STUBCODER: Automated Generation and Repair of Stub Code for Mock Objects[*]

HENGCHENG ZHU, The Hong Kong University of Science and Technology, China

LILI WEI, McGill University, Canada

VALERIO TERRAGNI, The University of Auckland, New Zealand

YEPANG LIU, Southern University of Science and Technology, China

SHING-CHI CHEUNG[†], The Hong Kong University of Science and Technology, China

JIARONG WU, The Hong Kong University of Science and Technology, China

QIN SHENG, BING ZHANG, and LIHONG SONG, WeBank Co. Ltd., China

Mocking is an essential unit testing technique for isolating the class under test (CUT) from its dependencies. Developers often leverage mocking frameworks to develop stub code that specifies the behaviors of mock objects. However, developing and maintaining stub code is labor-intensive and error-prone. In this paper, we present STUBCODER to automatically generate and repair stub code for regression testing. STUBCODER implements a novel evolutionary algorithm that synthesizes test-passing stub code guided by the runtime behavior of test cases. We evaluated our proposed approach on 59 test cases from 13 open-source projects. Our evaluation results show that STUBCODER can effectively generate stub code for incomplete test cases without stub code and repair obsolete test cases with broken stub code.

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

Additional Key Words and Phrases: Software Testing, Mocking, Test Generation and Repair, Genetic Programming, Evolutionary Computation, Program Analysis

1 Introduction

Unit testing is an important testing paradigm that focuses on the correctness of a single software component (e.g., class in JAVA) [2]. In practice, a class is commonly implemented to leverage other classes' functionality. These classes constitute *test dependencies*, which are invoked when testing the *class under test* (CUT). To test the CUT in isolation, developers often substitute dependencies with *test doubles* [38], which play the role of dependencies for testing purpose only. There are five main types of test doubles: dummy, stub, mock, spy, and fake [38]. Following the popular terminology of the MOCKITO framework [12], we use the term *mock objects* [47] to collectively refer to dummy, stub, and mock test doubles. The mock objects in MOCKITO can act like any of these three types of test doubles in a unit test [12]. In a nutshell, mock objects are designed to simulate the reactions of

---

Authors' addresses: Hengcheng Zhu, hzhuaq@connect.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Lili Wei, lili.wei@mcgill.ca, McGill University, Montreal, Canada; Valerio Terragni, v.terragni@auckland.ac.nz, The University of Auckland, Auckland, New Zealand; Yepang Liu, liuyp1@sustech.edu.cn, Department of Computer Science and Engineering, Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China; Shing-Chi Cheung, scc@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Jiarong Wu, jwubf@cse.ust.hk, The Hong Kong University of Science and Technology, Hong Kong, China; Qin Sheng, entersheng@webank.com; Bing Zhang, billzzhang@webank.com; Lihong Song, jeaninesong@webank.com, WeBank Co. Ltd., Shenzhen, China.

```
 1 @Test
 2 public void loginRetryTest() {
 3     // Arrange: Prepares the dependencies and input data
 4     //          for the class under test.
 5     var dao = mock(UserDao.class); // mock object
 6     var user = mock(User.class);   // mock object
 7     var sha1 = DigestUtils.sha1Hex("bar");          STUB CODE
 8     when(user.getPasswordHash()).thenReturn(sha1); // Stub Call 1
 9     when(dao.findUser(eq("foo")))                  // Stub Call 2
10         .thenThrow(new TimeoutException())         // First reaction
11         .thenReturn(user);                         // Second reaction
12
13     // Act: Exercises the feature
14     var underTest = new LoginService(dao);
15     var loginResult = underTest.login("foo", "bar");
16
17     // Assert: Verifies the result.
18     verify(dao, times(2)).findUser(any());// mocking call
19     assertTrue(loginResult.isSuccessful());
20 }
21
22 // CUT
23 public LoginResult login(String userName, String password) {
24     User user = null;
25     while(user == null) {
26         try {
27             user = dao.findUser(userName);
28         } catch (TimeoutException e) { ... }
29     }
30     var hash = DigestUtil.sha1Hex(password);
31     if (hash.equals(user.getPasswordHash()))
32         // success
33     else
34         // failure
35 }
```

Listing 1. An Illustration of Unit Test with Mock Objects

test dependencies (i.e., via stub calls) or validate their interactions with the CUT (i.e., via mocking calls) [52].

Listing 1 illustrates a JUnit test case with mock objects implemented using the Mockito framework [12]. The unit test aims to validate the login function of LoginService, which leverages its test dependency UserDao to establish a database connection. The test case simulates the dependent database service UserDao and a database entity User with two mock objects. Lines 7–11 give the stub code that specifies the behaviors of the mock objects when their methods are (indirectly) invoked at Line 15. The first invocation of the method findUser throws an exception to simulate an unstable connection (Line 10). The second invocation of the method findUser returns a User object to simulate a successful database query (Line 11). The returned User object is another mock object to simulate a database entity. It returns the SHA-1 digest of a predefined password when its method getPasswordHash is invoked (Line 7). At Line 18, a mocking assertion verify checks whether the CUT invoked the method findUser twice.

With such mock objects, developers no longer need to set up a database for testing or unplug the network cable to trigger an exception. Similar to the example, developers often replace dependencies with mock objects and specify their behaviors with stub code when the dependencies are hard to set up, flaky, faulty, or even not yet implemented [47, 59]. The use of mock objects allows developers to focus on the CUT without worrying about the correctness or availability of its dependencies.

Developing and maintaining stub code is challenging. When developing stub code for a mock object, developers need to carefully consider its possible interactions with the CUT, and simulate the reactions accordingly when its methods are called. Stub code is tightly coupled with a specific

implementation of the CUT (and its dependencies) and is easy to get broken when the implementation of CUT evolves. Take the test case in Listing 1 as an example. When the implementation of UserDao, User, or LoginService changes, the stub code will become broken since it no longer specifies the behaviors for the APIs needed by the test case. For example, when the signature of findUser is changed from findUser(userName) to findUser(userName, passwordHash), the stub code are broken and does not compile. In this case, the stub code needs to be updated to adapt to the new implementation. In real-world projects, such updates need to be done frequently to keep the behaviors of mock objects consistent with the production code [47]. This activity is labor-intensive and error-prone [20].

Previous works on automatic stub code generation for mock objects rely on a capture-and-replay approach [18, 21, 32, 45, 46]. Given an executable test case, such techniques generate stub code in three steps: (1) execute the test cases capturing the interactions between the CUT and its dependencies, (2) replace the dependencies with mock objects, and (3) create stub code according to the captured interactions. As such, capture-and-replay techniques are able to generate stub code for only those test cases without mock objects. This is because they need to invoke the actual dependencies. However, the study of Spadini et al. [47] shows that for 83% of test cases that use mock objects, the mock objects are introduced when the actual dependencies are hard to set up, flaky, or unavailable. Therefore, capture-and-replay techniques are inapplicable to the majority of situations where mock objects are used.

Our goal is to synthesize stub code for mock objects without capturing the actual behaviors of the dependencies. This is challenging because it requires identifying the desired mock object's behavior for a specific test case. Indeed, mock objects are often test-specific because the same dependency class often has different behaviors in different test cases [59].

In regression testing, we want to synthesize stub code to test future versions of the CUT. In such a context, we do not aim to generate a stub code that makes the test pass or fail depending on whether the current version is faulty or not. We aim to generate the stub code that makes the test pass on the current CUT version, which aims to detect regression bugs introduced in future versions. Our observation is that the expected behavior of such test-passing stub code is often encoded in the CUT execution code and test oracles. For example, consider the test case of Listing 1 without the stub code highlighted in yellow (Lines 7 to 11). The expected behavior of the test case is given by Lines 14 and 15, which specify how the CUT should be invoked, and Lines 18 and 19, which specify the expected output of the method under test. The desired stub code (Lines 7 to 11) is the one that makes the test pass.

In this paper, we present **StubCoder**, the first technique to automatically generate stub code without executing the actual dependencies. Given an incomplete test case without stub code, StubCoder leverages the CUT execution code and test oracles as specifications to guide the synthesis of stub code to make the tests pass for the current implementation. As mentioned above, the synthesized stub code satisfies the expected behavior of the test case in the regression testing scenario and could detect bugs in future versions.

Due to the huge search space of possible stub code, it is infeasible to randomly or systematically explore all the possible candidate stub code to find a test-passing one. As such, we design StubCoder based on an evolutionary algorithm that drives the search by examining the runtime behavior of each candidate stub code. In particular, StubCoder employs a novel fitness function that captures how close a candidate stub code is to test-passing stub code. The fitness function captures several runtime aspects like the distance between the expected and actual value of each oracle assertion, which effectively directs the search towards the candidates that are more likely to pass the test.

Notably, StubCoder can also be used for repairing stub code that is broken due to code evolution. In such cases, StubCoder prioritizes the selection of code elements in the broken stub code when

constructing a new candidate stub code. Indeed, test-passing stub code might be syntactically similar to the obsolete one.

STUBCODER has two application scenarios: (1) When adding a new test case, developers can write the code that exercises the CUT using mock objects and specify the expected behavior for the test case by writing oracle assertions (i.e., JUNIT assertions and mocking calls). STUBCODER will automatically generate the stub code. (2) When the stub code in some test cases becomes obsolete due to software evolution, developers can run STUBCODER to repair the broken stub code. By supporting these two scenarios, STUBCODER helps relieve developers from the tedious manual effort of stub code development and maintenance.

We evaluated STUBCODER on 59 test cases collected from 13 open-source projects in both application scenarios. Although modern program synthesis tools (e.g., GitHub Copilot) can suggest possible statements to complete test cases, they do not aim to synthesize test-passing stub code. Since there is no related tool that generates stub code under these scenarios, we compared with a variant of STUBCODER based on an unguided strategy. In the both scenario, STUBCODER successfully generates stub code for 76% of the test cases in half of the repetitions. Compared with the unguided variant, STUBCODER has a higher success rate and synthesizes the stub code faster. Moreover, 57% of the synthesized stub code have identical fault detection capability as those written by developers (measured by mutation coverage in Table 4 and Table 5).

To summarize, this paper makes three major contributions:

- We design and implement STUBCODER, the first automatic stub code synthesis technique that can effectively synthesize stub code for the test dependencies of a target unit test case. We equipped STUBCODER with a novel fitness function that examines the runtime behaviors of the test case to guide the search of the test-passing stub code.
- We construct the first benchmark for evaluating stub code generation and repair techniques. It is composed of 59 test cases from 13 open-source projects. STUBCODER can effectively synthesize stub code for incomplete test cases in both application scenarios and it outperforms the baselines as well as its unguided variant.
- We publicly release STUBCODER and the benchmark to facilitate future research in this area. The dataset is available at https://doi.org/10.5281/zenodo.7816758.

The remainder of this paper is organized as follows: Section 2 formulates the stub code synthesis problem with a motivating example and highlights the technical challenges. Section 3 presents the design and implementation of STUBCODER. Section 4 describes our evaluation of STUBCODER on 59 test cases collected from 13 open-source projects. Section 5 discusses the related work. Section 6 concludes the paper and points out possible future work.

2 Motivating Example & Problem Formulation

In this section, we leverage the example in Listing 1 to illustrate how we formulate and address the problem of stub code generation and repair for unit tests.

2.1 Formulation of Unit Test Cases

Unit test cases are commonly executed in three phases, following the AAA pattern [56] (i.e., Arrange, Act, Assert). First, the Arrange phase sets up the test environment, which includes the setup of dependencies with mock objects. Next, the Act phase exercises the CUT by invoking its methods. Finally, the Assert phase checks whether the CUT produces the expected test outputs.

We represent a test case as a tuple $\tau = \langle V, S, E, A \rangle$. Such representation is in line with the AAA pattern:

```
1 // Stub ode written by developers
2 when(endPoint.getPath("health")).thenReturn("/actuator/health");
3
4 // Stub code synthesized by StubCoder (Repair Mode)
5 String var0 = "/actuator/health";
6 doReturn(var0).when(pathMappedEndpoints).getPath(any());
```

Listing 2. Stub Code for Subject #36

- **Arrange Phase:** V is the set of variables that are used in E and A, and S is the stub code that specifies the behaviors of mock objects in V.
- **Act Phase:** E represents the bytecode instructions that exercise the CUT.
- **Assert Phase:** A is the test oracle, including mocking calls.

Take the test case in Listing 1 as an example. V contains two mock objects dao and user (Lines 5–6). S contains the stub code (the highlighted region) that sets up the behavior of the mock objects in M. For example, Lines 7–8 set the behavior of dao and specify that its method getPasswordHash should return the SHA-1 digest of string "bar". E contains Lines 14–15 that exercise the login function of the CUT LoginService. A contains Lines 18–19, which check whether the login function performs as expected using a mocking call (Line 18) and a JUNIT assertion (Line 19).

2.2 Problem Statement

Following our formulation of unit test cases, we define our stub code synthesis problem in two application scenarios.

