Graph-Based Seed Object Synthesis for Search-Based Unit Testing

Yun Lin
dcsliny@nus.edu.sg
National University of Singapore
Singapore

You Sheng Ong
ong.aaron@u.nus.edu
National University of Singapore
Singapore

Jun Sun
junsun@smu.edu.sg
Singapore Management University
Singapore

Gordon Fraser
gordon.fraser@uni-passau.de
University of Passau
Germany

Jin Song Dong
dcsdjs@nus.edu.sg
National University of Singapore
Singapore

ABSTRACT

Search-based software testing (SBST) generates tests using search algorithms guided by measurements gauging how far a test case is away from exercising a coverage goal. The effectiveness of SBST largely depends on the continuity and monotonicity of the fitness landscape decided by these measurements and the search operators. Unfortunately, the fitness landscape is challenging when the function under test takes *object inputs*, as classical measurements hardly provide guidance for constructing legitimate object inputs. To overcome this problem, we propose *test seeds*, i.e., test code skeletons of legitimate objects which enable the use of classical measurements. Given a target branch in a function under test, we first statically analyze the function to build an object construction graph that captures the relation between the operands of the target method and the states of their relevant object inputs. Based on the graph, we synthesize test template code where each "slot" is a mutation point for the search algorithm. This approach can be seamlessly integrated with existing SBST algorithms, and we implemented *EvoObj* on top of the well-known *EvoSuite* unit test generation tool. Our experiments show that *EvoObj* outperforms *EvoSuite* with statistical significance on 2,750 methods taken from 103 open source Java projects using state-of-the-art SBST algorithms.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Search-based software engineering**.

KEYWORDS

object oriented, software testing, search-based, code synthesis

ACM Reference Format:
Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-Based Seed Object Synthesis for Search-Based Unit Testing. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468619

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468619

1 INTRODUCTION

Search-based software testing (SBST) generates test cases guided by measurements that gauge how far tests are from reaching a coverage objective. Using the measurement as fitness functions, the search algorithms can gradually improve the tests to achieve high coverage. SBST requires only light-weight instrumentation and has been demonstrated to be practically applicable in many scenarios in practice [13, 25, 53, 60, 73]. Despite the success of SBST and random testing tools like *EvoSuite* [25] and *Randoop* [50], recent empirical results suggest that object oriented code challenges the fitness functions traditionally used in search-based test generation, resulting in limited coverage performance [62, 63] and ability to detect real faults [61]. A primary cause appears to be the inability of search-based test generators to instantiate and configure valid objects [9, 11]. Unfortunately, object inputs are very common. Figure 1 summarizes the distribution of Java methods with object inputs in the SF100 benchmark [7], a dataset of 100 open source Java projects: 84.6% of all Java methods take at least one object input.

The effectiveness of SBST relies on the assumption that the search space is overall *continuous* and *monotonic* with respect to fitness measurements. For example, consider Listing 1, in which one would use the traditional branch distance [46] measurement to guide the search towards executing the true-branch (b) in line 3. This branch is based on two branch variables, i.e., operand op_1 as the returned value of the call of `getAwardNum()`, and operand op_2 as the constant 3. Given a test t and the condition of b as $b.c$, the branch distance of t from exercising b is defined as:

$$distance(b,t) = \begin{cases} op_2 - op_1 + K & \text{if } b.c \text{ evaluates to } \texttt{false} \\ 0 & \text{otherwise} \end{cases}$$

where K is a constant as minimum value of the branch distance when the branch is not covered. Let K be 0, assuming a test whose runtime valuation of op_1 is 0, and the valuation of op_2 (a constant) is always 3, the branch distance is $3 - 0 + 0 = 3$; a branch distance of 0 indicates that the branch is covered. Based on the guidance of the branch distance, SBST algorithms [13, 26, 60] treat the relation between the input variables and target branch condition as a black

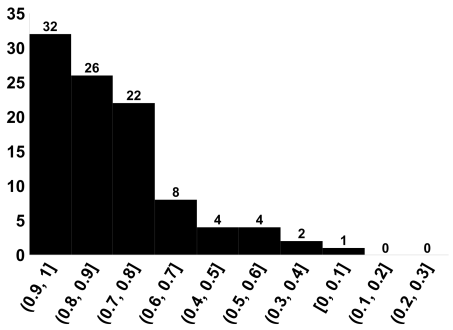

Figure 1: The distribution of the ratio of Java methods with object inputs from the projects in SF100 benchmark. Among the 100 Java projects, 32 projects have over 90% of methods with object inputs, 26 projects have 80-90% of methods with object inputs, etc. Overall, the mean ratio is 84.6%, and the median ratio is 83.3%.

```
1 public int example(Student s) {
2     if (s.getSupervisor().getCV().getAwardNum()
3         > 3)
4         return 1;
5 }
6 class Student{Supervisor s; ...}
7 class Supervisor{CV cv; ...}
8 class CV{int awardNum; ...}
```

Listing 1: A Branch Relevant to Object States

```
1 Student s = new
        Student();
2 example(s);
3
4
5
6
7
8
9
```

(a) Initial Test t_{ini}

```
1 Student obj0 = new Student();
2 Supervisor obj1 = new
        Supervisor();
3 CV obj2 = new CV();
4 int a0 = 0;
5 obj2.setAwardNum(a0);
6 obj1.setCV(obj2);
7 obj0.setSupervisor(obj1);
8 example(s);
9
```

(b) Effective Test t_{eff} for SBST

Figure 2: There is no continuity and monotonicity in the search space landscape to evolve from the initial test t_{ini} to the effective test t_{eff} for SBST with mutations regarding the branch distance measurement.

box, randomly mutate the evolving tests (e.g., via statement modification, insertion and deletion), and select more promising tests (i.e., with the minimum branch distance) in each evolving iteration.

Object inputs, unfortunately, usually lead to search spaces that are neither continuous nor monotonic. In particular, given a test with randomly constructed object inputs, it is hard for the branch distance measurement to guide the evolution. Consider the initialized test t_{ini} in Figure 2a: Since `s.getSupervisor()` will return a null value, a null pointer exception will be thrown at line 2 in Listing 1. Thus, the branch distance of t_{ini} for b is inaccessible as

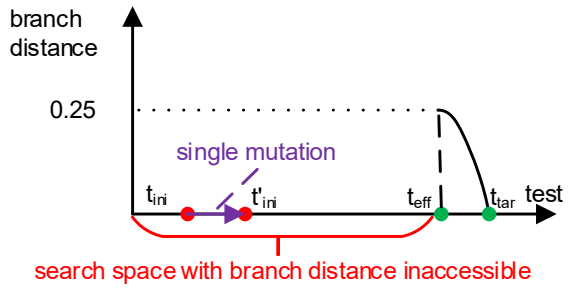

Figure 3: The idea underlying *EvoObj*: traditional mutation can hardly help t_{ini} escape from a search landscape with local optima. In this work, we aim to synthesize a seed point t_{eff}, which has a more continuous and monotonic landscape than point t_i towards the global optimum.

the branch operands are not even executed. Even worse, any tests t'_{ini} resulting from a mutation (e.g., inserting, deleting, and modifying statements in t_{ini}) on t_{ini} also leads to an inaccessible branch distance. Consequently, the search has no guidance.

