying test-adequate (i.e., plausible) patches; however, these patches can still be incorrect due to the overfitting issue [2]. EVOREPAIR mitigates the overfitting issue by generating additional tests. Further, we also manually assess and report the correctness of the identified patches.

Our replication package is available via:
https://figshare.com/s/d800c8b6498d207f2cdd

VII. RELATED WORK

The related work includes automated coevolutionary searches, especially for tests and patches, testing and repair (in particular, the approaches that leverage evolutionary searches), and test generation for patch testing.

A. Coevolution of Tests and Patches

Most related to our work is the approach by Arcuri and Yao [5]. It takes a formal specification and a buggy program as input and co-evolves test cases and programs using genetic programming with the goal of repairing the buggy program. The formal specification allows them to generate as many unit tests as they want, which is an essential part of their technique. Moreover, the designed fitness function that drives the evolution is derived from the formal specification. With the general lack of formal specifications in practice, we decided to instead focus on available bug reports and extract generally valid constraints on the expected behavior, and use them as test oracles. The work of FIX2FIT [47] generates new tests (to prune overfitting patches) but does not evolve existing ones.

Our work is distinct in that it evolves the test and the patch pool by reusing the mutation-based search in EvoSuite. In the past, EvoSuite has been used to perform related adaptions, e.g., to evolve assertions [48], [49]. Contrary to these works, we do not invalidate the oracles. Instead, we use the oracles for invalidating and evolving patches.

Concolic Program Repair (CPR) [50] proposes generating new tests to prune overfitting patches. CPR still needs a lightweight, user-provided constraint to reason about the generated patches. CPR does not mutate the patches after test generation; their proposed refinement only removes overfitting patches from the patch pool. The limitation in CPR is the assumption of having the correct patch in the patch pool, whereby the correct patch can be identified after the removal of overfitting patches. We make no such assumptions.

B. Evolutionary Repair

Automated Program Repair (APR) [1] can be broadly categorized into three areas: search-based, semantic-based, and learning-based. EvoRepair is most related to evolutionary repair algorithms such as GenProg [29] that operate under the redundancy assumption, also known as the plastic surgery hypothesis [51]. The assumption is that the repair ingredients can be extracted from elsewhere in the buggy program itself. Empirical studies [44] show that this assumption does not hold true for most of the bugs in the popular Defects4J benchmark [7]. Repair techniques that restrict the search space to the buggy program itself have not been able to successfully find a plausible patch for many of the bugs in the considered benchmarks. ARJA-e [8] is a more recent study on more fine-grained changes that extends the search space beyond the redundancy assumption, allowing to fix a much higher number of bugs in the Defects4J benchmark. Similar to SimFix [52], it combines the search space with repair templates [9], [53] and extends the redundancy assumption by allowing more fine-grained modifications to AST node elements, thereby expanding the search space. Modifying AST node elements by replacing them with similar alternatives provides ARJA-e the capability to generate new code, which allows fixing bugs previously not fixed by its predecessors ARJA [11] and GenProg [29]. Since ARJA-e represents the state of the art in evolutionary repair, we build our repair based on the patch enumeration in ARJA-e. Note that approaches like VarFix [28], which exploit similarities between test executions to speed up the exploration, are complementary to our approach.

C. Evolutionary Testing

Our implementation builds on EvoSuite [6], the state-of-the-art evolutionary framework for whole test suite generation. Driven by coverage metrics, the test generation thereby handles test data instantiation, method call sequences, and retrieval of regression oracles via mutation testing. In our program repair context, we have no access to a reference solution and hence cannot use the regression oracles generated by EvoSuite. Instead, we have to generate our own test oracles. EvoSuite interfaces with numerous genetic algorithms such as MOSA [36], DynaMOSA [35], and NSGA-II [30]. EvoRepair also incorporates these well-researched search strategies for multiple objectives in each domain.