Scenario #1: Generation Mode. Given an incomplete test case $\tau = \langle V, \varnothing, E, A \rangle$ without stub code, generate S such that $\tau = \langle V, S, E, A \rangle$ passes (i.e., all the oracle assertions in A pass without uncaught exceptions).

In this scenario, our technique helps developers to develop test cases that are independent of their test dependencies. For example, in Listing 1, we can synthesize the stub code in the highlighted region given the remaining lines such that the oracle assertions at Line 18 and Line 19 hold. When developers are creating a new test case, they can simply instantiate the CUT with mock dependencies and finish the remaining parts without needing to consider the possible interactions between the CUT and the dependencies. After that, they can launch STUBCODER to synthesize the stub code to complete the test case. We target at regression testing in this scenario, where we assume that the current system is correct and try to capture regressions in future versions.

Scenario #2: Repair Mode. Given an obsolete test case $\tau = \langle V, S_{bk}, E, A \rangle$ containing broken stub code S_{bk}, synthesize S to replace S_{bk} such that $\tau = \langle V, S, E, A \rangle$ passes.

In this scenario, our technique helps developers to repair test cases whose stub code is broken due to program or library changes. For example, when the stub code in the highlighted part is broken because of code updates in User, UserDao, or LoginService, developers can specify the code lines that contain the broken stub code and STUBCODER can replace the broken stub code with a synthesized one that can be compiled and can make the test pass. Compared with scenario #1, which synthesizes stub code from scratch, we leverage the information in the broken stub code S_{bk} to guide the synthesis in scenario #2. In this scenario, we target at the repair the stub code that are broken dur to refactoring or library upgrades. It need developers to decide whether the stub code needs to be repaired.

In both modes, STUBCODER can be implemented as an IDE plugin. In generation mode, developers can place the cursor at where the stub code needs to be generated. In repair mode, developers

can select the obsolete stub code that needs to be repaired. Developers can place the cursor at where the stub code need to be generated or selecte the obsolete stub code that need to be repaired. After that, they can launch StubCoder via a menu item provided by the plugin. The stub code is synthesized to facilitate regression testing based on the current program version. The test case with synthesized stub code captures the implemented behavior of the CUT, and helps to detect regression bugs in future versions of the CUT. It is important to clarify that StubCoder does not guarantee the semantic equivalence between the synthesized stub code and developer-written stub code. In the context of mocking, the stub code helping the test case achieving the same adequacy may not need to be syntactically or semantically similar. As an example, Listing 2 shows the stub code written by developer and synthesized by StubCoder. First, the developer-written stub code and the synthesized stub code are syntactically different. The developer-written stub code is using the API when(...).thenReturn(...) while the synthesized stub code is using the API doReturn(...).when(...). Also, the semantics of the two stub codes are not exactly the same. The developer-written stub will return "/actuator/health" only when the method getPath is invoked with an argument "health" while the synthesized stub code will return "/actuator/health" when the invocation is done with any argument. Although they are neither syntactically nor semantically the same, the test cases with both stub codes execute exactly the same set of instructions, traverse exactly the same execution paths, and kill exactly the same set of mutants as shown in our evaluation (Table 5). This is because in that test case, the method getPath will only be invoked with argument "health". In other words, the two stub codes are semantically equivalent in the context of that specific test case.

Indeed, StubCoder has available only the CUT executions E and assertions A to guide the generation and repair of stub code. Such available information is unlikely to be a complete specification of the behavior of the test. However, assertions should predicate of the salient expected behaviours of the test that makes the test pass or fail. In our evaluation (Section 4.5), we conjecture that obtaining stub code that fulfills such behaviors (i.e., it makes the assertions pass) would be enough to achieve the same (or similar) test adequacy with the ground-truth stub code.

2.3 Technical Challenges

It is challenging to synthesize a stub code, say S, to pass an input test τ due to the huge search space of possible candidates that can be generated for τ. This is because a stub code candidate is free to stub any method of any mock object in M for an arbitrary number of times, and to return any value or throw any exception for each stub call. Even by bounding the number of lines of code of the generated stub code (50 in our experiments), the search space is too huge to exhaustively explore. However, only specific stub code S can make τ pass. S should stub the correct set of methods with proper values so that E executes without exceptions and all the constraints in A are satisfied. Take the test case shown in Listing 1 as an example. Lines 14–15 create a LoginService object with mock object dao and invoke login method with username "foo" and password "bar". The test case passes only when satisfying two oracle assertions: (1) the findUser method is called twice (the mocking call at Line 18), and (2) the loginResult returned by login is successful (the assert statement at Line 19). Lines 23–35 show the implementation of the login method. In this method, dao.findUser will be called twice only if it throws a TimeoutException when it is first called (executing Line 28), and returns a User object when it is called for the second time (breaking the loop at Line 25). The LoginResult will be successful only if the getPasswordHash of the User object returns the SHA-1 checksum of password "bar" (passing the condition at Line 31). Lines 7–11 show the specific stub code that makes the pass.
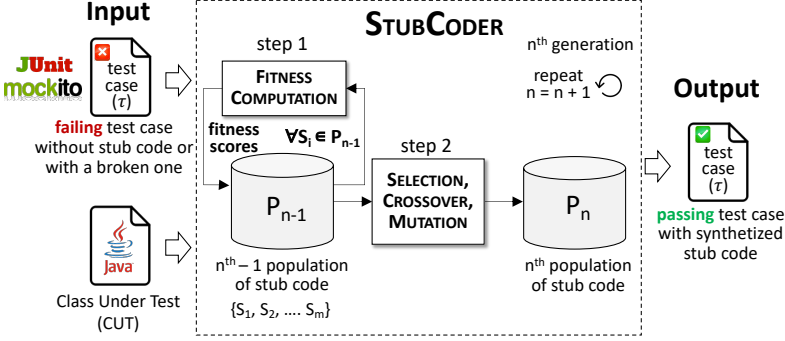
Fig. 1. Overview of StubCoder

It is infeasible to identify the test-passing stub code $\mathcal{S}$ by randomly or systematically exploring all the possible stub code candidates. To address this problem, we propose to use an evolutionary algorithm to guide the synthesis of the stub code and search for $\mathcal{S}$.

Key Idea. The evolutionary algorithm searches for the test-passing stub code $\mathcal{S}$ by generating new candidate stub code via crossover and mutations of existing ones in a guided manner. It guides the search by a fitness function that evaluates the distance between an arbitrary S and a passing stub code. As discussed in Section 2.2, $\mathcal{S}$ is the set of stub code that can pass τ. In other words, the stub code should (1) make the code in E executable, and (2) satisfy all oracle assertions in A. Based on this observation, we propose a multi-objective fitness function: with an arbitrary S, we integrate it with τ and capture (1) the percentage of bytecode instructions in E that can be successfully executed, and (2) the percentage of oracle assertions in A that can be satisfied, and (3) the distance between the value outputted by S and its expected value for an unsatisfied oracle assertion. Intuitively, S is closer to pass τ if it can make more bytecode instructions in E executable, satisfy more oracle assertions in A, and for the unsatisfied oracle assertions, the outputted value is closer to the expected value.

3 StubCoder

Figure 1 shows the logical architecture of StubCoder. The input is a test case without stub code ($\tau = \langle V, \varnothing, E, A \rangle$) or with a broken one ($\tau = \langle V, S_{bk}, E, A \rangle$) and the corresponding CUT. The output is the test case with a synthesized stub code that makes the test pass. Specifically, StubCoder implements a population-based evolutionary algorithm that guides the search for stub code using a multi-objective fitness function, as discussed in Section 2.3. At each generation, StubCoder evolves a population of stub code candidates until it finds one that can pass the test or the budget runs out. Figure 1 shows the process of producing P_i the population at the ith generation. First, StubCoder computes the fitness score for each candidate individual (stub code) $S \in P_{i-1}$. Then, it performs selection, crossover, and mutation to obtain the new population P_i. In particular, the selection phase selects two parent individuals from P_{n-1}. Individuals with higher fitness scores are more likely to be selected. The crossover phase combines the parents' genetic material (code elements in our case) to produce two offspring individuals. The mutation phase applies random mutations to the offspring individuals and adds them to P_i. These three phases repeat until P_i is full. In the following, we present the fitness function and explain how we adapt the selection, crossover, and mutation phases for the problem of stub code generation and repair.

3.1   Fitness Function

In this paper, we formulate the synthesis of stub code as a multi-objective optimization problem (MOOP) [44, 49] with three objectives. These objectives take into account the runtime behaviors of the Act and Assert phases of a test case. Each of them focuses on a particular aspect of the runtime behavior of the test case τ with the candidate stub code S.

Stub Utilization (*SU*). A given stub code S can have multiple stub calls to specify the behaviors of the mock objects. However, not necessarily all of the specified behaviors will be used during the Act phase. For example, the CUT might not invoke a stubbed method or the argument does not match. Such unused stub calls do not affect the behavior of the test, and mutating its return value has lower chances to make the test pass. Therefore, we define stub effectiveness (SU) of a stub code S based on the number of stub calls that are used by the CUT during the Act phase, denoted by $used(S)$.

$$SU(S) = \tanh\left(\frac{1}{C}used(S)\right)$$

The hyperbolic tangent (tanh) normalizes the value to $[0, 1)$. Since the curve of tanh is more steep in the interval $[0, 1)$ than in $[1, \infty)$, we divide the integer counter by a constant $C > 1$ to make use of the range $[0, 1)$. In our experiment, we chose $C = 10$.

Exercise Coverage (*EC*). Apart from oracle assertion violations, a test case fails when E invokes the CUT and the CUT throws an uncaught exception. Such uncaught exceptions are caused by the incorrect behaviors specified by the stub code. In general, a stub code that does not lead to uncaught exceptions when the test invokes the CUT is preferred, compared to one that does. As such, we define the exercise coverage (EC) of a stub code S as the ratio of the executed bytecode instructions in E.

$$EC(S) = \frac{|\{e \in E, e \text{ is executed}\}|}{|E|}$$

EC penalizes the individuals with an early failure of E due to incorrect behaviors specified in the stub code.

Assertion Status (*AS*). AS is derived from the runtime behavior in the Assert phase where the test executes the assertions in A. It is computed from the score of each of the assertion oracles in A by taking their average.

$$AS(S) = \frac{1}{|A|} \cdot \sum_{a \in A} score(a)$$

The score of each assertion oracle is a number in the range of $[0, 1]$, indicating how likely the assertion oracle is satisfied. It is defined as the following.

$$score(a) = \begin{cases} 1.0 & a \text{ is satisfied} \\ 1.0 - d(a.\text{expected}, a.\text{actual}) & a \text{ is } \texttt{assertEquals} \text{ and } a \text{ is failing} \\ 0.0 & \text{otherwise} \end{cases}$$

Specifically, for JUnit `assertEquals` assertions, we measure the distance between the expected and actual values to estimate how far it is from being satisfied, which is similar to the branching

```
1  @Test
2  public void should_use_application_uri() {
3      var endpoint = mock(PathMappedEndpoints.class);
4      when(endpoint.getPath("health")).thenReturn("/actuator/health");
5      var factory = new CFApplicationFactory(..., endpoint, ...);
6      var app = factory.createApplication();
7      assertEquals("base_url/actuator/health", app.getHealthUrl());
8  }
9
10 // CUT
11 public Application createApplication() {
12     return Application.builder().setHealthUrl(
13     "base_url" + endpoint.getPath("health")).build();
14 }
15
16 //Candidate stub code
17 S_a:
18     when(endpoint.getPath("health")).thenReturn("random");
19 S_b:
20     when(endpoint.getPath("health")).thenReturn("/actuator/hea");
```

Listing 3. A Test Case Adapted from Spring Boot Admin [27]

condition distance [55].

$$d(x, y) = \begin{cases} \tanh\left(\frac{|x-y|}{|x|}\right) & x, y \text{ are numeric types} \\ \tanh\left(\frac{Lev(x,y)}{|x|}\right) & x, y \text{ are strings} \\ d(str(x), str(y)) & x, y \text{ are complex objects} \end{cases}$$

where $Lev(x, y)$ is the Levenshtein distance [43]. Such a distance function considers the actions of insertions, deletions, and substitutions, which is in line with our mutation operators. For complex objects, function $str(x)$ serializes an object x into a string in a deep-copy manner [8]. Specifically, it recursively converts the fields of complex objects into string representations. This is because complex objects are often equated based on the values of its fields. Therefore, we chose such an strategy to approximate the distance between two complex objects. We implemented it using ReflectionToStringBuilder provided by the Apache Commons library [25]. We fall back the denominator to 1.0 if it is zero to avoid division-by-zero error and further normalized the result into the range of $[0, 1)$ with the hyperbolic tangent.