Figure 3 illustrates this problem using the fitness landscape induced by the branch distance: Given t_{ini} (Figure 2a), the branch distance exists neither for t_{ini} nor for any of its "neighbour" tests resulting from a mutation (e.g., t'_{ini}). As a consequence, there is no gradient that guides the search to reaching the target test t_{tar}. However, consider test case t_{eff}, as shown in Figure 2b: For this test case the null pointer exception does not occur, and therefore the branch distance measurement now exists (i.e., 0.25)[1], and neighbour tests by mutating line 4 in Figure 2b can also have a different branch distance. In the fitness landscape (Figure 3) there is now a gradient that guides the search from t_{eff} to t_{tar}.

In this paper, we propose a systematic and practical approach to synthesize object inputs by analyzing static control and data flow to facilitate SBST. The approach creates *seed tests* such as t_{eff} in Figure 3 from which SBST can start the search with a more continuous and monotonic fitness landscape. This test seed synthesis approach works as follows: First, for an uncovered branch, we construct a test template which leaves some slots for an SBST algorithm to mutate their values. Each slot represents a descendant attribute (of primitive type) of an object input, representing the relevant object state to the branch. To this end, we construct an *object construction graph* by statically analyzing the interprocedural data and control flow from the object inputs of the target method to the operands used in the target branch. The object construction graph depicts what descendant attributes of the object input are relevant and how to construct object input accordingly (see Figure 4). The test templates are then generated through traversing the graph. Second, the test template code is used in the search through its statements assigning values for its slots. Here, each slot in the template represents a state variable of the object input. Finally, we integrate our approach into existing SBST algorithms at the minimum cost.

[1] According to Listing 1, the branch distance will be normalized by some function as $f(n) = \frac{n}{n+1}$. Thus, for the example code of Figure 2b, we have the branch distance as 3-0=3, which will be further normalized to $\frac{3}{3+1} = 0.25$.

```
1  class BasicRules{
2      // target method
3      boolean checkRules(Action action, GameState state) {
4          Player actor = state.player(action.getActor());
5          Player target = state.player(action.getTarget());
6
7          if (actor == null)
8              return false;
9          if (target == null)
10             return false;
11
12         // target branch
13         if (actor.getAction() == 1)
14             return true;
15     }
16 }
```

Listing 2: Target Method `checkRule()`

```
1  public void test(){
2      BasicRules basicRules0 = new BasicRules();
3      Action action0 = null;
4      GameState gameState0 = new GameState();
5      BasicRules basicRules1 = new BasicRules();
6      gameState0.notifyObservers();
7      int int0 = 1648;
8      String string0 = gameState0.toString();
9      gameState0.setGameState(int0);
10     byte[] byteArray0 = gameState0.pack();
11     basicRules0.checkRules(action0, gameState0);
12 }
```

Listing 3: Test Initialized by *EvoSuite*

```
1  class GameState extends Observable{
2      private Player[] players;
3      ...
4
5      public Player player(int id) {
6          if (id < 0 || id >= 128) return null;
7          return this.players[id];
8      }
9  }
10
11 class Action{
12     private int actor;
13     private int target;
14     ...
15
16     public Action (int actor, int target) {...}
17     public void setActor(int actor){...}
18     public int getActor(){return actor;}
19 }
20
21 class Player{
22     private int action;
23     ...
24 }
```

Listing 4: The Class of `GameState`, `Action`, and `Player`

We implemented this approach in the *EvoObj* tool, which extends *EvoSuite* for testing Java programs, and conducted experiments on 2,750 methods from 103 open source Java projects (SF100 dataset [7]). Although we implemented and evaluated this approach in the context of Java, it is independent of the language used and generalizes to any programming language where test generation requires the construction of complex objects via sequences of calls. Our experiment results show that *EvoObj* improves branch coverage compared to *EvoSuite* on various SBST algorithms [13, 26, 60] with moderate runtime overhead.

In detail, our work makes the following contributions:

- We propose a test seed synthesis approach designed for facilitating SBST on object oriented programs, mitigating the problem of non-monotonic or non-continuous landscapes in the search space (Section 3).
- We present an implementation of our approach, *EvoObj*, the source code and binaries of which are publicly available [2] (Section 4).
- We conduct an experiment consisting of 2,750 methods in 103 projects, showing *EvoObj* outperforms *EvoSuite* using various SBST algorithms (Section 5).

2 MOTIVATING EXAMPLE

In this section, we illustrate our approach using an example. Listing 2 shows a simplified excerpt of the *Gangup* project (the 27th project in SF100 benchmark [7]), which we will use as a running example in the remaining sections. In this example, the target method is the `checkRules()` method, and the target branch is the true branch of the if-statement in line 13. The target method takes two object inputs of type `Action` and `GameState`, and the operand in the target branch has both control and data dependencies on their attributes and returned value of the method calls (e.g., `player()`). Even with a time budget of 30 minutes *EvoSuite* does not succeed in generating a test that reaches the target branch.

Listing 3 shows an example test case generated by *EvoSuite*. Since `action0` is `null`, the call to `checkRules` immediately leads to a null-pointer exception. Even if `action` were not `null`, there are further challenges: The target branch is control dependent on the return values of the calls of `state.player()` (line 4 and 5 in Listing 2), which is influenced by the internal states of `action` and `state`. However, the branch distance measurement for the target branch provides no guidance for evolving these states.

Listing 4 shows the relevant dependency code: The return value of the method `player()` defined in the `GameState` class is (1) guarded by the condition on line 6 in Listing 4, and (2) depends on the `players` attribute in the `state` object (line 2 in Listing 4). Without constructing the `players` field with at least one element for the `GameState` object, the execution of test case will always terminate with a null pointer exception in line 7 of Listing 4, resulting in an unavailable branch distance on the target branch. However, random construction of `state` and `action` objects is very likely to result in `null` values returned from the `player()` method, making the branch distance on the target branch ineffective. As a result, the target branch is unlikely to be covered.

In this work, we overcome the challenges by synthesizing promising test templates for SBST to start the search. To this end, *for an uncovered branch*, our approach (1) builds its construction graph from the target method, depicting how to construct the object input with states relevant to the target branch, (2) generates test template code based on the graph, and (3) searches appropriate valuation for instantiating the template.

Object Construction Graph. An object construction graph links object inputs with their transitively relevant data fields. Figure 4

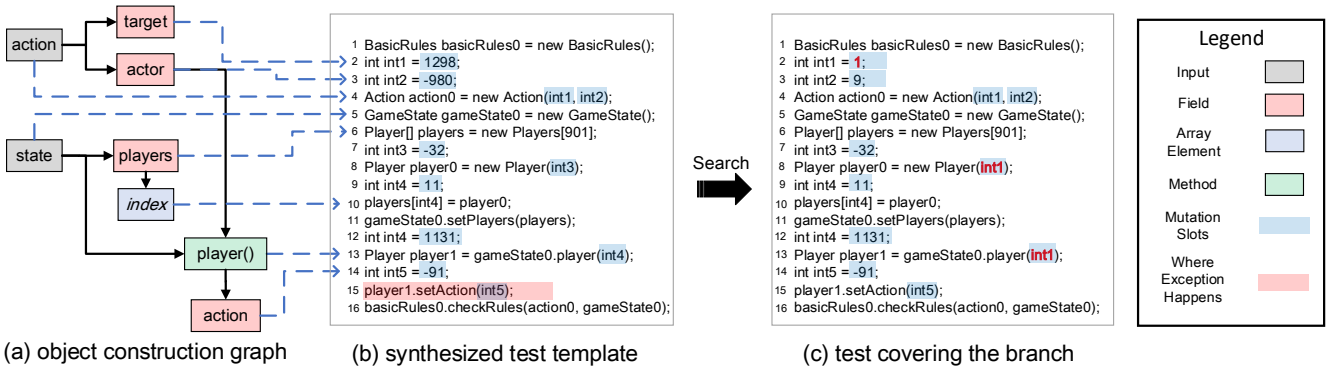

Figure 4: Example of graph-based test template generation. In the object construction graph, grey rectangles represent graph inputs; red rectangles represent fields/get-field operations; green rectangles represent call/call operations; and blue rectangles represent array element/get-array-element operations. Each node in the object construction graph can be used to generate a statement in the test code. Then, we apply mutations on the slots of the generated test template to evolve the test.