D. Test Generation for Patch Testing

Existing works in patch testing aim at alleviating the overfitting issue [50], [47], [54]. For example, CPR [50] co-explores tests and abstract patches with the goal of alleviating overfitting patches. It generates new test cases with a lightweight, user-provided constraint as a test oracle. However, CPR does not mutate the patches after test generation but simply refines the abstract patches to exclude patches that violate the test oracle. Fix2Fit [47] identifies overfitting patches by using grey-box fuzzing to generate new tests and using crash-freedom as an oracle. It is a post-processing technique that can enhance any APR technique. Differently, UnsatGuided [54] targets input data of synthesis-based repair techniques like Nopol [41] and Semfix [55]. It aims to strengthen repair constraints by generating additional tests that can supplement the developer test suite. They show that their approach can indeed help to prune patches that are overfitting because of regression-introducing edits. However, they also acknowledged that due to the oracle problem, UnsatGuided is unsuccessful in pruning other kinds of overfitting patches. Other works in patch testing focus on the identification of regression errors [56], [57], [58]. In contrast to these techniques, EvoRepair evolves the test pool as well as the patch pool via re-use of the mutation-based search in EvoSuite, and goes beyond regression errors with test oracles derived from bug reports. We further integrate test generation as an integral element of the coevolution, instead of having it as a pre- or post-process of APR.

VIII. Conclusion

Our work presents a practical perspective on the coevolution of patches and tests. We addressed the oracle problem in test generation not by assuming a formal specification but by considering which information and requirements a user and bug reporter can provide. Therefore, we extracted test oracles from bug reports in Defects4J. We implemented the proposed workflow in our tool EvoRepair as an extension of EvoSuite, a well-known tool for test generation. We hope that our practical assumption of test oracles and the integration into EvoSuite lowers the entrance barrier for software developers to apply APR in practice. Furthermore, the coevolution concept is not restricted to EvoSuite and can be ported to other patch and test generators. Our evaluation, including the comparison with the state-of-the-art techniques, shows that EvoRepair is highly competitive and that the coevolution improves the patch and test quality.

Acknowledgments

This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001.

References

[1] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, p. 56–65, nov 2019. [Online]. Available: https://doi.org/10.1145/3318162

[2] 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*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 532–543. [Online]. Available: https://doi.org/10.1145/2786805.2786825

[3] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 361–370. [Online]. Available: https://doi.org/10.1145/1134285.1134336

[4] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "Ifixr: Bug report driven program repair," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 314–325. [Online]. Available: https://doi.org/10.1145/3338906.3338935

[5] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, 2008, pp. 162–168.

[6] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[7] 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*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[8] Y. Yuan and W. Banzhaf, "Toward better evolutionary program repair: An integrated approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, jan 2020. [Online]. Available: https://doi.org/10.1145/3360004

[9] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[10] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1506–1518. [Online]. Available: https://doi.org/10.1145/3510003.3510222

[11] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, 2020.

[12] E. Popovici, A. Bucci, R. P. Wiegand, and E. D. De Jong, *Coevolutionary Principles*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 987–1033. [Online]. Available: https://doi.org/10.1007/978-3-540-92910-9_31

[13] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust enhancement issues in program repair," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2228–2240. [Online]. Available: https://doi.org/10.1145/3510003.3510040

[14] "Math-227 bug," https://issues.apache.org/jira/browse/MATH-227.

[15] E. J. Weyuker, "On Testing Non-Testable Programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 11 1982. [Online]. Available: https://doi.org/10.1093/comjnl/25.4.465

[16] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. Jenny Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121213000563

[17] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[18] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, "Perfect is the enemy of test oracle," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 70–81. [Online]. Available: https://doi.org/10.1145/3540250.3549086

[19] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2130–2141. [Online]. Available: https://doi.org/10.1145/3510003.3510141

[20] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2920–2938, 2022.