AS can provide additional guidance to generate values that can satisfy oracle assertions specified with assertEquals. For example, Listing 3 shows a code snippet adapted from a test in open-source project Spring Boot Admin [27]. It tests the creation of an Application object from a factory class CFApplicationFactory (Lines 5–6). The factory class sets up API end-points based on the information in an input PathMappedEndPoints that encapsulates a map from end-point names to their URLs. In Lines 12–13, the factory sets the application's HealthUrl by concatenating the base URL with the health URL encapsulated in endpoint by invoking its getPath method. The oracle assertion of the test case verifies whether HealthUrl of the created application equals the string "base_url/actuator/health" (Line 7). To make this test pass, the getPath method should be stubbed to return "/actuator/health" when it is invoked with argument "health". SU, and EC cannot guide the generation of these specific values. For example, candidate stub code S_a stubbing getPath to return "random" (Lines 17–18) and S_b stubbing getPath to return "/actuator/hea" (Lines 19–20) will achieve the same fitness score with only SU and EC. Both candidates can make the test execute to Line 6 but violate the oracle assertion at Line 7. However, returning "/actuator/hea" is much closer to passing the test as the returned string is much more similar to the expected value "/actuator/health". This difference can be captured by AS.

$$S ::= Elem^*$$
$$Elem ::= VarDef \mid StubCall$$
$$VarDef ::= v \leftarrow Expr$$
$$Expr ::= Literal \mid \mathsf{Array}(v^*) \mid API(v^*) \mid Mock$$
$$StubCall ::= \langle v, m, ArgMatcher^* \rangle \rightarrow Reaction$$
$$ArgMatcher ::= \mathsf{Any} \mid \mathsf{Eq}(v)$$
$$Reaction ::= \mathsf{Return}(v) \mid \mathsf{Throw}(v)$$

Fig. 2. Grammar of Synthesized Stub Code S.

Fitness Computation. In this paper, we designed a dominance based fitness computation approach. This is because, in our scenario, the three objectives are of different importance. Based on the execution order of the arrange, act, and assert phases, there is a natural order of the three objectives: SU, EC, and AS. Our rationale is that individuals who perform better (the functions have higher values) in later phases are more likely to converge to the test passing-stub code because in a test case, a later phase depends on the outcome of the former phases. For example, the assert phase will be executed only when the arrange and act phases executed successfully. Therefore, we define the dominance relationship $>$ between as follows. For two stub codes S_1 and S_2, $S_1 > S_2$ if any of the following holds:

- $AS(S_1) > AS(S_2)$
- $AS(S_1) = AS(S_2) \wedge EC(S_1) > EC(S_2)$
- $AS(S_1) = AS(S_2) \wedge EC(S_1) = EC(S_2) \wedge SU(S_1) > SU(S_2)$

During selection, for two individuals (i.e., stub code) S_1 and S_2, we favor S_1 if $S_1 > S_2$.

3.2 Representation of Stub Code

This section defines the possible stub code S that STUBCODER is able to synthesize (the possible individuals of a population). It also describes how STUBCODER represents an individual. Specifically, a candidate stub code can be constructed by any of the possible strings on the context-free grammar shown in Figure 2. Specifically, we represent S as a finite sequence of code elements, each of them can be either a *variable definition* or a *stub call*. A variable definition constructs a value and stores it in a variable. Then, a stub call can associate such variables with method calls on the mock objects. These code elements in the stub code work together to specify the behavior of the mock objects. By default, we set the length limit of S to 50, which is adequate for most of the stub code in practice (as shown in Table 3, the length of developer written stub code are less than 50).

Variable Definition. A variable definition $v \leftarrow Expr$ defines a new variable v and initializes it with $Expr$. The $Expr$ can be a literal value in JAVA [28], an array of previously defined variables [1], or an API call. Specifically, an API call can be either a method call, a constructor call, or a field access, which also takes previously defined variables as arguments. In addition, $Expr$ can be the creation of a mock object. This enables us to synthesize the mock objects that may be absent from the input (e.g., user in Listing 1).

---

[1]The synthesized stub code is inserted before the first reference of the mock objects in the test case. All the variables defined before the stub code can be used in the synthesized stub code.
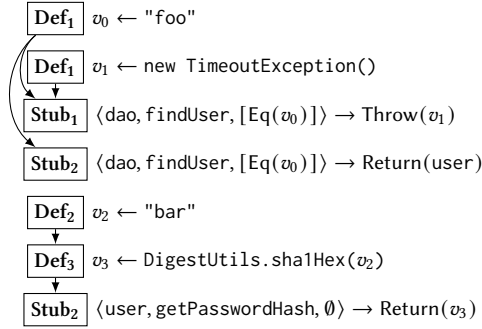
$\boxed{\textbf{Def}_1}$ $v_0 \leftarrow$ "foo"

$\boxed{\textbf{Def}_1}$ $v_1 \leftarrow$ new TimeoutException()

$\boxed{\textbf{Stub}_1}$ $\langle \mathtt{dao}, \mathtt{findUser}, [Eq(v_0)] \rangle \rightarrow \mathrm{Throw}(v_1)$

$\boxed{\textbf{Stub}_2}$ $\langle \mathtt{dao}, \mathtt{findUser}, [Eq(v_0)] \rangle \rightarrow \mathrm{Return(user)}$

$\boxed{\textbf{Def}_2}$ $v_2 \leftarrow$ "bar"

$\boxed{\textbf{Def}_3}$ $v_3 \leftarrow$ DigestUtils.sha1Hex(v_2)

$\boxed{\textbf{Stub}_2}$ $\langle \mathtt{user}, \mathtt{getPasswordHash}, \emptyset \rangle \rightarrow \mathrm{Return}(v_3)$

Fig. 3. Representation of the Stub Code in Listing 1

Stub Call. A stub call specifies the reaction for a certain method call received by a mock object when the argument matcher matches all the arguments. The reaction can be either Return(v), which returns the value referenced by the variable v, or Throw(v), which throws the exception referenced by the variable v. There can be multiple stub calls matching the same method call on the same mock object. Their reactions will be executed in the order that they appear.

Figure 3 illustrates our representation of the stub code in Listing 1. There are four variable definitions and three stub calls. As specified by $Stub_1$ and $Stub_2$, dao.findUser will throw a TimeoutException for the first call and return user for the second call. The return value of user.getPasswordHash will be the SHA-1 digest of the string "bar" stored in v_3. The arrows indicate the def-use dependencies among these code elements.

The grammar in Figure 2 is based on the APIs provided by the MOCKITO [12] framework, which is the most popular mocking framework for JAVA. It can be adapted to support the syntax of other object-oriented programming languages and mocking frameworks. Mocking frameworks tend to provide APIs with similar functionalities to aid the development. For example, a *StubCall* can be mapped to a expect(...) call in EASYMOCK [11] or a Mock<T>.Setup(...) call in MOQ4 [13]. Return(v) can be mapped to a andReturn(...) call in EASYMOCK or a MethodCall.Return(...) call in MOQ4. In our experiment, we implemented STUBCODER in JAVA using MOCKITO.

3.3 Evolutionary Algorithm

Algorithm 1 details the key steps in the evolution of stub code. It takes a test case with a void or broken stub code $\tau = \langle V, \emptyset, E, A \rangle$ or $\langle V, S_{bk}, E, A \rangle$ as input, and outputs a stub code S such that the $\tau = \langle V, S, E, A \rangle$ is passing. The evolution process is controlled by the population size N and generation budget MAX_GEN.

Symbol Pool. One challenge in synthesizing the stub code is to properly construct the return values. However, it is less efficient to start searching from default values (e.g., 0, null, an empty string) or random values. To address this challenge, we construct a symbol pool to provide heuristics for the search process. Specifically, the function CONSTRUCT-SYMBOL-POOL in Algorithm 1 extracts the literals and API calls from the τ and the CUT. It also includes the symbols in the broken stub code S_{bk}, if available. Such a constructed symbol pool contains useful values for synthesizing a test-passing stub code. After construction, the symbol pool B is passed to CROSSOVER-AND-MUTATION. Mutation operators can take the literals and API calls in B to generate variable definitions.

Algorithm 1: Evolution of Stub Code in STUBCODER

    Input: Input test case $\tau = \langle V, \varnothing, E, A \rangle$ or $\langle V, S_{bk}, E, A \rangle$
    Input: Population size N, generation budget MAX_GEN
    Output: Stub code S

1   $B \leftarrow$ CONSTRUCT-SYMBOL-POOL(τ) ;
2   $P \leftarrow$ CREATE-INITIAL-POPULATION (N, B) ;
3   $P \leftarrow$ FITNESS-COMPUTATION(P);
4   $gen \leftarrow 1$;
5   **repeat**
6      $P' \leftarrow$ ELITISM-SELECTION(P) ;
7   **while** $|P'| < N$ **do**
8          $\langle p_1, p_2 \rangle \leftarrow$ SELECT-PARENTS(P);
9          $\langle o_1, o_2 \rangle \leftarrow$ CROSSOVER-AND-MUTATE(p_1, p_2, B);
10         $P' \leftarrow P' \cup \{o_1, o_2\}$;
11   **end**
12     $P \leftarrow P'$;
13     $P \leftarrow$ FITNESS-COMPUTATION(P);
14     $gen \leftarrow gen + 1$;
15 **until** $\exists S \in P, \tau = \langle V, S, E, A \rangle$ *passes* $\vee gen > MAX\_GEN$;
16 **return** S;

Initial Population. STUBCODER starts the synthesis process from an initial population. Specifically, the function CREATE-INITIAL-POPULATION returns a population of N stub code where each of them contains randomly generated code elements for the mock objects in M.

Elitism Selection. Before starting populating the new population, STUBCODER retains the best individuals. At Line 6 of Algorithm 1, the function ELITISM-SELECTION selects the top 1% of individuals with the highest fitness and brings them directly to the next generation. With elitism selection, STUBCODER avoids losing the best individuals in the next generation.

Parent Selection. At Line 8 of Algorithm 1, STUBCODER selects two parent individuals to produce the offspring. In the function SELECT-PARENTS, we leverage tournament selection [39] to select the parents. Tournament selection is widely used in genetic programming [24, 29] because it has less stochastic noise compared with other selection methods [7]. Specifically, it randomly chooses K individuals and runs a tournament among them, after which the winner is chosen. In this paper, we choose $K = 2$ to mitigate premature convergence and the local optimum problem [33, 34]. Thus, we run two tournaments to get two parents $\langle p_1, p_2 \rangle$.

Crossover and Mutation. The function CROSSOVER-AND-MUTATION exchanges the genetic materials of two parents $\langle p_1, p_2 \rangle$ and produces two offspring individuals $\langle o_1, o_2 \rangle$, which are then mutated to introduce new genetic materials.

First, we exchange the stub calls that stub a mock object in M because they directly contribute to the outcome of the test case. Specifically, we gather all such stub calls from p_1 and p_2, and copy each of them to o_1 and o_2 with probability 50%. In the stub code, a stub call relies on other code elements to function (e.g., the variable definition for its return value). Therefore, when copying a stub call, we perform a backward slicing [22] to obtain all its dependencies and copy all dependent

code elements to the offspring. For example, in Figure 3, when copying $Stub_2$, we bring together Def_3 and Def_2 since they are required to define v_3, which is the return value of $Stub_2$.

Next, the two offspring individuals o_1 and o_2 are mutated by one of the following mutation operators (randomly chosen with uniform probability).

- **Inserting a Code Element.** We randomly generate a stub call and a variable definition, and insert it into the stub code. Variable definitions are generated by randomly choosing a literal or an API call from the symbol pool B.
- **Altering the Parameters.** Some code elements in the stub code take variables as their parameters. For example, in Figure 3, stub call $Stub_1$ takes v_1 to be the exception and variable definition Def_3 uses v_2 to perform an API call. We randomly choose a parameter for such code elements and replace it with another variable of the same type in the stub code.
- **Altering the Literals.** Some variable definitions in the stub code are numeric, string, or Boolean literals. For example, Def_2 in Figure 3 defines v_2 with a string literal. We apply a randomly chosen numeric or string operation (e.g., add/subtract a random value, alter a character) to the literal. For a Boolean literal, we simply flip it.
- **Swapping Two Code Elements.** We randomly choose two code elements and interchange them.
- **Dropping a Code Element.** We randomly remove a stub call or an unused variable definition from the stub code.

Mocking Decisions. The synthesized stub code can contain two types of mock objects. The mocking decision is made depending on how they are declared.

- **Type I Mock Objects.** For the mock objects declared by developers in the test cases, we follow the original mocking decision by the developers.
- **Type II Mock Objects.** During mutation, using a real object or a mock object is an alternative way of generating an object, and StubCoder will choose randomly between these alternative ways. For instance, when populating a variable of a complex type T, the mutation operator will randomly choose between using a generator of T (e.g., the constructors of T, the fields of type T, and the methods that return T), and a mocked version of T (i.e., mock(T.class)). Since the goal of StubCoder is to synthesize a stub code to pass the test, the fitness function will then prioritize the stub code that is more likely to make the test pass.

Stopping Criterion. As shown at Line 15 of Algorithm 1, we stop the algorithm if the best stub code makes the test pass or the maximum number of generations is reached.

4 Evaluation

This section presents the evaluation of StubCoder. Specifically, we aim to answer the following four research questions.