shows the object construction graph for Listing 2, which starts with two object inputs `action` and `state` (in grey). The rectangles of different colors represent different types of variables (e.g., fields, array elements, or method return values). Edges between nodes indicate dataflow relations relevant for guiding test code construction. For example, there is an ownership relation between the input object `action` and the field `target`; this information allows to construct a `target` attribute for the object input `action` when generating the test code. Each leaf node in the graph represents a variable which (1) is a descendant attribute from some object input, and (2) has data and control dependencies with the operands of the target branch.

Graph-based Template Synthesis and Search Valuation. Based on the object construction graph, we can construct a test seed template where each node in the graph corresponds to a statement in the test template. A test template consists of code preparing the object inputs with its required descendant attributes, followed by a call of the target method. The assigning variable values in the statements test template can be considered as "slots". Each slot is a value of primitive type, allowing the search algorithms to more effectively evolve tests based on the resulting fitness guidance. Thus, generated test templates serve as shortcuts towards global optima.

In our experiments, *EvoObj* can cover the target branch in Listing 2 in 12 seconds on average, while *EvoSuite* cannot even cover it within 30 minutes search budget.

3 APPROACH

In this section, we describe in detail (1) how object construction graphs are created, (2) how code is synthesized from these graph, and (3) how to integrate this approach into SBST algorithms.

3.1 Building Object Construction Graphs

3.1.1 Object Construction Graphs. Given a target branch b in the target method m_t, we define its object construction graph $G(b) = \langle I, N, E \rangle$, where I denotes the set of graph inputs (i.e., the inputs for m_t), N denotes the set of variables (e.g., fields, local variables, etc.), and E denotes the set of information flows for constructing one variable based on other variables, i.e., $E \leftarrow (I \cup N) \times (I \cup N)$. More specifically, an object construction graph describes (1) what descendant object attributes are relevant state-variables for exercising a branch and (2) how to construct a child node (i.e., variable) based on its parent nodes (variables).

Figure 4 (a) shows an object construction graph for the target method in Listing 2. The graph inputs (I) are represented as rectangles in grey. The variables (N) are represented as rectangles in red, green, and blue colors, representing fields (or, *get field* operations), array elements (or, *get array index* operations), and intermediate local variables (or, invoke *call method* operation such as `player()`). The flow from `player()` to `action` means that the return value of calling the `player()` method serves as an input for an `action` get-field operation. Moreover, the leaf nodes `target` (accessed via method `getTarget()` called on `action` in method `checkRules`), `actor` (accessed via method `getActor()`), `index` (accessed via method `player()` called on `GameState`) and `action` (accessed via method `getAction()` called on the `actor` object) are the relevant data/control-dependent attributes for the target branch. Explicitly setting these attributes in the tests improves the effectiveness of SBST, as it allows the search to evolve these values towards values that satisfy the target branch condition.

In addition, given an operand in a branch and its relevant object input, we need to know how to *access (or transform)* the operand from an object input to synthesize test code. To this end, we also regard each node in an object construction graph as an operation, serving as a micro-function which takes at least one input and generates one and only one output. For example, in Figure 4, the `players` node, to which the parent `state` node points, can be considered either as (1) a variable representing an attribute of `state` object or as (2) an operation taking the `state` object as an input and generating the `players` attribute as output. The "operation" view on the graph nodes allows us to generate code based on the graph. In this regard, each operation is similar to an instruction in an intermediate representation form (e.g., Java bytecode), and the flow between the operations represents how the output of an operation serves as the input of another operation. Thus, a node in the graph

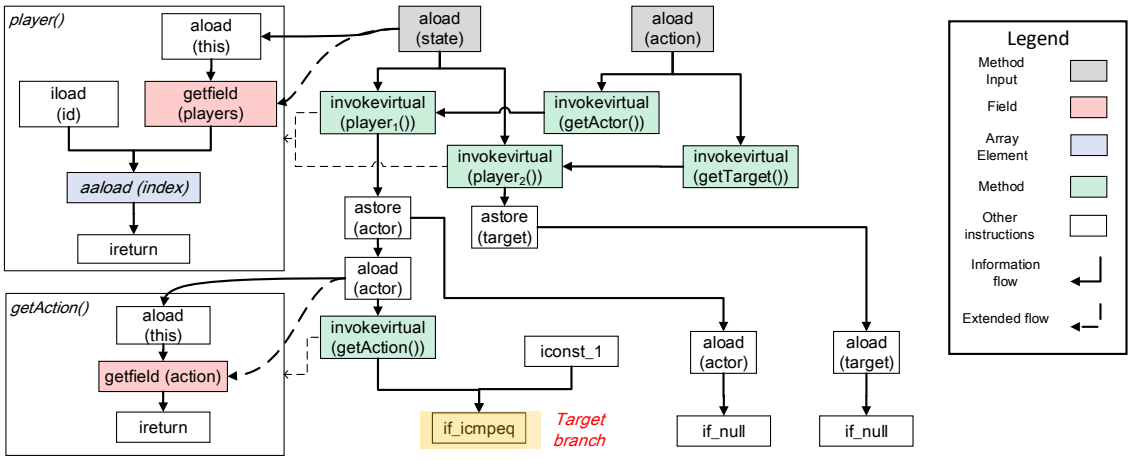

Figure 5: A partial graph of sliced program dependency graph on the target branch represented by the `if_icmpeq` instruction. Each node represents an instruction, and each edge represents the producing/consuming relation between the instructions.

can represent both an operation and its output variable. Each leaf node n_l indicates a variable reachable from a graph input and it is data/control-dependent on the operands of the target branch b.

3.1.2 Constructing Object Construction Graphs. Overall, constructing an object construction graph (OCG) takes three steps. First, given a target branch b in the target method m_t, we take b as a slicing criterion and slice a partial graph from the program dependency graph of m_t, which captures the interprocedural control- and data-dependencies. Second, on the partial graph, we identify relevant state-variables of object inputs of m_t and add information flow for objects across method calls. Finally, we remove irrelevant paths to the state-variables in the partial graph, resulting in an OCG of branch b in m_t.

Graph Slicing. Backwards slicing starts from the target branch b and ends at instructions satisfying one of these conditions:

- an instruction which reads a method input,
- an instruction which reads a global variable (e.g., static field in Java), and
- an instruction with no incoming flow.

Here, the method inputs of m_t include both its parameters and accessed instance fields. When slicing interprocedural information flow, we introduce a threshold t_{dep} to control the depth of the call graph hierarchy to limit the performance overhead. That is, once there is a call chain from m_t to m_k, for example $m_t \rightarrow m_1 \rightarrow ...m_k$, we let $k \leq t_{\text{dep}}$.