[21] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 727–738. [Online]. Available: https://doi.org/10.1145/2950290.2950295

[22] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 789–799. [Online]. Available: https://doi.org/10.1145/3180155.3180182

[23] A. Ghanbari and A. Marcus, "Patch correctness assessment in automated program repair based on the impact of patches on production and test code," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 654–665. [Online]. Available: https://doi.org/10.1145/3533767.3534368

[24] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 242–253. [Online]. Available: https://doi.org/10.1145/3213846.3213872

[25] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, and A. Carzaniga, "Memo: Automatically identifying metamorphic relations in javadoc comments for test automation," *Journal of Systems and Software*, vol. 181, p. 111041, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001382

[26] M. Motwani and Y. Brun, "Better automatic program repair by using bug reports and tests together," in *International Conference on Software Engineering (ICSE)*, 2023.

[27] L. Pagie and M. Mitchell, "A comparison of evolutionary and coevolutionary search," *International Journal of Computational Intelligence and Applications*, vol. 02, no. 01, pp. 53–69, 2002. [Online]. Available: https://doi.org/10.1142/S1469026802000427

[28] C.-P. Wong, P. Santiesteban, C. Kästner, and C. Le Goues, "Varfix: Balancing edit expressiveness and search effectiveness in automated program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 354–366. [Online]. Available: https://doi.org/10.1145/3468264.3468600

[29] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2009.

[30] 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.

[31] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294

[32] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering*, M. Barros and Y. Labiche, Eds. Cham: Springer International Publishing, 2015, pp. 93–108.

[33] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, Jun. 2015. [Online]. Available: https://doi.org/10.1007/s10664-013-9299-z

[34] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[35] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.

[36] ——, "Reformulating branch coverage as a many-objective optimization problem." in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 4 2015, pp. 1–10.

[37] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 441–444. [Online]. Available: https://doi.org/10.1145/2931037.2948705

[38] ——, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, ser. Lecture Notes in Computer Science, T. E. Colanzi and P. McMinn, Eds., vol. 11036. Springer, 2018, pp. 65–86. [Online]. Available: https://doi.org/10.1007/978-3-319-99241-9_3

[39] 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*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 24–36. [Online]. Available: https://doi.org/10.1145/2771783.2771791

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

[41] J. Xuan, M. Martinez, F. DeMarco, M. Clément, 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.

[42] A. Ochiai, "Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions," *Bulletin of Japanese Society of Scientific Fisheries*, vol. 22, pp. 526–530, 1957.

[43] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, M. Goedicke, T. Menzies, and M. Saeki, Eds. ACM, 2012, pp. 378–381. [Online]. Available: https://doi.org/10.1145/2351676.2351752

[44] T. Durieux *et al.*, "Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts," in *Proceedings of the 27th ACM ESEC/FSE '19*, 2019.

[45] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 55–56. [Online]. Available: https://doi.org/10.1145/3135932.3135941

[46] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An extensible java bug benchmark for automatic program repair studies," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, feb 2019, pp. 468–478. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER.2019.8667991

[47] X. Gao, S. Mechtaev, and A. Roychoudhury, *Crash-Avoiding Program Repair*. New York, NY, USA: Association for Computing Machinery, 2019, p. 8–18. [Online]. Available: https://doi.org/10.1145/3293882.3330558

[48] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1178–1189. [Online]. Available: https://doi.org/10.1145/3368089.3409758

[49] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Oasis: Oracle assessment and improvement tool," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 368–371. [Online]. Available: https://doi.org/10.1145/3213846.3229503

[50] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 390–405. [Online]. Available: https://doi.org/10.1145/3453483.3454051

[51] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 306–317. [Online]. Available: https://doi.org/10.1145/2635868.2635898

[52] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 298–309. [Online]. Available: https://doi.org/10.1145/3213846.3213871

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

[54] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system," *Empirical Software Engineering*, vol. 24, no. 1, pp. 33–67, 2019.

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

[56] S. Shamshiri, G. Fraser, P. Mcminn, and A. Orso, "Search-based propagation of regression faults in automated regression testing," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 396–399.

[57] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a doubt: Testing for divergences between software versions," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1181–1192. [Online]. Available: https://doi.org/10.1145/2884781.2884845

[58] Y. Noller, C. S. Păsăreanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske, "Hydiff: Hybrid differential software analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1273–1285. [Online]. Available: https://doi.org/10.1145/3377811.3380363