- **RQ1 (Stub Code Generation):** *Is StubCoder effective in generating stub code?*
- **RQ2 (Stub Code Repair):** *Is StubCoder effective in repairing obsolete test cases due to broken stub code? Does it outperform state-of-the-art program repair techniques?*
- **RQ3 (Effectiveness of the Fitness Function):** *Can the fitness function effectively guide stub code synthesis?*
- **RQ4 (Fidelity of the Synthesized Stub Code):** *To what extend does the stub code synthesized by StubCoder preserve the effect of the ground-truth stub code?*

RQ1 and RQ2 evaluate StubCoder in the two application scenarios (synthesis and repair of stub code, see Section 2.2). RQ3 evaluates the effectiveness of StubCoder's fitness function by comparing it with an unguided (random) strategy. RQ4 compares the fidelity of the stub code synthesized by

StubCoder with that written by developers. Specifically, we evaluate to what extent the test case with synthesized stub code can preserve the runtime effect of the test case with the ground-truth stub code by comparing their executed instructions, execution paths, and ability to kill mutants.

4.1 Evaluation Subjects

Benchmark Description. To answer the four research questions, we constructed a benchmark of 59 real-world test cases selected from 13 open-source projects (see Table 2). Each entry in the benchmark contains:

- The test case with the removed stub code, which is the input of StubCoder in RQ1.
- The broken version of the stub code, which is the additional input of StubCoder in RQ2.
- The ground-truth stub code written by developers that makes the test pass, which is used for comparison in RQ4.
- The production code and the dependent libraries, which are required to compile and run the test.

Project Selection. To build the benchmark, we searched on GitHub [30] for open-source Java projects and sorted the results by the number of stars, which is an indicator of popularity. We manually went through the top 150 projects to identify those meeting the four criteria below:

(1) It has at least 1,000 lines of Java code. This is to filter out small projects.
(2) It uses the Mockito framework [12] to simulate/verify the behaviors of test dependencies. This is because we implemented StubCoder based on Mockito, which is the most popular mocking framework for Java [41].
(3) It is not an Android project since Android is currently not supported by our implementation.
(4) It uses Maven or Gradle as build automation tools so that we can automate the dependency collection procedure.

We identified 40 candidate projects that satisfy our selection criteria.

Benchmark Preparation. For each candidate project, we automatically explored their commit history since 2018. We performed an AST-level diff using GumTree [19] between the two versions of each commit to locate changes to the Mockito stub code. The diff returned 2,295 code changes. Since preparing the benchmark requires intensive manual effort, we performed a pre-selection on the code changes. For each of the projects, we sampled at most 100 code changes, obtaining a total of 871 candidates. Then, we manually read the code diff and commit messages to understand the semantics of the change. We ignored the code changes that simply rename code elements (i.e., GumTree classifies the ASTs before and after the changes as isomorphic). This is because we do not regard such trivial code changes as the target application scenario of StubCoder. Repairing stub code in such cases can be easily achieved using the refactoring feature of modern IDEs.

After dropping the trivial cases and duplicate commits due to git branch merges, we retained 261 code changes. Each code change specifies two versions of a stub code: a broken one (before the change), and a correct one (after the change). To turn each code change into a benchmark entry, we performed the following procedure:

- We ran the Gradle or Maven build script to resolve the dependencies and compile the project.
- We rewrote each oracle assertion written with custom assertion frameworks into semantically equivalent JUnit assertions. Table 1 lists the rewritten rules applied by us. Specifically, the rules were drafted by one author and then independently validated by two other authors independently. One more author joined and resolved disagreements when they occurred. In

```
@Test
public void test_bk() throws Exception {
    CallableStatement cs = mock(CallableStatement.class);
    LocalDateTime LOCAL_DATE_TIME = LocalDateTime.now();
    TypeHandler<LocalDateTime> TYPE_HANDLER = new LocalDateTimeTypeHandler();
```

S_{bk}
```
    Timestamp TIMESTAMP = Timestamp.valueOf(LOCAL_DATE_TIME);
    when(cs.getTimestamp(1)).thenReturn(TIMESTAMP);
```

```
    assertEquals(LOCAL_DATE_TIME, TYPE_HANDLER.getResult(cs, 1));
    verify(cs, never()).wasNull();
}
```

Obsolete Test Case (4dfea24)

```
@Test
public void test_gt() throws Exception {
    CallableStatement cs = mock(CallableStatement.class);
    LocalDateTime LOCAL_DATE_TIME = LocalDateTime.now();
    TypeHandler<LocalDateTime> TYPE_HANDLER = new LocalDateTimeTypeHandler();
```

S_{gt}
```
    when(cs.getObject(1, LocalDateTime.class)).thenReturn(LOCAL_DATE_TIME);
```

```
    assertEquals(LOCAL_DATE_TIME, TYPE_HANDLER.getResult(cs, 1));
    verify(cs, never()).wasNull();
}
```

Ground Truth Test Case (963a8a5)

Fig. 4. Version Relationship between S_{bk} and S_{gt} in Project MB3

Table 1. List of Rewritten Assertions in Benchmark

Original Assertion	Rewritten JUnit Version
assertThat(x, equalTo(y))	assertEquals(y, x)
assertThat(x).isEqualTo(y)	assertEquals(y, x)
assertThat(x).isNotNull()	assertNotNull(x)
assertThat(x).isInstanceOf(Y.class)	assertTrue(x instanceof Y)
assertThat(x).isSameAs(y)	assertSame(y, x)
assertThatThrownBy(() -> x) .isInstanceOf(E.class).hasMessage(y)	E e = assertThrows(E.class, () -> x) assertEquals(y, e.getMessage())
assertThatThrownBy(() -> x) .isInstanceOf(E.class).hasMessageStartsWith(y)	E e = assertThrows(E.class, () -> x) assertTrue(e.getMessage().startsWith(y))

addition, we ran the test cases after rewriting the assertions to check that the test is still passing. This was done for 14 subjects in project AZK, SBA, JIB, GRC, ZKN, and SPB.
- We executed the test case to ensure that it passes with the correct stub code written by developers, since we will use it as the ground truth.
- We removed the stub code so that the test fails. This is to ensure that the stub code is required to pass the test.

We discarded a code change if we failed to perform any of the steps above on it. Finally, we constructed a benchmark of 59 test cases containing 167 mock objects collected from 13 projects. Each of the entries in the benchmark consists of two elements $\langle \tau_{bk}, \tau_{gt} \rangle$. $\tau_{bk} = \langle V, S_{bk}, E, A \rangle$ is the obsolete test case containing broken stub code S_{bk} and $\tau_{gt} = \langle V, S_{gt}, E, A \rangle$ is the fixed version of the test case, which contains the ground-truth stub code S_{gt} written by developers. Figure 4 shows an example of a benchmark entry in project MB3. The obsolete test case from version 4dfea24 contains broken stub code. Developers fixed the broken stub code in version 963a8a5 by modifying the stub code. Table 2 summarizes the benchmark.

Table 2. Demographics of Benchmark

Artifact ID	GitHub Project ID	LOC (Java)	Stars	# Test Cases	# Total Mock Objects
ADR	apache/druid	856K	11.9K	2	7
ADU	apache/dubbo	199K	37.6K	13	25
AHP	apache/hadoop	1,834K	12.7K	5	9
AZK	apache/zookeeper	116K	10.6K	1	2
APL	apolloconfig/apollo	52K	27.0K	8	40
SBA	codecentric/spring-boot-admin	18K	11.0K	2	2
JIB	GoogleContainerTools/jib	55K	11.9K	2	7
GRC	grpc/grpc-java	226K	9.9K	3	3
MB3	mybatis/mybatis-3	68K	17.4K	12	37
N4J	neo4j/neo4j	731K	10.2K	4	25
ZUL	Netflix/zuul	232K	12.0K	1	2
ZKN	openzipkin/zipkin	42K	15.5K	5	7
SPB	spring-projects/spring-boot	347K	62.1K	1	1
Total				59	167

4.2 RQ1: Stub Code Generation

Experiment Setup. To answer RQ1, we ran StubCoder on the 59 test cases without stub code in our benchmark to generate stub code for them. We ran StubCoder with population size $N = 200$, and set the generation budget $MAX\_GEN = 400$. We selected these parameters based on a few trial runs following previous work [51]. Due to the stochastic nature of evolutionary algorithms, we evaluated whether StubCoder can successfully synthesize the stub code in 10 repetitions. Our algorithm relies on a pseudo-random number generator when making random decisions. We chose 10 randomly-generated prime numbers as the random seeds for each repetitions, these random seeds are used across all the evaluation subjects. We also designed two alternative optimization strategies in addition to our dominance based approach for comparison.

- **Weighted Sum.** The first alternative optimization strategy is to combine these objectives using weighted sum. In this setup, we combined the three objectives into a single fitness function with different weights. With the same rationale as for the NSGA-II variant, we assigned higher weights for the functions measuring the later stages of test execution, following the powers of 2.

$$\textbf{fitness}(S) = 2^0 \cdot SU(S) + 2^1 \cdot EC(S) + 2^2 \cdot AS(S)$$

- **NSGA-II.** In addition, we tried out the most popularly adopted MOEA, NSGA-II [17]. NSGA-II employs a fast non-dominated sorting based on Pareto optimality [49] and crowding-distance based comparison. Such an approach will produce solutions that offer the best trade-off between competitive objectives [49].

Results. Column "StubCoder (G)" of Table 3 shows the results for each test case. Column "SR" shows the number of successful runs for each test case. For successful runs, we also report the generations taken, time taken (in seconds), and the size of stub code, in columns "Gen", "Time", and "$|S|$", respectively.

For 45 of 59 test cases, StubCoder successfully generated test-passing stub code in at least 5 of the 10 repetitions, which counts for 76% of the subjects. The median of time taken by all the successful syntheses is 182 seconds.

Column $|S|$ shows the length of the stub code (in terms of lines of code). StubCoder is able to synthesize non-trivial stub code. The type of variables in the stub code contains primitive types, strings, and complex objects. For the 48 subjects with at least one successful run, 30 of them contain

Table 3. RQ1 — RQ3: Comparison of the Success Rate in Different Setups

GT is for ground truth, which are the test cases written by developers. "StubCoder (G)", "StubCoder (R)", "NSGA-II", "Weighted Sum", and "Unguided" are StubCoder in generation mode, repair mode, with NSGA-II, with weighted sum, and random selection, respectively. $|M|$ is the number of mock objects in V in the test case. $|A|$ is the number of assertions (including verify assertions on mock objects) in the test case. $|S|$ is the size of the stub code. SR is the number of successful runs. Gen, Time, and $|S|$ are the number of generations taken, time taken (in seconds), and the size of stub code, respectively.