Figure 5 shows the sliced partial graph based on the target branch in Listing 2. For simplicity, we use Java bytecode (a stack-based instruction architecture) as intermediate representation in this example. Note that our approach is applicable in more general cases (e.g., register-based instruction architecture as LLVM) just as well. In Figure 5, each rectangle represents an instruction and each edge represents the dataflow between the instructions, specifically, how a (temporary) variable is produced by one instruction and consumed by another. For example, an `invokevirtual` instruction (e.g., `getAction()`) produces a temporary variable (as its returned value), which is consumed as an operand of the target branch represented as an `if_icmpeq` instruction. We refer to the JVM documentation [3] for more details on the semantics of Java bytecode.

Relevant State Variables and Interprocedural Analysis. We use the descendant fields of an object input as its state variables. In an object-oriented program, a branch can be covered only if an object is in a specific state (e.g., the array elements in a stack object, `players` field in a `GameState` object, and `actor` field in a `Action` object). We consider the instance fields and their descendant array elements as the state variable of an object. Therefore, when we track the instruction reading a field (e.g., a `getfield` instruction) or an array element (e.g., an `aaload` instruction), and keep them as a node in the graph, representing a relevant state-variable (i.e., field or array element) is required in certain object input.

In addition, in order to track all the relevant state-variables across method calls to the target branch, we extend the object-relevant flow across method calls to preserve complete information flows. When a method m_c is invoked by the method m_a, we analyze how the descendant fields of m_c's caller object and method parameters (defined in m_a) are used in m_c. Thus, we can track the relevant state variables used in m_c back to the caller method m_a.

For example in Figure 5, the `player`$_1$`()` method[2] is called from the object `state` of type `GameState`. In such a call, the `players` field in the `state` object input will be used to compute the returned value, which is in turn used to calculate the operand of the target branch. A lightweight alias analysis allows us to track the object produced by `aload (state)` (in grey in the target method) to the object produced by `aload (this)` in the `player()` method. Thus, we can extend the information flow from `aload (state)` to `getfield (players)` in the `player()` method. We use a dashed curve line in Figure 5 for the extended information flow. Similarly, we also extend the information flow from `aload (actor)` to `getfield (action)` in the `getAction()` method.

[2]Note that the method `player()` is called in line 4 and line 5 in the target method in Listing 2. Given they are called in different call sites, we use different subscripts to distinguish them.

Distilling Relevant Paths. Finally, we further reduce the partial graph by only keeping the paths starting with a method input and ending with an instruction reading a field (or array element). Note that, our interprocedural graph slicing can produce a path ending at a operand and starting at an instruction reading a constant (e.g., iconst_0). Given that we cannot modify such a constant when generating a test, such a path is irrelevant to generate a test template. Moreover, we also remove the aliased caller object and method parameters during the interprocedural analysis (e.g., the `aload (this)` instruction in the `player()` method) as their flow information has been duplicated with the extended interprocedural flow. Figure 4 (a) shows an example.

As a result, each path in the resultant OCG indicates (1) what state variables of an object input are relevant, and (2) how to construct an object with the state variables. For example, one path in the resultant OCG is: `aload(state)` → `invokevirtual(player`$_1$`())` → `astore (actor)` → `aload (actor)` → `getfield (action)`. The first instruction is the method input, the last instruction is the relevant state variable, and the whole path indicates the construction order which we can reverse-engineer into source code in the test.

3.2 Graph-Based Test Code Synthesis

In this section, we describe how we traverse object construction graphs in order to synthesize test template.

3.2.1 Code Skeleton Synthesis. We traverse the object construction graph in a breadth first manner for generating a test template. Overall, the traversal start from the nodes corresponding to an object input. Each time we visit a graph node, we can synthesize its statements in the test code.

Algorithm 1 shows the overall approach, which takes an object construction graph G as input and generates a test *test* as output. We first initialize the *test* by calling the target method with random inputs (line 1). Then, we select the nodes in top layer of G, i.e., the nodes with no parents, and push them into a queue (line 2-3). Afterwards, we use the queue to generate the statements in *test* in breadth first order (line 5-14). Note that we keep a map `map<node, statement>` to track the location of statements in test of each graph node (line 4 and line 10). Each time a node *node* is taken from the queue, we first check whether its code has already been generated. If it has, we find the corresponding code statement, *s*, of *node* in *test* and build the mapping relation between *node* and *s* (line 16-17). Otherwise, we check whether all its parent nodes in G have a corresponding statement in *test*. If not, we re-enqueue *node* so that we can generate code for its parent nodes first (line 14). Otherwise, we generate the statements for *node* (line 9-12). Note that, the code of *node* (e.g., a method call) cannot be generated if any of its parents are missing (e.g., the object to start the call or its required parameters). Once a new statement is generated in *test*, we further push its children into the queue (line 11). Readers can refer to the links (in blue) between Figure 4(a) and Figure 4(b) to go through the BFS algorithm.

3.2.2 Code Element Synthesis. Each time we visit a non-leaf node n in the object construction graph G, we generate a statement s regarding the instruction of n. Our transformation from a graph node to a code statement aims to meet the following needs: First, the generated test code should strictly conform to the dataflow indicated in the OCG. Second, when a state variable (i.e., a field or array element) in the OCG is private, we need to search for the most promising setter/getter function for the test code.

Algorithm 1: Test Code Template Synthesis

Input : An object construction graph, G
Output: A generated test case, *test*

```
1  test ← call the target method with random initialization;
2  nodes ← top layer nodes in G;
3  queue.push(nodes);
4  map⟨node, statement⟩ ← ∅;
5  while queue is not empty do
6      node ← queue.pop();
7      if node's code has not been generated then
8          if all the parent nodes of node have a corresponding
             statement in test then
9              s ← generate statement for node with map;
10             map.push(node, s);
11             t ← insert s into test;
12             queue.push(node.children);
13         else
14             queue.push(node);
15     else
16         s ← find the corresponding statement of node;
17         map.push(node, s);
18 return test;
```

Dataflow Preservation. In order to preserve the dataflow in the generated test, we maintain a node-statement map where the keys are OCG graph nodes and values are statements (and their defined variables) in the test. Overall, the generated test consists of the code to (1) prepare the method inputs and (2) call the target method at the end. Thus, we do not need to synthesize control-flow code in the test. Moreover, we track the *defined variable* of each statement. For example, the statement "`BasicRules basicRules0 = new BasicRules();`" has a defined variable of `basicRule0`. If a statement does not define a variable (e.g., `obj.setAction();`), its defined variable is *void*. When visiting a graph node n, we first check all its parent nodes $N_d = \{n_1, n_2, ..., n_m\}$ in OCG, and use the node-statement map to locate their corresponding statements $code(N_d) = \{code(n_1), code(n_2), ..., code(n_m)\}$ in the test. Each defined variable of $\texttt{code}_i$ ($i \in [1, m]$) plays a role to synthesize the statement of n. Given each node is essentially an instruction, based on the information flow between the instructions, we know what role (e.g., caller object, method parameter, operand of assignment, etc) each defined variable plays in synthesizing a new statement.

For the example in Figure 4, the graph node `player()` in green depends on the node `state` and the node `actor`, where the node `state` is mapped to "`GameState gameState0 = new GameState();`" (line 5) and the node `actor` is mapped to "`int int2 = -980;`" (line 3). Moreover, the flow between instructions allows us to know that `state` is the caller object and `actor` is the method parameter.

Thus, we generate the statement for the node `player()` as "`Player player1 = gameState0.player(actor);`" (line 13 in Figure 4 (b)).

Note that some graph nodes such as loading or storing a variable (e.g., `aaload` and `astore`) may not derive new statements. It is because such instructions generate no additional variable semantically, which allow us to reuse generated variable in the test template. In this case, we still map those nodes to an existing statement so that their defined variable can be used in synthesizing the follow-up statement in the test.

Setting Object States. We use fields to represent the state-variables of the method input objects[3]. Different from the graph nodes such as calling a method, synthesizing a statement from a *non-static* and *non-public* field-related node is more complex, and requires selecting appropriate method calls to get or set the field. Note that there can be multiple public methods to get or set a field in a direct or indirectly way. First of all, when we visit a graph node n representing a non-static and non-public field `f`, we synthesize a getter method for `f` if n is a non-leaf node and a setter method for `f` if n is a leaf node. Given the set of methods as $M = \{m_1, m_2, ..., m_k\}$, we sample a candidate method `m` from `M` based on an estimate of the likelihood of `m`'s invocation to get or set `f`.

A field-accessing node will derive a field-setting statement if it is a leaf node, and a field-reading statement otherwise. When visiting a node related to an instance field (e.g., with instruction as `getfield`), we must start with an object `obj`. First, we scan all the methods accessible by `obj` in its class (and superclass), and build their call graph. Given a path in the call graph $p = \langle m_s, ..., m_e \rangle$ where m_s is the starting node and m_e is the ending node, if its ending node m_e sets the required field, we keep it as an *interesting path*.

For each interesting path $p = \langle m_s, ..., m_e \rangle$, we then estimate its likelihood to set the field by (1) the number of method inputs in m_s, N_v, which are data dependent by the instruction in m_e to set the required field, and (2) the complexity of p (defined as follows). For each node m_i, we assigning a score $comp(m_i)$ by heuristically estimating how likely m_i can call m_{i+1} in the call chain p. For the ending node m_e, the score estimates how likely m_e can set the required field. Given a node m_i, we build its control flow graph. Let the set of branches to access the required method call or field-setting instruction be B' and the set of total branches B, $comp(m_i) = \frac{|B'|}{|B|}$. As a result, we use Equation 1 to assign the score for each path, where $|p|$ means the length of p:

$$score(p) = (N_v + 1) \times \frac{\sum_1^{|p|} comp(m_i)}{|p|} \quad (1)$$

Given a method m serving as the starting node for multiple interesting paths $\{p_1, ..., p_n\}$, we evaluate its score $score(m)$ as $max(score(p_1), ..., score(p_n))$. Finally, we sample the methods based on their score, and call the selected method to set the field.

Figure 6 shows an example where all four methods can set the field f directly or indirectly. We build five interesting paths and their score as follows:

- $p_1 = \langle m1, m2, m3 \rangle$, $score(p_1) = (1 + 1) \times \frac{(1/2+1+1)}{3} = 1.67$;
- $p_2 = \langle m2, m3 \rangle$, $score(p_2) = (1 + 1) \times \frac{(1+1)}{2} = 2$;

[3]Here, we regard array element as a special form of field of an array object.

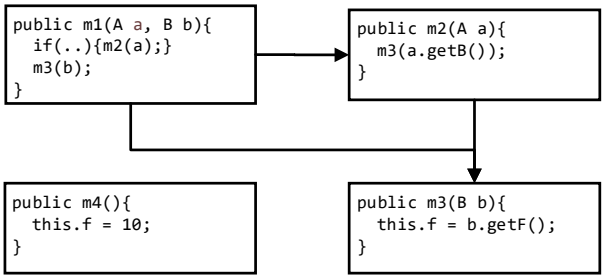

Figure 6: An Example of Setter Selection

- $p_3 = \langle m1, m3 \rangle$, $score(p_3) = (1 + 1) \times \frac{(1+1)}{2} = 2$;
- $p_4 = \langle m3 \rangle$, $score(p_4) = (1 + 1) \times \frac{1}{1} = 2$;
- $p_5 = \langle m4 \rangle$, $score(p_5) = (0 + 1) \times \frac{1}{1} = 1$;

As a result, $score(m1) = max(1.67, 2) = 2$, $score(m2) = 2$, $score(m3) = 2$, $score(m4) = 1$. Thus, we can normalize their score to the sampling probability as $p(m1) = p(m2) = p(m3) = 2/7 = 28.6\%$ while $p(m4) = 1/7 = 14.3\%$.

3.3 Integration to SBST

Note that, static data/control flow analysis and code generation can incur additional runtime overhead, which can influence the efficiency of test generation. In order minimize the incurred runtime overhead, we integrate the test code synthesis during the evolution stage using a user-defined probability. It means that if a branch relevant for an object input is trivial (i.e., easy to cover by random test generation in the initialization stage), we do not bother to generate test code templates. We only pay the runtime computational resource on "hard" branches.

During test evolution, we collect the set of uncovered program branches B after each iteration, and randomly select one branch b from B. Taking b as the target branch, we synthesize tests for b with a probability of p_{app}.

4 IMPLEMENTATION

We implemented the proposed approach in the tool *EvoObj* on top of *EvoSuite* [25], a state-of-the-art Java testing framework. In *EvoObj*, we integrate our approach into three search algorithms (i.e., DynaMOSA, MOSA, and MonotonicGA), and leave extensible interface to integrate with more search algorithms in the future. Moreover, we introduce new configurable options in *EvoObj* to support the introduced new features. Its source code and binaries are available on our website [2]. Note that, although our implementation of *EvoObj* depends on the *EvoSuite* framework, the challenge of synthesizing object inputs exists in any search-based testing/-fuzzing solutions such as Randoop [50] and AFL [1]. The idea of *EvoObj* to parse the structure of the object inputs and compute their relevance to uncovered branches, is orthogonal to (and thus can complement) various search algorithms [51, 52, 58], seeding techniques [57], testability transformation [32], and test-generation heuristics under many testing frameworks.

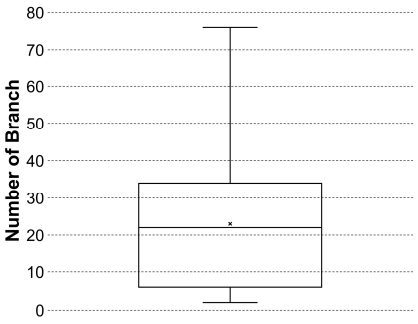

Figure 7: Distribution of the number of branches

5 EXPERIMENT

We evaluate our approach with the following research questions:

- **RQ1:** Can *EvoObj* outperform the state-of-the-art?
- **RQ2:** How does *EvoObj* perform with different time budgets?
- **RQ3:** How does *EvoObj* perform with different SBST algorithms?

5.1 Experiment Setup

Baseline: We choose *EvoSuite* [25] as a baseline for comparison. *EvoSuite* supports various object-oriented mutations such as randomly initializing Java objects and invoking method calls on these objects. More importantly, it integrates many state-of-the-art approaches [13, 26, 60] for testing object oriented programs. For example, it supports useful heuristics such as generating seeds with static and dynamic constants and constructing objects of diversified types to cover challenging branches [25, 26].

Experiment Subjects: Our subject dataset consists of two sources, the SF100 dataset and three popular complementary open source Java projects. The SF100 dataset [7] is a standard benchmark for evaluating unit testing, which consists of 100 open-source Java projects. Since some of the projects contained in SF100 are no longer actively maintained, we further included the Weka [8], JFeeChart [5], and JEdit [4] projects in our experiments. These three projects are still well-maintained, often used in the annual test generation contest [59], thus enhancing the diversity and representativeness of our dataset. As targets for our experiments, we randomly selected 2,000 Java methods from the SF100 dataset and 750 Java methods from the three complementary Java projects. Figure 7 shows the distribution of the number of branches in the total 2750 methods (omitting outliers for clarity). Overall, the mean number of branches of the sampled methods is 23.0, the median number is 22, the maximum is 240, and the minimum is 2. More details of our sampled methods can be found online [2].