Subject		GT					StubCoder (G)					StubCoder (R)					NSGA-II					Weighted Sum					Unguided		
	$|M|$	$|A|$	$|S|$		SR	Gen	Time	$|S|$		SR	Gen	Time	$|S|$		SR	Gen	Time	$|S|$		SR	Gen	Time	$|S|$		SR	Gen	Time	$|S|$
N4J #1	1	1	3		10	9	511	3		10	13	749	3		4	335	26155	3		10	8	534	3		10	25	983	3
#2	5	4	26		10	24	1118	11		10	10	454	20		10	27	1670	4		10	14	861	9		10	37	1892	4
#3	9	5	11		0	-	-	-		2	342	16738	9		0	-	-	-		1	387	23048	18		0	-	-	-
#4	10	3	9		9	75	4949	17		8	92	5882	20		0	-	-	-		10	149	11294	39		1	307	20908	12
SPB #5	1	1	1		10	1	38	2		10	1	36	2		10	1	45	2		10	1	56	2		10	1	40	2
GRC #6	1	2	1		10	2	69	2		10	2	67	2		10	2	95	2		10	2	94	2		2	156	6254	2
#7	1	2	14		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-
#8	1	2	12		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-
MB3 #9	4	2	1		10	2	86	1		10	1	43	2		10	2	97	1		10	1	51	1		10	1	43	1
#10	2	2	1		10	2	82	1		10	2	85	2		10	1	54	1		10	2	96	1		10	1	40	1
#11	4	2	1		10	2	89	1		10	3	131	1		10	3	145	1		10	3	196	1		10	2	91	2
#12	4	2	1		10	2	81	3		10	2	85	1		10	2	95	2		10	2	136	4		10	1	46	1
#13	1	2	1		10	2	76	1		10	2	77	1		10	1	43	2		10	1	57	1		10	1	40	1
#14	4	2	1		10	2	89	1		10	1	42	1		10	5	237	1		10	1	68	1		10	2	90	2
#15	4	2	1		10	3	127	1		10	2	86	2		10	2	98	1		10	1	55	2		10	1	44	1
#16	4	2	1		10	2	85	1		10	3	128	1		10	4	203	1		10	4	218	1		10	1	44	1
#17	1	2	1		10	2	78	1		10	4	158	1		10	2	91	2		10	2	114	1		10	1	42	2
#18	4	2	1		10	2	85	1		10	5	219	2		10	2	99	1		10	3	159	1		10	1	45	1
#19	1	2	1		10	2	79	1		10	4	156	1		10	4	218	1		10	1	47	2		10	1	41	1
#20	4	2	1		10	5	210	1		10	2	86	1		10	3	145	1		10	2	111	1		10	1	46	1
ZKN #21	1	2	7		10	1	35	2		10	1	37	2		10	13	567	2		10	1	56	2		10	1	38	2
#22	2	1	8		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-
#23	1	2	7		10	25	953	3		10	20	745	3		0	-	-	-		10	14	313	5		0	-	-	-
#24	2	1	7		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-
#25	1	2	10		10	10	381	6		10	18	689	4		0	-	-	-		10	20	428	3		0	-	-	-
APL #26	5	3	3		10	2	78	2		10	3	121	2		10	3	133	2		10	2	131	4		9	61	2646	3
#27	5	4	3		10	2	79	5		10	2	80	6		10	2	90	4		10	2	103	3		10	2	86	3
#28	6	2	10		10	111	5116	9		9	85	3895	9		7	158	8409	10		9	84	6597	11		1	324	16501	10
#29	4	2	10		1	65	2777	12		4	311	13022	18		1	143	7282	17		0	-	-	-		0	-	-	-
#30	6	5	20		5	80	3673	11		5	233	10805	6		5	151	7906	7		7	279	14574	19		0	-	-	-
#31	6	11	24		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-
#32	4	4	10		10	1	41	2		10	1	39	2		10	1	45	10		10	1	53	6		10	1	52	8
#33	4	3	12		10	26	1196	3		10	22	949	5		9	98	4980	4		10	17	918	5		0	-	-	-
ZUL #34	2	1	2		10	1	39	1		10	1	37	1		10	6	273	1		10	1	51	1		10	1	51	3
SBA #35	1	3	1		0	-	-	-		10	17	639	2		1	72	3095	2		0	-	-	-		0	-	-	-
#36	1	3	1		0	-	-	-		10	5	186	2		0	-	-	-		0	-	-	-		0	-	-	-
JIB #37	5	18	13		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-
#38	2	1	15		0	-	-	-		2	379	15510	32		0	-	-	-		0	-	-	-		0	-	-	-
AZK #39	2	20	11		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-		0	-	-	-
AHP #40	1	1	9		10	1	43	4		10	1	43	2		10	1	50	2		10	1	62	8		10	1	44	2
#41	1	3	9		10	7	312	6		10	8	364	7		10	5	237	10		10	7	380	7		10	3	141	10
#42	3	3	20		10	1	66	6		10	1	66	6		10	2	143	6		10	1	87	13		10	1	66	5
#43	2	3	7		8	27	1252	14		10	17	780	34		7	100	4827	14		10	30	1695	21		10	77	3577	13
#44	2	4	8		10	32	1487	13		10	75	3553	12		4	203	9984	9		10	29	1648	13		10	56	2831	13
ADR #45	1	1	1		2	232	12684	2		0	-	-	-		0	-	-	-		1	169	10534	2		0	-	-	-
#46	6	6	16		10	5	482	9		10	5	391	8		10	4	257	4		10	4	506	10		10	4	212	5
ADU #47	2	1	3		7	109	4404	17		2	174	7052	15		0	-	-	-		8	112	5413	18		0	-	-	-
#48	3	3	10		9	41	1805	9		8	71	3234	10		9	120	5712	21		9	91	4713	20		0	-	-	-
#49	1	2	4		10	54	2118	7		10	25	966	9		0	-	-	-		10	72	3411	6		0	-	-	-
#50	1	2	4		10	80	3153	8		10	15	552	10		0	-	-	-		10	62	3320	8		0	-	-	-
#51	2	1	7		6	46	1878	12		3	73	3053	19		6	303	14601	8		9	36	1844	7		0	-	-	-
#52	1	1	2		10	1	35	1		10	1	35	1		10	1	38	1		10	1	59	1		10	1	37	1
#53	1	2	9		10	77	3191	14		10	50	2143	12		1	384	15915	15		10	88	4661	29		0	-	-	-
#54	8	2	14		8	61	2733	28		8	183	8284	33		0	-	-	-		5	135	7114	23		0	-	-	-
#55	1	1	2		10	1	36	1		10	1	35	1		10	1	40	1		10	1	60	1		10	1	38	1
#56	1	1	5		10	43	1848	9		10	32	1291	14		1	378	16034	9		10	29	1512	10		10	196	7896	15
#57	1	2	7		1	223	11755	7		3	373	12003	6		0	-	-	-		3	136	7933	6		0	-	-	-
#58	1	1	3		10	25	878	7		10	27	1047	7		5	215	8896	7		10	26	1400	6		9	52	2416	12
#59	2	1	7		10	2	73	8		10	2	72	5		5	79	2691	2		10	3	171	8		10	4	175	16
Success Rate > 50%					76%					76%					58%					76%					54%			

The value of Y-axis is the average success rate over the 10 repetitions at the generation budget specified by the X-axis.

Fig. 5. Comparison of the Success Rate between Experiment Setups

complex objects in the synthesized stub code. For most of the test cases, the length of the stub code synthesized by STUBCODER is slightly longer than the ones written by developers. This is because STUBCODER does not inline the variables that are used only once in the synthesized stub code, which is often done by developers (e.g., In Listing 3, developers put the String literal right in the thenReturn). Such a refactoring can be done trivially by using some refactoring tools. For some test cases (e.g., #2, #30, and #32), STUBCODER synthesized much fewer lines of code, yet obtained a test-passing stub code. We found that for these test cases, developers copied the same stub code across several test cases, creating redundant stub calls. Such a bad practice can complicate future maintenance of the stub code. In comparison, the stub code synthesized by STUBCODER will be easier to maintain.

STUBCODER is more likely to be successful for subjects with simple stub code. For instance, the stub codes in project MB3 mostly comprise a single line and do not contain complicated string values. STUBCODER is more likely to fail when a subject needs a complicated stub code to pass. For example, in project JIB, there are multiple stub calls in the developer-written test containing complicated string values. Although our fitness component *AS* can capture the edit distance between the expected value and the actual value in the assertions, it cannot help when the string value returned by the stub code does not flow directly to the assertions (e.g., used as branch conditions). Also, for project ZKN, the developer-written stub code contains a custom implementation that mutates the variables outside the stub, which is beyond the capability of STUBCODER.

Figure 5 shows a comparison of the average success rate versus generation budget between different optimization strategies. The dominance-based approach adopted by STUBCODER performs similarly as weighted sum while it requires less parameter tuning effort. NSGA-II performs much poorer than dominance-based approach and weighted sum. The reason for this performance drop is that, NSGA-II aims to produce solutions that offer the best trade-off between competitive objectives [49], which will treat all the optimization objectives as of equal importance. However, in our scenario, the three objectives measures the quality of candidate stub code in different stages of test execution and they are of different importance. In this case, we can observe a large performance gap between NSGA-II and dominance-based approach.

> **♥ RQ1 in Summary:** STUBCODER successfully synthesizes the stub code for 76% of the test cases in our benchmark in at least half of the repetitions. Optimization strategies that consider the importance of each objectives help STUBCODER achieve better performance.

4.3 RQ2: Stub Code Repair

Experiment Setup. The application scenario of RQ2 is stub code repair. To answer RQ2, we followed the same setup as in RQ1, with the only difference that we fed StubCoder with the broken stub code so that it could make use of its tokens to construct the symbol pool.

StubCoder aims to generate and repair stub code in unit tests. It is the first technique of its kind. Existing test script repair techniques either repair oracle assertions [15, 16], repair GUI test scripts [9, 26, 48], or focus only on CUT calls [35, 40]. These techniques cannot repair obsolete stub code. As a result, we do not use them as baselines in our evaluation. Instead, we selected state-of-the-art program repair techniques as our baselines. Specifically, we selected the techniques using the following criteria:

- It is most recently published at a peer-reviewed venue.
- It has an artifact that works for Java projects.
- It needs only faulty code and test cases as input.

Following these criteria, we selected two state-of-the-art program repair techniques: Arja [57] and the Cardumen mode [37] of Astor, and we applied the two baselines on our subjects. Since the stub code being repaired is in the form of test code, which is not supported by these baselines, we make the following adaptations for our subjects.

- Since the two baselines can only repair application code, we migrate the test case that contains the broken stub code (together with its dependencies) to the application code directory.
- The baselines rely on fault localization techniques to find repair candidates. However, in our scenario, the statements to be repaired are already known. Therefore, we implement a fault localizer that returns the statements containing the obsolete stub code as faulty locations. This can force the baselines techniques to repair only the stub code.
- For each migrated test case, we create a simple test case to trigger it. We specify these simple test cases as the failing tests when applying the two baselines.

After applying the adaptation, we run the two baselines with their default configurations with a time budget of six hours.

Results. As shown in Table 3 (Column "StubCoder (R)"), StubCoder successfully repaired 76% of the test cases in our benchmark in no fewer than 5 repetitions. There are 30 subjects where StubCoder synthesized stub code with complex objects. As shown in Figure 5, StubCoder took fewer generations to find a test-passing stub code. With the help of the tokens in the broken stub code, StubCoder is able to produce test-passing stub code for the subjects where it fails in generation mode. Take subject #36 as an example, a complicated string "/actuator/health" must be stubbed to pass the test. During code evolution, the signature of the method being stubbed changed, and the stub code was broken. Nevertheless, the string literal in the broken stub code is still useful for StubCoder and enables it to converge quickly to the stub code that makes the test pass. As shown in Listing 2, StubCoder synthesized a two-line stub code to pass the test by reusing the literal string in the broken stub code, which was done in only four generations. In comparison, without the help of the tokens, it is hard for StubCoder to synthesize such a complicated string literal from scratch and therefore, StubCoder failed to synthesize a test-passing stub code in generations mode.

For the two state-of-the-art baseline techniques, they failed to repair the stub code in any of our evaluation subjects. There are two reasons for the poor performance achieved by the baseline techniques. First, for 35 of the 59 subjects, the broken stub code leads to a compilation error. The baseline techniques require compiling tests to run and therefore, are not applicable to these

subjects. Second, for the remaining 24 subjects, they failed to repair the stub code because they lack awareness of the semantics of the APIs in the mocking frameworks (mocking APIs). Without understanding the mocking APIs, it is difficult for such techniques to find a test-passing stub code by randomly mutating the AST nodes.

> **⚲ RQ2 in Summary:** STUBCODER successfully repairs the stub code for 76% of the test cases in our benchmark in no fewer than 5 repetitions. The tokens in the broken stub code can help reduce search effort and synthesize shorter stub code in some cases. State-of-the-art program repair techniques cannot repair the stub code in any of the evaluation subjects.

4.4 RQ3: Effectiveness of Fitness Function

Experiment Setup. RQ3 aims to evaluate the contribution of the fitness function to steering the search for stub code. Towards this goal, we constructed a variant of STUBCODER with random selection, which conducts the search process without the guidance of the fitness function. Enumerating and (uniformly) sampling the whole search space would have been the ideal random baseline. However, it is infeasible due to the huge size of the search space. As such, we opted for a variant of STUBCODER that uses the same crossover and mutation operations to explore the search space, but without any guidance by the fitness function. We ran this variant of STUBCODER with the same configurations as in RQ1.

Results. Column "Unguided" of Table 3 shows the performance of the unguided variant of STUBCODER. Without the support of the fitness function, the unguided variant only successfully synthesizes stub code for 54% of the test cases in our benchmark in no fewer than 5 repetitions, which is less than the generation mode and the repair mode. In general, when the unguided variant successfully synthesizes stub code, it takes significantly more generations (e.g., #4, #28, and #56) than the guided version of STUBCODER. For five test cases, only the unguided variant of STUBCODER fails to synthesize the stub code (e.g., #30 and #33). Interestingly, four of them have multiple oracle assertions in their test oracle. For such test cases, STUBCODER with fitness guidance can successfully synthesize the stub code because the fitness function examines the status of each assertion in the test oracle, and thus can prioritize the candidate stub code that can satisfy some of the assertions. Such results show that our fitness function can effectively guide the search for test-passing stub code.

> **⚲ RQ3 in Summary:** STUBCODER outperforms its unguided variant in both the generation and repair modes. Our fitness function provides useful guidance for synthesis of stub codes.

4.5 RQ4: Fidelity of Synthesized Stub Code

Experiment setup. RQ1 and RQ2 evaluate the effectiveness of STUBCODER in generating and repairing stub code that makes the developer-specified assertions pass. Different from them, RQ4 evaluates the fidelity of the stub code synthesized by STUBCODER with respect to the ground-truth stub code. Specifically, we evaluate to what extent the test cases with synthesized stub code can preserve the runtime behavior of the test case with ground-truth stub code. For each of the test cases in which STUBCODER successfully synthesizes stub code in at least one run, we prepared $\tau_{gt} = \langle V, S_{gt}, E, A \rangle$ with the stub code written by developers, and $\tau_s = \langle V, S, E, A \rangle$ with the stub code synthesized by STUBCODER. Next, we opted for the similarities in three metrics to estimate similarity in the runtime behaviors of τ_s and τ_{gt}. A higher similarity in the runtime behaviors indicates a higher fidelity of the synthesized stub code.