Configuration: Both *EvoSuite* and *EvoObj* are configured to run the same three search algorithms, i.e., DynaMOSA [52], MOSA [51], and a Monotonic Genetic Algorithm [58]. This selection of algorithms covers many/single objective optimization, test cases/test suites as individuals for evolution, and different heuristics. Note that, single-objective approaches calculate a single cumulative fitness value by aggregating the fitness values for all individual coverage objectives of a method/class, while many-objective approaches consider each branch to cover as an independent objective. Since both approaches try to maximize the coverage, their performance can be directly compared using coverage. DynaMOSA has been empirically demonstrated to be the best performing evolutionary algorithm in the context of test generation [19, 52].

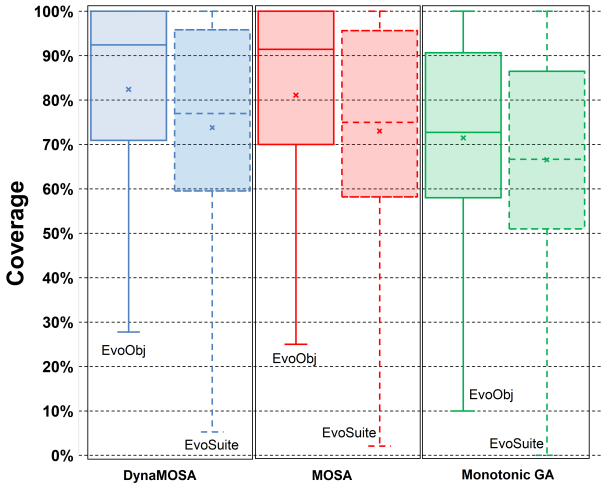

Figure 8: Coverage distribution of different configurations: the blue color represents DynaMOSA, the red color represents MOSA, and the green color represents MonotonicGA; the solid line represents *EvoObj* and the dashed line represents *EvoSuite*.

Performance Evaluation: We set the overall time budget to 200 seconds for each method. While generating tests for each method and configuration, we record intermediate coverage values every 10 seconds, which allows us to (1) compare their coverage performance under various budgets (such as 60s, 70s, ..., 190s, 200s) and (2) observe how the coverage of each approach grows with the increase of the time budget. Given that the common range for unit test generation tools is about 2 minutes, we choose the budget of 200 seconds to evaluate the impact of the additional computational effort. Both tools stop when either the budget is used up or 100% branch coverage is achieved. For each method, we run each tool under a configuration for 10 times, and then compare the average coverage and time respectively.

We set the bound on call depth t_{dep} to be 5, the maximum length of generated test case to be 200, and the probability to apply object construction technique to be 0.3. We run our experiment on 20 nodes on the NCL cloud [6], consisting of nodes with Intel Xeon E5-2620 CPU of 2.1GHz and 64G DDR4 Memory. Besides, we let *EvoObj* and *EvoSuite* share the default runtime configurations. The details can be found on our tool website [2].

5.2 RQ1: Performance of *EvoObj*

Figure 8 and Figure 9 show how *EvoObj* and *EvoSuite* are compared with respects to the three SBST algorithms in terms of coverage and time distribution within the runtime budget of 200 seconds. Moreover, Table 1 shows a detailed coverage comparison of *EvoObj* and

Table 1: Coverage Performance Comparison on Different SBST Algorithms regarding the budget of 100s, 150s, and 200s.

Coverage Performance	Budget-100s (%)			Budget-150s (%)			Budget-200s (%)		
	DynaMOSA	MOSA	MonotonicGA	DynaMOSA	MOSA	MonotonicGA	DynaMOSA	MOSA	MonotonicGA
EvoObj	78.82	77.35	67.22	81.70	80.43	70.44	82.40	81.11	71.48
EvoSuite	70.35	69.36	62.15	72.98	72.21	65.41	73.80	73.03	66.56
p-value	<10E-10	<10E-10	<10E-10	<10E-10	<10E-10	<10E-10	<10E-10	<10E-10	<10E-10
Effect size	0.33	0.36	0.36	0.34	0.37	0.36	0.21	0.22	0.22

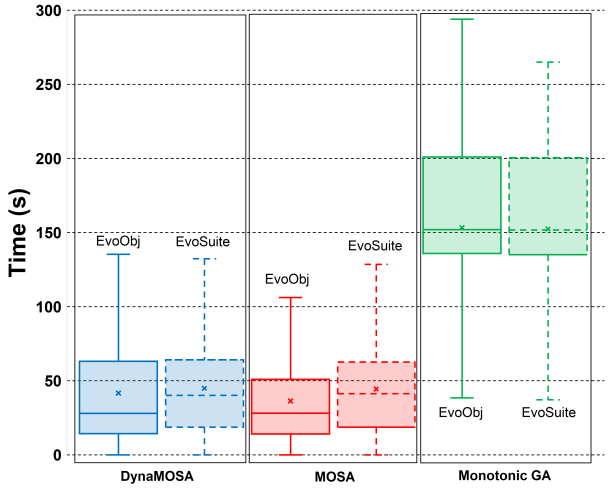

Figure 9: Time distribution of different configurations, the representation is the same as Figure 8.

Table 2: Runtime Overhead on Different SBST Algorithms

Approach	DynaMOSA (s)	MOSA (s)	MonotonicGA (s)
EvoObj	41.68	36.36	153.33
EvoSuite	44.92	44.39	152.43
p-value	<10E-10	<10E-10	0.13
Effect size	0.09	0.26	-0.02

EvoSuite for three specific points in time: After 100s we expect to see negative effects of the overhead produced by *EvoObj*; after 200s we expect that both tools had sufficient time for the search to converge; and 150s is an intermediate point between these two values. In addition, Table 2 shows the runtime overhead of *EvoObj* and *EvoSuite* given the total 200-second budget. We apply the Mann-Whitney U test [24] on coverage and calculate the two-tailed significance value p value as well as the Cohen's d as effect size. We consider *EvoObj* to outperform *EvoSuite* in coverage if *EvoObj* achieves a higher average coverage than *EvoSuite* and the p value is smaller than 0.05; and *EvoObj* outperforms *EvoSuite* in runtime overhead if *EvoObj* achieves less average overhead than *EvoSuite* and the p value is smaller than 0.05.

Overall, we observe that *EvoObj* outperforms *EvoSuite* in terms of branch coverage with statistical significance for all three algorithms. This suggests that *EvoObj* explores the structure of object inputs

```
1  public boolean equals(Object obj){
2      if(this == obj) return true;
3      if(obj instanceof CustPayeeModRsSequence2){
4          CustPayeeModRsSequence2 temp = (CustPayeeModRsSequence2)obj;
5      if(_custPayeeId != null) {
6          if(temp._custPayeeId == null) return false;
7          if(!_custPayeeId.equals(temp._custPayeeId)) return false;
8          else if(temp._custPayeeId != null) return false;
9          if(_custPayeeInfo != null){...}
10         ...
11         if(_SPRefId != null){...}
12         else if(temp._SPRefId != null)
13         return false;
14         ...
15     }
16 }
```

Listing 5: Complicated containment relationship

uses the branch distance more effectively. This is more impressive considering the runtime overhead in *EvoObj* for code analysis and synthesis. Moreover, we also observe that the coverage/runtime improvement on the MonotonicGA is not so significant as DynaMOSA/MOSA, we will discuss more in Section 5.3. Next, we discuss the negative impact incurred by *EvoObj*.