- **Executed Instructions.** This metric measures the behavior of the test case with respect to exercising the code under test. In this paper, we identify the set of JAVA bytecode instructions in the production code that are executed by τ_s and τ_{gt}, denoted as $I(\tau_s)$ and $I(\tau_{gt})$, respectively. Test cases with similar runtime behaviors should execute similar sets of instructions. Therefore, we also report the Jaccard similarity coefficient [42] between $I(\tau_s)$ and $I(\tau_{gt})$. However, similar sets of executed instructions are not our only metric, since it is not a sufficient condition for similar behaviors. It is possible that two test cases behaving differently share similar sets of executed instructions.
- **Execution Path.** In addition to the set of executed instructions, we also traced the execution paths, which are the ordered sequence of instructions that are executed by the test cases. Comparing the execution paths of τ_{gt} and τ_s would give us more information about fidelity because, unlike executed instructions, the execution path captures the instruction execution order. For each of the test cases where STubCoder successfully synthesizes stub code, we collected the execution paths generated by τ_s and τ_{gt}, denoted as $P(\tau_s)$ and $P(\tau_{gt})$, respectively. Since Jaccard similarity coefficient cannot be applied to execution paths, we report their similarity based on edit distances as follows.

$$\text{Similarity}\left(P(\tau_s), P(\tau_{gt})\right) = 1 - \frac{DLev\left(P(\tau_s), P(\tau_{gt})\right)}{|P(\tau_s)| + |P(\tau_{gt})|}$$

where $DLev$ is the Damerau–Levenshtein distance [14]. For test cases spawning multiple threads, we match the threads that share similar traces, and $DLev$ denotes the sum of Damerau-Levenshtein distances between those thread pairs. A small edit distance indicates that τ_s and τ_{gt} traverse similar execution paths.
- **Killed Mutants.** Mutation analysis [31] measures the adequacy of a test case with respect to detecting faults. It injects artificial faults in the program and checks if the test cases can "kill" them (i.e., the test fails). In this paper, we mutated the CUT using PIT [10] by seeding faults and ran τ_s and τ_{gt} against the mutants. We identified the mutants that are killed by τ_s and τ_{gt}, denoted as $K(\tau_s)$ and $K(\tau_{gt})$, respectively. Test cases with similar behaviors should be able to kill similar sets of mutants. Therefore, we also report the Jaccard similarity coefficient [42] between $K(\tau_s)$ and $K(\tau_{gt})$.

We choose these metrics because they estimate the intent or behaviors of test cases. For example, the executed instructions and execution path are relaxed and tighten versions of path conditions. They are validated to be a good abstraction of test intents in a recent study on test repair [35]. Mutation coverage is a proxy for reflecting the behaviors of the test cases in terms of detecting potential bugs, and it was used to measure the behavioral similarity of test cases in a recent study that automatically refactors test cases with mocking [54].

Results. For generation mode, Table 4 shows the comparisons on executed instructions, execution path, and killed mutants by τ_s and τ_{gt} for each subject in our benchmark that STubCoder successfully synthesizes stub code in at least one run.[2] In our experiment, τ_s covers the similar set of the instructions as τ_{gt}, with the median of the Jaccard similarity coefficient to be 100%. In 24 of the 46 subjects, τ_s covers exactly the same set of instructions as τ_{gt}. In such cases, τ_s is capable for exercising the same instructions as τ_{gt}. The execution paths traversed by τ_s and τ_{gt} are also similar. The median of similarity is 99.99%, which indicates that τ_s exercise the CUT in a way similar to what τ_{gt} does. In 22 of the 46 subjects, the edit distance between the execution paths generated by τ_s and τ_{gt} is 0, which means that they execute the instructions in the production code in exactly the same order. In such subjects, τ_s and τ_{gt} exercise the CUT with the same intent. This is because

---

[2]PIT crashed due to its bug on subject #54 and therefore we cannot report the result for #54 in this RQ4.

Table 4. RQ4: Fidelity of the Synthesized Stub Code (Generation Mode)

τ_{gt} denotes the result generated by the ground truth. τ_s denotes the result generated by the test with synthesized stub code.
$\tau_{gt} \cap \tau_s$ denotes the intersection of the ground truth and the test with synthesized stub code.
Jaccard denotes Jaccard similarity coefficient.
$DLev$ denotes Damerau-Levenshtein distance.

Subject ID		Executed Instructions				Execution Path				Killed Mutants				
		τ_{gt}	τ_s	$\tau_{gt} \cap \tau_s$	Jaccard	τ_{gt}	τ_s	$DLev$	Similarity	Injected	τ_{gt}	τ_s	$\tau_{gt} \cap \tau_s$	Jaccard
N4J	#1	40	40	40	100.00%	40	40	0	100.00%	27	3	3	3	100.00%
	#2	431	431	431	100.00%	564	561	3	99.73%	32	4	3	3	75.00%
	#3	-	-	-	-	-	-	-	-	-	-	-	-	-
	#4	258	258	258	100.00%	258	258	0	100.00%	378	12	11	11	91.67%
SPB	#5	117	117	117	100.00%	171	171	0	100.00%	2	1	1	1	100.00%
GRC	#6	255	255	255	100.00%	257	257	0	100.00%	89	2	2	2	100.00%
	#7	-	-	-	-	-	-	-	-	-	-	-	-	-
	#8	-	-	-	-	-	-	-	-	-	-	-	-	-
MB3	#9	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#10	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#11	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#12	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#13	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#14	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#15	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#16	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#17	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#18	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#19	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#20	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
ZKN	#21	55	41	41	74.55%	57	41	16	83.67%	29	4	1	1	25.00%
	#22	-	-	-	-	-	-	-	-	-	-	-	-	-
	#23	46	4	4	8.7%	46	4	42	16.00%	29	2	0	0	0.00%
	#24	-	-	-	-	-	-	-	-	-	-	-	-	-
	#25	46	4	4	8.7%	62	4	58	12.12%	29	3	0	0	0.00%
APL	#26	40	40	40	100.00%	40	40	0	100.00%	5	0	3	0	0.00%
	#27	28	28	28	100.00%	28	28	0	100.00%	5	2	2	2	100.00%
	#28	327	317	317	96.94%	336	324	12	98.18%	40	8	7	7	87.50%
	#29	118	99	99	83.90%	120	99	21	90.41%	22	8	8	6	60.00%
	#30	265	265	265	100.00%	396	396	0	100.00%	40	6	6	6	100.00%
	#31	-	-	-	-	-	-	-	-	-	-	-	-	-
	#32	113	35	35	30.97%	113	35	78	47.30%	22	10	3	2	18.18%
	#33	-	-	-	-	-	-	-	-	-	-	-	-	-
ZUL	#34	31	31	31	100.00%	31	31	0	100.00%	4	2	2	2	100.00%
SBA	#35	-	-	-	-	-	-	-	-	-	-	-	-	-
	#36	-	-	-	-	-	-	-	-	-	-	-	-	-
JIB	#37	-	-	-	-	-	-	-	-	-	-	-	-	-
	#38	-	-	-	-	-	-	-	-	-	-	-	-	-
AZK	#39	-	-	-	-	-	-	-	-	-	-	-	-	-
AHP	#40	2197	2027	2026	92.17%	5083	4841	243	97.55%	288	8	5	5	62.50%
	#41	903	903	903	100.00%	9995	9998	5	99.97%	142	15	15	15	100.00%
	#42	1500	1436	1432	95.21%	11144	11016	136	99.39%	147	17	18	17	94.44%
	#43	1055	1055	1055	100.00%	9343	9346	5	99.97%	120	11	17	11	64.71%
	#44	1056	1093	1054	96.26%	9217	5400	3964	72.88%	120	14	15	14	93.33%
ADR	#45	59	59	59	100.00%	67	67	0	100.00%	17	3	3	3	100.00%
	#46	169	164	150	81.97%	320	208	126	76.14%	29	4	4	2	33.33%
ADU	#47	188	174	173	91.53%	235	221	15	96.71%	5	1	1	1	100.00%
	#48	61	62	60	95.24%	66	67	2	98.50%	30	5	5	5	100.00%
	#49	74	60	60	81.08%	78	62	16	88.57%	22	2	2	2	100.00%
	#50	70	56	56	80.00%	74	58	16	87.88%	22	1	2	1	50.00%
	#51	41	12	12	29.27%	41	12	29	45.28%	5	1	2	1	50.00%
	#52	64	64	64	100.00%	66	66	0	100.00%	2	1	1	1	100.00%
	#53	356	100	88	23.91%	448	106	355	35.92%	29	2	1	0	0.00%
	#54	-	-	-	-	-	-	-	-	-	-	-	-	-
	#55	64	64	64	100.00%	66	66	0	100.00%	2	1	1	1	100.00%
	#56	387	100	88	22.06%	465	106	372	34.85%	29	2	2	0	0.00%
	#57	296	178	165	53.40%	372	194	247	56.36%	22	1	2	0	0.00%
	#58	155	143	143	92.26%	185	173	12	96.65%	29	8	8	8	100.00%
	#59	92	6	6	6.52%	95	6	90	10.89%	5	2	0	0	0.00%
Median					100.00%				99.99%					100.00%

Table 5. RQ4: Fidelity of the Synthesized Stub Code (Repair Mode)

τ_{gt} denotes the result generated by the ground truth. τ_s denotes the result generated by the test with synthesized stub code.
$\tau_{gt} \cap \tau_s$ denotes the intersection of the ground truth and the test with synthesized stub code.
Jaccard denotes Jaccard similarity coefficient.
$DLev$ denotes Damerau-Levenshtein distance.

Subject ID		Executed Instructions				Execution Path				Killed Mutants				
		τ_{gt}	τ_s	$\tau_{gt} \cap \tau_s$	Jaccard	τ_{gt}	τ_s	$DLev$	Similarity	Injected	τ_{gt}	τ_s	$\tau_{gt} \cap \tau_s$	Jaccard
N4J	#1	40	40	40	100.00%	40	40	0	100.00%	27	3	3	3	100.00%
	#2	431	431	431	100.00%	564	559	5	99.55%	32	4	3	3	75.00%
	#3	305	277	277	90.82%	327	293	34	94.52%	67	15	14	13	81.25%
	#4	258	385	258	67.01%	258	389	131	79.75%	378	11	12	11	91.67%
SPB	#5	117	117	117	100.00%	171	171	0	100.00%	2	1	1	1	100.00%
GRC	#6	255	255	255	100.00%	257	257	0	100.00%	89	2	2	2	100.00%
	#7	-	-	-	-	-	-	-	-	-	-	-	-	-
	#8	-	-	-	-	-	-	-	-	-	-	-	-	-
MB3	#9	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#10	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#11	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#12	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#13	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#14	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#15	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#16	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#17	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#18	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#19	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
	#20	39	39	39	100.00%	39	39	0	100.00%	5	1	1	1	100.00%
ZKN	#21	55	41	41	74.55%	57	41	16	83.67%	29	4	1	1	25.00%
	#22	-	-	-	-	-	-	-	-	-	-	-	-	-
	#23	46	4	4	8.70%	46	4	42	16.00%	29	2	0	0	0.00%
	#24	-	-	-	-	-	-	-	-	-	-	-	-	-
	#25	46	4	4	8.70%	62	4	58	12.12%	29	3	0	0	0.00%
APL	#26	40	40	40	100.00%	40	40	0	100.00%	5	0	3	0	0.00%
	#27	28	111	28	25.23%	28	111	83	40.29%	5	2	2	2	100.00%
	#28	327	317	317	96.94%	336	324	12	98.18%	40	8	7	7	87.50%
	#29	118	99	99	83.90%	120	99	21	90.41%	22	9	9	7	63.64%
	#30	265	265	265	100.00%	396	396	0	100.00%	40	6	6	6	100.00%
	#31	-	-	-	-	-	-	-	-	-	-	-	-	-
	#32	113	35	35	30.97%	113	35	78	47.30%	22	6	3	1	12.50%
	#33	-	-	-	-	-	-	-	-	-	-	-	-	-
ZUL	#34	31	31	31	100.00%	31	31	0	100.00%	4	2	2	2	100.00%
SBA	#35	268	268	268	100.00%	279	279	0	100.00%	49	15	14	14	93.33%
	#36	297	297	297	100.00%	374	374	0	100.00%	4	2	2	2	100.00%
JIB	#37	-	-	-	-	-	-	-	-	-	-	-	-	-
	#38	1245	1228	1228	98.63%	1588	1465	123	95.97%	46	12	12	12	100.00%
AZK	#39	-	-	-	-	-	-	-	-	-	-	-	-	-
AHP	#40	2197	2027	2026	92.17%	5083	4841	243	97.55%	288	8	5	5	62.50%
	#41	903	901	901	99.78%	9995	5460	4535	70.66%	142	15	15	15	100.00%
	#42	1500	1436	1432	95.21%	11144	11016	136	99.39%	147	17	18	17	94.44%
	#43	1055	1094	1055	96.44%	9343	9526	183	99.03%	120	11	12	11	91.67%
	#44	1056	1093	1054	96.26%	9217	5400	3964	72.88%	120	14	15	14	93.33%
ADR	#45	-	-	-	-	-	-	-	-	-	-	-	-	-
	#46	169	164	150	81.97%	320	208	126	76.14%	29	4	4	2	33.33%
ADU	#47	188	174	173	91.53%	235	221	15	96.71%	5	1	1	1	100.00%
	#48	61	61	61	100.00%	66	66	0	100.00%	30	5	5	5	100.00%
	#49	74	60	60	81.08%	78	62	16	88.57%	22	2	2	2	100.00%
	#50	70	63	56	72.73%	74	65	16	88.49%	22	1	2	1	50.00%
	#51	41	47	12	15.79%	41	47	44	50.00%	5	1	2	1	50.00%
	#52	64	64	64	100.00%	66	66	0	100.00%	2	1	1	1	100.00%
	#53	356	115	94	24.93%	448	121	350	38.49%	29	2	1	0	0.00%
	#54	-	-	-	-	-	-	-	-	-	-	-	-	-
	#55	64	64	64	100.00%	66	66	0	100.00%	2	1	1	1	100.00%
	#56	387	117	101	25.06%	465	142	372	38.71%	29	2	1	0	0.00%
	#57	296	94	81	26.21%	372	104	281	40.97%	22	1	1	0	0.00%
	#58	155	143	143	92.26%	185	173	12	96.65%	29	8	8	8	100.00%
	#59	92	82	82	89.13%	95	85	10	94.44%	5	2	2	2	100.00%
Median					99.78%				99.39%					100.00%