The most important challenge for *EvoObj* is that large object construction graphs can incur large runtime overhead and may increase the risk for runtime exceptions when constructing the object inputs. We observe significant runtime overhead when the required object is complicated, i.e., there are many fields and many layers of objects. In such as a case, *EvoObj* constructs a huge object construction graph, which leads to a long test case. The longer the test, the more likely it results in unexpected runtime exceptions. More importantly, a huge graph means that synthesizing test templates takes more time at runtime, which reduces the number of iterations of the evolutionary algorithm within a limited time budget.

For example, Listing 5 shows the code of the `equals()` method in the `CustPayeeModRsSequence` class in project 84 (ifx-framework). The method compares two objects regarding their attributes layers by layers. Figure 10 shows a part of its object construction graph which consists of 128 nodes and the synthesized test consists of over 700 statements. Note that, a node in an object construction graph may correspond to multiple statements. For example, a method call requires preparing the code for all its parameters. As executing tests with such a length will incur unacceptable runtime overhead, our implementation sets a test case length limit of 200 statements. In contrast, *EvoSuite* has a default test length threshold (i.e., 40), which avoids long tests. We further discuss this in Section 5.3.

In order to cover a deeply nested branch, a test with sufficient statements to set all the relevant object state is necessary. *EvoObj* only conducts static taint analysis to track the branch operand back

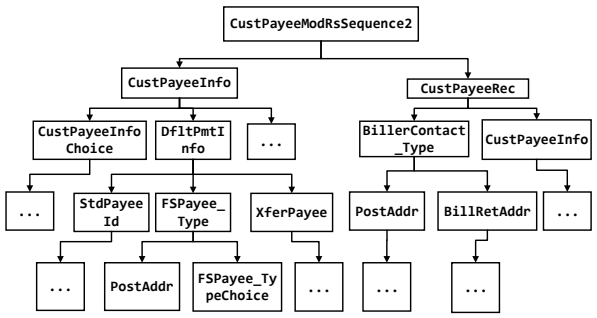

Figure 10: The object containment relation for the `equals()` method for the `CustPayeeModRsSequence2` class. The parent-child relationship represents the parent class has an attribute of the child class.

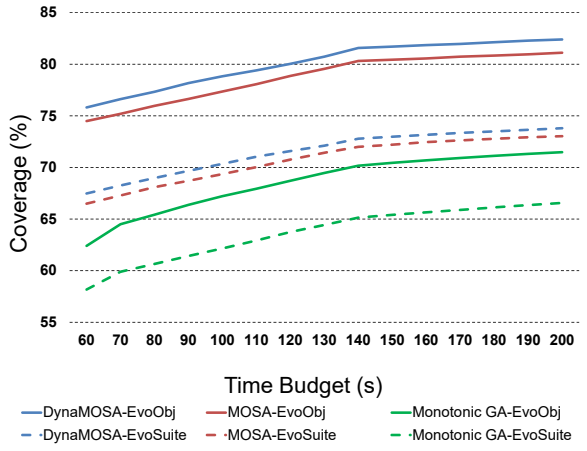

Figure 11: The coverage of *EvoObj* and *EvoSuite* on different SBST algorithms under the budget from 60s, 70s, ..., to 200s.

to some state variable of an input object, which favors completeness over soundness. A more optimal solution would be to conduct dynamic taint analysis [22] to discriminate some state variable over others, thus we can minimize the test case length while preserving the precision. We will explore this solution in our future work.

Finally, we observe that *EvoObj* can sometimes be ineffective as some instructions in the computation path cannot be reverse-engineered into source code statements. For example, usually a `getfield` instruction can derive a statement in the test to set or get a field. However, if the field relevant to a branch operand is private and there are no public getters and setters available in the target class, our generated test template can be incomplete, and thus less effective at facilitating the search process.

5.3 RQ2-3: Search Budgets and Algorithms

Figure 11 summarizes the coverage performance of *EvoObj* and *EvoSuite* on different SBST algorithms under the budget from 60s to 200s. In Figure 11, the SBST algorithms sharing the same color (e.g., DynaMOSA algorithm is presented by blue lines, MOSA algorithm is presented by brown lines, and Monotonic GA algorithm is presented by green lines.) Moreover, the *EvoObj* solutions use solid lines and the *EvoSuite* solutions use dashed lines.

Generally, Figure 11 shows that many-objective optimization solutions can benefit more from our approach. Our experiment confirms the existing literature [51, 52] in that DynaMOSA/MOSA has a better performance than the Monotonic GA (see the three dashed lines in Figure 11). In this experiment, the average coverage gaps of DynaMOSA and MOSA (across different time budget) are about 8.5% and 8.1%, while that of Monotonic GA is only 4.9%. The results also align with Figure 8 and Figure 9 in that the advantage of *EvoObj* on a Monotonic GA is smaller than that on DynaMOSA and MOSA. The Monotonic GA is a single-objective optimization solution to evolve a test suite on the overall branch coverage, while DynaMOSA and MOSA are many-objective optimization solutions to evolve test cases regarding each individual branch. Moreover, *EvoObj* synthesizes test templates for each branch. Thus, improving the branch distance of a single branch may not be well reflected in the overall branch coverage measurement of the Monotonic GA algorithm. Hence, *EvoObj* aligns with Monotonic GA being less effective compared to DynaMOSA and MOSA.

We also observe that the coverage gap between *EvoObj* and *EvoSuite* on each algorithm slightly increases during the initial stage of test generation. For example, after 60s, the coverage gap is 8.3%, 8.0%, and 4.1% for DynaMOSA, MOSA, and Monotonic GA respectively. In contrast, the gap increases to 8.5%, 8.0%, and 5.1% after 100s, and finalizes to 8.6%, 8.1%, and 4.9% after 200s. We observe that the coverage increase is larger up to around 150 seconds for *EvoObj* and *EvoSuite*, after which the coverage grows more slowly. This indicates that the total budget of 200 seconds is sufficient to evaluate method-wise unit testing.

5.4 Threats to Validity

Threats to external validity arise from the use of the SF100 dataset, as some projects can be obsolete and we observe that some project are not maintained. To mitigate the risk, we include three popular Java projects to enhance the diversity and representativeness of the dataset, resulting in a dataset of 103 open source Java projects.

Threats to internal validity may arise from the randomness pf the search algorithms. To mitigate this threat we evaluated *EvoObj* and *EvoSuite* with 10 repetitions on each method. In the future, we will run experiments on more Java projects and larger number of iterations for more generalizable results.

6 RELATED WORK

6.1 Search-Based Software Testing

Search based Software Testing (SBST) is an established approach for automating software testing [30, 33, 46]. The first SBST technique can be traced to Miller et al.'s work [47] for generating test cases for functions with parameters of primitive type. Following their work, researchers have proposed novel SBST techniques for automating functional testing [18], test case prioritization [39, 68], mutation testing [36, 64], as well as regression testing [37, 40]. These approaches leverage meta-heuristic algorithms [15] to cover various test goals (e.g., branch coverage, path coverage, use-def coverage, etc.) with different search strategies (hill-climbing algorithm,

genetic algorithm, etc) [10, 20, 27, 48, 65], fitting test representations [14, 20, 31, 74], and the fitness metrics [14, 31, 41, 56, 74, 78]. Readers can refer to survey papers [33, 46] for more details.