the same execution path indicates that the tests share the same path conditions, which was shown to be a good abstraction of test intent [35]. The set of mutants killed by τ_s and τ_{gt} are also similar, with the median of similarity to be 100%. In 26 of the 46 subjects, τ_s kills exactly the same set of mutants as τ_{gt}, which means τ_s has the similar ability to detect injected bugs as τ_{gt}.

Table 5 gives the fidelity comparison for the repair more. The median of similarity in executed instructions is 99.78%, and τ_s covers the same set of instructions as τ_{gt} does in 24 of 49 subjects. The median of similarity in execution path is 99.39%, and τ_s shares exactly the same execution path as τ_{gt} in 23 of 49 subjects. The median of similarity in killed mutants is 100%, and τ_s kills the same set of mutants as τ_{gt} does in 28 of 49 subjects. In summary, STUBCODER synthesizes stub code similar to the ground truth with respect to executed instructions, execution path, and killed mutation. The high similarities indicate that the test cases with the synthesized stub code have adequacy similar to that of the ground truth. Such test cases are useful for detecting regression bugs when the CUT evolves.

Besides most of the high hifelities subjects, we also observed several cases that worth discussion:

- **Synthesis is successful but fidelity is low.** Such cases are caused by weak test oracles. The test oracles in these subjects (e.g., #23 and #25 in both modes) allow multiple execution paths to pass the test. Listing 4 shows an example to illustrate such cases. As shown in the code snippet, both τ_{gt} and τ can pass the test since both of them will result in a `CustomException` to be thrown. However, the execution paths of τ_{gt} and τ are different, and the mutants injected into the branch cannot be killed by τ. Such situations can be easily mitigated by enhancing the test oracle with a few mocking calls, specifying that certain methods should be called on the mock objects. After that, STUBCODER will synthesize stub code that helps cover the code lines invoking the methods specified by such mocking calls.

- **Killed mutants are the same but execution paths are different.** Such cases (e.g., #47, #49, and #58 in both modes) happened because τ_{gt} and τ takes different ways to construct certain objects. To illustrate such a difference, Listing 5 show a comparison of two ways to construct a string `"10"`. In the developer-written stub code, the string is constructed directly with a literal. In the synthesized stub code, the string is converted from an interger value. In this case, the instructions in `String.valueOf` will be included in the execution path of τ but not τ_{gt}. Nevertheless, the synthesized stub code is sill useful for developers as τ kills exactly the same set of mutants as τ_{gt}.

- **Execution paths are similar but killed mutants are different.** Such cases (e.g., #2, #35 in repair mode) are due to different return values specified in the stub code. Listing 6 illustrates such a case. The generated stub code and the developer-written stub code specify different return values for the method `getOffset`. Without mutation, both test cases can enter the then branch, and therefore they share the same set of executed instructions, and both of them make the test pass. However, when the mutation operator changes the + to -, the test case with developer-written stub code does not enter the then branch while the test case with generated stub code enters the then branch. In this case, the test fails with the developer-written code while it passes with the generated stub code.

- **Same executed instructions but different execution paths.** This is because there are loops in the production code and τ and τ_{gt} executed the loops for different number of times.

Overall, as shown in Table 4 and Table 5, the encoded information is sufficient for obtaining a useful stub code most of the time. This verifies our intuition that deriving the stub code from the information encoded in the CUT execution code and test oracle leads to adequate test cases.

```
1  // Synthesized stub code
2  when(someMock.getValue()).thenThrow(new CustomException());
3
4  // Developer-written stub code
5  when(someMock.getValue()).thenReturn(5);
6
7  // Class under test
8  int value = someMock.getValue();
9  if(value < 10) {
10     // ... mutants are injected here
11     throw CustomException();
12 }
13
14 // Assertions
15 assertThrows(CustomException.class, ()-> {
16     // CUT execution
17 });
```

Listing 4. Illustration of Weak Oracle

```
1  // Generated code
2  int var1 = 10;
3  String var2 = String.valueOf(var1);
4  when(someMock.foo()).thenReturn(var2);
5
6  // Developer-written code
7  when(someMock.foo()).thenReturn("10");
```

Listing 5. Illustration of Alternatve Object Construction

```
1  // Generated stub code
2  doReturn(2).when(someMock).getOffset();
3
4  // Developer-written stub code
5  doReturn(4).when(someMock).getOffset();
6
7  // In the class under test
8  int offset = someMock.getOffset() + 5;
9  if (3 <= offset && offset <= 10) {
10     // ...
11 } else {
12     // fail
13 }
```

Listing 6. Illustration of Different Killed Mutants due to Return Values

---

> ⚲ **RQ4 in Summary:** StubCoder synthesizes stub code with high fidelity, which means that they share a runtime behavior similar to that of the ground truth in terms of their effects on the code under test. The information encoded in CUT execution code and test oracle is useful for deriving stub code. StubCoder works well when the test oracle contains adequate information.

4.6 Threats to Validity

Subject Collection. We evaluated StubCoder on 59 test cases collected from 13 projects. Our results might not be generalized to other projects and test cases. The subject collection requires intensive manual effort, which limited the number of projects and test cases that we could use. To mitigate this threat, we selected large, actively maintained, diverse, and popular GitHub projects. These projects belong to different domains: big data, database, web apps, containers, etc. Our benchmark dataset reflects the real-world usage of stub code in these areas.

Also, when preparing the evaluation subjects, we rewrote the assertions written in other libraries into those using JUnit framework. Such manual modification might be affected by human mistakes and thus change behavior of the test cases. To mitigate this issue, we cross-checked the documentation of the corresponding assertion framework and JUnit to make sure the rewritten assertions preserves the original semantics. We also ran the test cases before and after modification to make sure that they yield the same result.

Fidelity Measurement. When measuring the fidelity of the stub code, we leveraged a metric based on instruction coverage. However, the similarity on the instruction coverage may not be ideal to reflect the differences. For example, when there are only 10% of the instructions in the class under test are in branches, the similarity of the instruction coverage will be at least 90%. To mitigate this threat, we introduced additional metrics such a execution path and killed mutants to further characterize the behavior of the test cases.

Experiments. Evolutionary algorithms are stochastic by nature, and the evaluation results may be different across several runs. In our experiments, we used 10 repetitions to evaluate StubCoder. The effectiveness of StubCoder is likely to increase with more attempts and a higher budget. However, StubCoder results are stable, as shown in Table 3. There are only a few test cases where the 10 attempts gave inconsistent results. Nevertheless, conducting more experiments is an important future work.

5   Related Work

Stub Code Generation. The first line of related work focuses on automatically generating stub code for mock objects. Capture-and-replay is a popular approach adopted by these techniques. In 2004, Saff et al. were among the first to develop a mock object construction technique [45, 46] aiming to improve the efficiency of unit testing. The technique runs a working test case to capture the interactions between the CUT and its dependencies. Next, those dependencies are replaced with mock objects, and stub code are generated using the captured information. A similar idea is used by Joshi et al. [32] and Elbaum et al. [18] for test craving. Fazzini et al. proposed MOKA [21] to collect and generate mock objects for testing mobile applications by observing the interactions between the application and its environment. More recently, Tiwari et al. designed Rick [53] to generate mock objects that mimic the behavior of the test dependencies in a production environment. Rick works by analyzing the runtime data captured in production systems and it successfully mimics 52.4% of the test executions as shown in evaluation.

These capture-and-replay techniques assume that dependencies are available when stub code is created. Conversely, StubCoder does not make this assumption as it generates stub calls without executing the actual dependency, which makes it applicable to a wider range of scenarios (e.g., for projects adopting TDD, the test dependencies may not be available when the test case is created). Moreover, capture-and-replay techniques may generate unreliable test cases when the captured behavior of the dependency is flaky or incorrect. Differently, StubCoder does not suffer from this issue.

Stub code is also generated by a few test generation techniques to increase test coverage. For instance, Arcuri et al. developed techniques for generating stub code for environment dependent classes [3, 4], which enable EvoSuite [23] to achieve higher coverage for the classes having such dependencies. Similar approaches are also adopted to construct stub code for databases [50], mobile apps [21], and web services [6, 58]. However, they can generate stub code for certain dependency types only (e.g., networking [4, 6, 58], database [50], file system [3, 36]) because they follow predefined rules, which are not applicable to an arbitrary mock object.

These techniques are closely coupled with domain knowledge. New rules have to be manually defined to generate stub code for the mock objects that are not considered by these techniques. As such, these techniques cannot be easily adapted to other types of mock objects. Also, they do not use mocking frameworks to specify the behavior of the dependencies, but light-weight implementations similar to "fake" mock objects. In comparison, StubCoder is domain agnostic and thus can synthesize stub code for an arbitrary mock object. Also, it allows developers to specify the behavior of mock objects with oracle assertions, which gives developers more control over the behavior of the synthesized stub code.

Empirical Studies on Mocking. The second line of related work focuses on the practices adopted by developers when using mock objects in their projects. Marri et al. [36] conducted an analysis on the usage of mock objects in testing file-system-dependent software and showed that mock objects can ease the process of unit testing. Mostafa et al. [41] analyzed the usage of mocking frameworks in 5,000 Java projects and revealed that mock objects are widely used although they are only used to substitute certain types of test dependencies. They also raise the need for an automated technique for synthesizing stub code. Spadini et al. [47] studied the usage of mock objects in three open-source projects and one commercial project. They highlighted the practice adopted by developers when making mocking decisions found that developers choose to substitute the classes that are hard to setup with mock objects. In addition, they reveal that stub code are frequently coupled with production code and need to be frequently updated, which make creating and maintaining stub code challenging. More recently, Zhu et al. [59] conducted an empirical study on four open-source projects and distilled 10 code-level rules that can affect the mocking decisions, based on which they proposed a machine learning based technique that recommends mocking decisions for developers. Wang et al. [54] proposed an auto refactoring tool to migrate inheritance based mock objects to mocking frameworks.

All of these studies provide evidence of the popularity and importance of mocking. They also discuss the challenges faced by developers when using mock objects. In this paper, we propose StubCoder to automatically generate and repair stub code for mock objects, helping developers to address some of the challenges.

Test Case Repair. The third line of related work aim to repair the broken test cases due to the evolution of production code. For example, Daniel et al. [16] proposed ReAssert, a test case repair technique implemented for JUnit. ReAssert suggests repairs to failing tests to make them pass again. The fixes suggested by it include replacing literals values and assertions. Daniel et al. [15] later enhanced the capability of ReAssert by proposing Symbolic Test Repair, which employs symbolic execution and constraint solving to update the expected values of the assertions. Compared with ReAssert, Symbolic Test Repair can repair the test cases with complex control flow or operations on the expected values. Similarly, Mirzaaghaei et al. [40] developed TestCareAssistant (TCA) to facilitate test evolution by repairing obsolete test cases and generating new test cases. TCA identify five common actions adopted by developers to adapt the test cases to new version, and apply these actions to the obsolete test case. While the above techniques can make the test pass again, they did not consider whether the intent of the test case are preserved. To fill this gap, Li et al. [35] proposed a technique for preserving the intent of the test case during test repair. They rank the repair candidates by the likelihood of preserving the intent of the original test case. The intent of a test case is characterized by analyzing the path conditions generated from a dynamic symbolic execution.

Test case repair techniques are also developed for GUI or web applications. Choudhary et al. [9] proposed WATeR to suggest repairs for automation script for testing web applications. The repairs are suggest by analyzing the the difference between a passing-failing pairs. WATeR can suggest

repairs for the test failure due to the type change of the web page elements and displaced or changed web page elements. Similarly, Stocco et al. [48] proposed VISTA to repair the test script of web applications by analyzing the visual information captured from test execution. They also equipped VISTA with a local crawling mechanism to handle non-trivial breakage scenarios. On the same theme, Gao et al. [26] developed SITAR, a semi-automated technique for repairing GUI test scripts. The repair is generated by reverse engineering the test script and map it to an event-flow graph. SITAR can amortizes the cost of human intervention across repairing multiple test scripts.

Although these techniques can effectively repair broken test cases, they focus on fixing the test exercise sequence and the assertions. They are not capable for repairing the obsolete stub code in broken test cases. In this paper, we proposed an application scenario for repairing the broken test cases by re-synthesizing the stub code to replace the obsolete ones.

6   Conclusions and Future Work

Mocking is an essential part of unit testing, as it allows testing a CUT in isolation from its dependencies [47, 59]. Mocking frameworks allow developers to write stub code to specify the behaviors of test dependencies when a test case invokes the CUT. However, developing and maintaining stub code is a labor-intensive and error-prone activity [47].

In this paper, we present STUBCODER to automatically generate and repair stub code for regression tests. STUBCODER is based on the intuition that the feedback given by the runtime behavior of a test case can drive the synthesis of stub code. In particular, STUBCODER implements an evolutionary algorithm guided by a fitness function that measures how close a candidate stub code is to pass the test.

Our evaluation on 59 test cases from 13 open-source projects shows that STUBCODER effectively synthesizes stub code. Moreover, STUBCODER outperforms its unguided variant, demonstrating the usefulness of the fitness function to steer the search towards generating test-passing stub code. Also, our results show that STUBCODER synthesizes stub code with similar behaviors as those written by developers.

To the best of our knowledge, STUBCODER is the first technique of its kind. There are several possible future work in this area. We point out the two most promising ones.

First, a possible future work to improve STUBCODER's effectiveness is to mine existing stub code in GitHub to learn recurrent patterns of stub calls. Indeed, different software projects often share the same libraries as test dependencies. Although stub code is test case specific, such recurrent patterns might help explore the search space more efficiently. For instance, the mutation operators of STUBCODER could give a higher probability to those mutations that match one of the mined recurrent patterns.

Second, some automated test generation techniques rely on mock objects to increase test coverage [1, 5]. However, such techniques do not explore the possible behaviors of mock objects during test generation. This is because they models each mock object and their stub calls as a single mutation unit. In this case, they cannot mutate each of the stub calls separately. In comparison, STUBCODER models the behavior of the stub code at a finer-grained level: it models each of the stub call as a mutation unit, and thus can explore more possible behaviors of mock objects.

The integration of such techniques and STUBCODER will enable finer control on the behavior of mock objects and thus achieve higher test coverage.

References

[1] Nadia Alshahwan, Yue Jia, Kiran Lakhotia, Gordon Fraser, David Shuler, and Paolo Tonella. 2010. AUTOMOCK: Automated Synthesis of a Mock Environment for Test Case Generation. In *Practical Software Testing: Tool Automation and Human Factors, 14.03. - 19.03.2010 (Dagstuhl Seminar Proceedings, Vol. 10111)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2010/2618/

[2] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press. https://doi.org/10.1017/CBO9780511809163

[3] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*. ACM, 79–90. https://doi.org/10.1145/2642937.2642986

[4] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2015. Generating TCP/UDP network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 155–165. https://doi.org/10.1145/2786805.2786828

[5] Andrea Arcuri, Gordon Fraser, and René Just. 2017. Private API Access and Functional Mocking in Automated Unit Test Generation. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*. IEEE Computer Society, 126–137. https://doi.org/10.1109/ICST.2017.19

[6] Thilini Bhagya, Jens Dietrich, and Hans W. Guesgen. 2019. Generating Mock Skeletons for Lightweight Web-Service Testing. In *26th Asia-Pacific Software Engineering Conference, APSEC 2019*. IEEE, 181–188. https://doi.org/10.1109/APSEC48747.2019.00033

[7] Tobias Blickle and Lothar Thiele. 1996. A Comparison of Selection Schemes used in Evolutionary Algorithms. *Evol. Comput.* 4, 4 (1996), 361–394. https://doi.org/10.1162/evco.1996.4.4.361

[8] Joshua Bloch. 2008. *Effective java*. Addison-Wesley Professional.

[9] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. 2011. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*. 24–29.

[10] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*. ACM, 449–452. https://doi.org/10.1145/2931037.2948707

[11] EasyMock contributors. 2022. EasyMock. https://easymock.org

[12] Mockito contributors. 2022. Mockito framework site. https://mockito.org

[13] Moq contributors. 2022. Moq4. https://github.com/moq/moq4

[14] Fred J. Damerau. 1964. A Technique for Computer Detection and Correction of Spelling Errors. *Commun. ACM* 7, 3 (mar 1964), 171–176. https://doi.org/10.1145/363958.363994

[15] Brett Daniel, Tihomir Gvero, and Darko Marinov. 2010. On test repair using symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*. 207–218.

[16] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. 2009. ReAssert: Suggesting repairs for broken unit tests. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 433–444.

[17] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* 6, 2 (2002), 182–197. https://doi.org/10.1109/4235.996017

[18] Sebastian G. Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006*. ACM, 253–264. https://doi.org/10.1145/1181775.1181806

[19] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*. ACM, 313–324. https://doi.org/10.1145/2642937.2642982

[20] Mattia Fazzini, Chase Choi, Juan Manuel Copia, Gabriel Lee, Yoshiki Kakehi, Alessandra Gorla, and Alessandro Orso. 2022. Use of Test Doubles in Android Testing: An In-Depth Investigation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2266–2278. https://doi.org/10.1145/3510003.3510175

[21] Mattia Fazzini, Alessandra Gorla, and Alessandro Orso. 2020. A Framework for Automated Test Mocking of Mobile Apps. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 1204–1208. https://doi.org/10.1145/3324884.3418927

[22] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[23] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)*. ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[24] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010*. ACM, 147–158. https://doi.org/10.1145/1831708.1831728

[25] Apache Software Fundations. 2022. Apache Commons. https://commons.apache.org

[26] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M Memon. 2015. SITAR: GUI test script repair. *Ieee transactions on software engineering* 42, 2 (2015), 170–186.

[27] GitHub. 2022. CloudFoundryApplicationFactoryTest of Spring Boot Admin. https://github.com/codecentric/spring-boot-admin/blob/d0085edfc757e1a83eb2ad4bf8f4764d2819eb6d/spring-boot-admin-client/src/test/java/de/codecentric/boot/admin/client/registration/CloudFoundryApplicationFactoryTest.java#L52

[28] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. 2018. *The Java® Language Specification* (java se 11 edition ed.). https://docs.oracle.com/javase/specs/jls/se11/html/index.html

[29] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[30] GitHub Inc. 2022. GitHub. https://github.com

[31] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[32] Shrinivas Joshi and Alessandro Orso. 2007. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007)*. IEEE Computer Society, 234–243. https://doi.org/10.1109/ICSM.2007.4362636

[33] Yuri Cossich Lavinas, Claus Aranha, Tetsuya Sakurai, and Marcelo Ladeira. 2018. Experimental Analysis of the Tournament Size on Genetic Algorithms. In *IEEE International Conference on Systems, Man, and Cybernetics, SMC 2018, Miyazaki, Japan, October 7-10, 2018*. IEEE, 3647–3653. https://doi.org/10.1109/SMC.2018.00617

[34] Shane Legg, Marcus Hutter, and Akshat Kumar. 2004. Tournament versus fitness uniform selection. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2004*. IEEE, 2144–2151. https://doi.org/10.1109/CEC.2004.1331162

[35] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2019. Intent-preserving test repair. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 217–227.

[36] Madhuri R. Marri, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. An Empirical Study of Testing File-System-Dependent Software with Mock Objects. In *Proceedings of the 4th International Workshop on Automation of Software Test, AST 2009*. IEEE Computer Society, 149–153. https://doi.org/10.1109/IWAST.2009.5069054

[37] Matias Martinez and Martin Monperrus. 2018. 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 (Lecture Notes in Computer Science, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, 65–86. https://doi.org/10.1007/978-3-319-99241-9_3

[38] James E McDonough. 2021. Test Doubles. In *Automated Unit Testing with ABAP*. Springer, 159–210.

[39] Brad L. Miller and David E. Goldberg. 1995. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Syst.* 9, 3 (1995). http://www.complex-systems.com/abstracts/v09_i03_a02.html

[40] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2012. Supporting test suite evolution through test case adaptation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 231–240.

[41] Shaikh Mostafa and Xiaoyin Wang. 2014. An Empirical Study on the Usage of Mocking Frameworks in Software Testing. In *2014 14th International Conference on Quality Software*. IEEE, 127–132. https://doi.org/10.1109/QSIC.2014.19

[42] Allan H. Murphy. 1996. The Finley Affair: A Signal Event in the History of Forecast Verification. *Weather and Forecasting* 11, 1 (1996), 3 – 20. https://doi.org/10.1175/1520-0434(1996)011<0003:TFAASE>2.0.CO;2

[43] Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM Comput. Surv.* 33, 1 (2001), 31–88. https://doi.org/10.1145/375360.375365

[44] Annibale Panichella, Rocco Oliveto, Massimiliano Di Penta, and Andrea De Lucia. 2015. Improving Multi-Objective Test Case Selection by Injecting Diversity in Genetic Algorithms. *IEEE Trans. Software Eng.* 41, 4 (2015), 358–383. https://doi.org/10.1109/TSE.2014.2364175

[45] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. 2005. Automatic test factoring for java. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*. ACM, 114–123. https://doi.org/10.1145/1101908.1101927

[46] David Saff and Michael D. Ernst. 2004. Mock object creation for test factoring. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04*. ACM, 49–51. https:

//doi.org/10.1145/996821.996838

[47] Davide Spadini, Mauricio Finavaro Aniche, Magiel Bruntink, and Alberto Bacchelli. 2019. Mock objects for testing java systems - Why and how developers use them, and how they evolve. *Empir. Softw. Eng.* 24, 3 (2019), 1461–1498. https://doi.org/10.1007/s10664-018-9663-0

[48] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 503–514.

[49] Hisashi Tamaki, Hajime Kita, and Shigenobu Kobayashi. 1996. Multi-Objective Optimization by Genetic Algorithms: A Review. In *Proceedings of 1996 IEEE International Conference on Evolutionary Computation*. IEEE, 517–522. https://doi.org/10.1109/ICEC.1996.542653

[50] Kunal Taneja, Yi Zhang, and Tao Xie. 2010. MODA: automated test generation for database applications via mock objects. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 289–292. https://doi.org/10.1145/1858996.1859053

[51] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary improvement of assertion oracles. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1178–1189. https://doi.org/10.1145/3368089.3409758

[52] Dave Thomas and Andy Hunt. 2002. Mock objects. *IEEE Software* 19, 3 (2002), 22–24.

[53] Deepika Tiwari, Martin Monperrus, and Benoit Baudry. 2022. Mimicking Production Behavior with Generated Mocks. *CoRR* abs/2208.01321 (2022). https://doi.org/10.48550/arXiv.2208.01321 arXiv:2208.01321

[54] Xiao Wang, Lu Xiao, Tingting Yu, Anne Woepse, and Sunny Wong. 2021. An automatic refactoring framework for replacing test-production inheritance by mocking mechanism. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 540–552. https://doi.org/10.1145/3468264.3468590

[55] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Inf. Softw. Technol.* 43, 14 (2001), 841–854. https://doi.org/10.1016/S0950-5849(01)00190-2

[56] Chak Shun Yu, Christoph Treude, and Mauricio Finavaro Aniche. 2019. Comprehending Test Code: An Empirical Study. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*. IEEE, 501–512. https://doi.org/10.1109/ICSME.2019.00084

[57] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Trans. Software Eng.* 46, 10 (2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648

[58] Linghao Zhang, Xiaoxing Ma, Jian Lu, Tao Xie, Nikolai Tillmann, and Peli de Halleux. 2012. Environmental Modeling for Automated Cloud Application Testing. *IEEE Softw.* 29, 2 (2012), 30–35. https://doi.org/10.1109/MS.2011.158

[59] Hengcheng Zhu, Lili Wei, Ming Wen, Yepang Liu, Shing-Chi Cheung, Qin Sheng, and Cui Zhou. 2020. MockSniffer: Characterizing and Recommending Mocking Decisions for Unit Tests. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*. IEEE, 436–447. https://doi.org/10.1145/3324884.3416539