6.2 SBST for Object Oriented Programs

Search-based software testing on object oriented programs is a long standing problem [67] with many different proposed solutions [13, 17, 23, 25, 60, 73]. Wappler et al. [73] used a tree-based representation of method call sequence to ensure that the generated tests are compilation-feasible. Arcuri et al. [13] proposed an approach to address the flag problem via testability transformation caused by container objects such as `List` and `Collections`. Sakti et al. [60] proposed to construct object instances with more diversity to improve upon SBST algorithms. The state-of-the-art SBST tool *EvoSuite* [25] incorporates all of the above techniques. Nevertheless, constructing appropriate objects with appropriate attributes remains challenging.

Braione et al. [17] proposed an approach to use object constraints generated by symbolic execution as the search objective, but this work is limited by the symbolic execution engine and has only been evaluated on a limited number of Java classes. *EvoObj* uses static analysis to construct object construction graph and generate a test template where mutation brings evolving gradients, facilitating the performance of SBST.

Aleti et al. [11] and Albunian et al. [9] investigated the fitness landscape for the SBST application in unit test generation, and showed that the most problematic landscape feature is the presence of many plateaus. The object input problem is a major contributor for these challenging fitness landscapes, as traditional fitness measurements are not sensitive to the mutations of changing the object state. In this work, we leverage static analysis to create a "shortcut" in that landscape so that the measurements such as branch distances can take effect.

6.3 Seed Generation Technique for SBST

Seeding refers to the inclusion of external information and solutions into the population of a search algorithm. Rojas et al. [57] discussed a variety of seeding strategies, including the use of static and dynamic constants as seeds for evolution, which achieves significant performance improvements. Following their idea, Anjum et al. [12] harvest the runtime constants to improve SBST for grammatical evolution. Moreover, Liu et al. [44] adopted domain knowledge to construct structural data for fuzzing Android native system services. The idea of OCAT [35] is to capture objects at runtime and reuse then during test generation; however, these objects are deserialized rather than constructed in the generated tests, and thus limit possibilities for mutation and maintenance. An alternative appraoch consists of mining sequences of method calls and information about common usage from existing executions [29, 66].

In contrast, our approach is a general approach for object construction by exploring data and control flow with interprocedural analysis. Moreover, our approach is also complementary with existing state-of-the-art seed generation approaches.

6.4 Object Construction

Constructing legitimate complex structural and object inputs to achieve higher coverage is one of the central problems in test generation, which applies to many programming languages. Existing solutions include SBST heuristics [13, 25, 53, 60, 73], symbolic execution [16, 34], and separation logics [54, 55, 79]. SBST heuristics define rules to explore more subclasses and diversify the object construction means [13, 25, 53, 60, 73]. Nevertheless, constructing more diversified types of objects can hardly address the issue of constructing legitimate complicated input and answer the questions like (1) "*which attributes of the object input should be considered?*" and (2) "*how many layers of object attributes should be constructed?*". Symbolic execution and separation logic extract object constraints and solve them by SMT solvers, but a challenge lies in reconstructing sequences of calls that configure objects according to these constraints. The SUSHI approach [16, 17] aims to transform object constraints extracted via symbolic execution into SBST fitness functions, and uses search to synthesize test cases. However, so far this approach has only been studied on 8 Java classes [16].

7 CONCLUSIONS AND FUTURE WORK

Although search-based unit test generation has resulted in mature tools, the limited ability of the resulting tests to capture real faults [61] implies a need to further improve SBST techniques. In order to address this problem, we considered the issue of generating complex object instances, which is considered an essential factor that affects the challenging fitness landscape in search-based unit test generation [9, 11]. Our approach synthesizes seeds by constructing an object construction graph for every branch and then synthesizing a test template based on the graph, such that resulting objects are more likely to represent valid objects. This enables traditional fitness metrics to better guide search algorithms, as our experiments confirm.

In future work, we will extend our work in the following aspects. First, we will improve the efficiency of the graph-based test code synthesis for *EvoObj* by minimizing the generated test cases of large object construction graph, integrate our solution into more algorithms (e.g., memetic algorithms [28] and hybrid algorithm [72]), and further generalize our experiment. Second, we will further investigate *EvoObj*'s capability to discover more software faults and integrate it with a set of debugging techniques [42, 43, 69, 71, 76, 77]. Third, we will apply our idea of synthesizing structural input in fuzzing scenarios to discover more binary software vulnerabilities [21, 38, 45, 49, 70, 75, 80].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable input to improve our work. This work was supported in part by the Minister of Education, Singapore (No. T2EP20120-0019 and No. T1-251RES1901), the National Research Foundation Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office (Award Number: NSOE-TSS2019-03 and NSOE-TSS2019-05), and by EPSRC project EP/N023978/2.

REFERENCES

[1] [n.d.]. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2021-05-30.

[2] [n.d.]. EvoObj Website. https://sites.google.com/view/evoobj/home. Accessed: 2021-02-28.

[3] [n.d.]. Java Bytecode Instruction. https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings. Accessed: 2021-02-28.

[4] [n.d.]. JEdit. http://www.jedit.org/. Accessed: 2021-05-30.

[5] [n.d.]. JFreeChart. https://www.jfree.org/index.html. Accessed: 2021-05-30.

[6] [n.d.]. National Cybersecurity R&D Lab. https://ncl.sg/. Accessed: 2021-02-28.

[7] [n.d.]. SF100 Benchmark. https://www.evosuite.org/experimental-data/sf100/. Accessed: 2021-02-28.

[8] [n.d.]. Weka. https://www.cs.waikato.ac.nz/ml/weka/. Accessed: 2021-05-30.

[9] Nasser Albunian, Gordon Fraser, and Dirk Sudholt. 2020. Causes and effects of fitness landscapes in unit test generation. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. 1204–1212.

[10] Aldeida Aleti and Lars Grunske. 2015. Test Data Generation with a Kalman Filter-based Adaptive Genetic Algorithm. *J. Syst. Softw.* 103, C (May 2015), 343–352.

[11] Aldeida Aleti, I. Moser, and Lars Grunske. 2017. Analysing the Fitness Landscape of Search-based Software Testing Problems. *Automated Software Engg.* 24, 3 (Sept. 2017), 603–621.

[12] Muhammad Sheraz Anjum and Conor Ryan. 2020. Seeding Grammars in Grammatical Evolution to Improve Search Based Software Testing. In *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 18–34.

[13] Andrea Arcuri and Xin Yao. 2008. Search based software testing of object-oriented containers. *Information Sciences* 178, 15 (2008), 3075–3095.

[14] André Baresel, Harmen Sthamer, and Michael Schmidt. 2002. Fitness Function Design to Improve Evolutionary Structural Testing. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation* (New York City, New York) *(GECCO'02)*. 1329–1336.

[15] Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J Gutjahr. 2009. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing* 8, 2 (2009), 239–287.

[16] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining symbolic execution and search-based testing for programs with complex heap inputs.. In *ISSTA*. 90–101.

[17] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2018. SUSHI: a test generator for programs with complex structured inputs. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 21–24.

[18] Oliver Bühler and Joachim Wegener. 2008. Evolutionary Functional Testing. *Comput. Oper. Res.* 35, 10 (Oct. 2008), 3144–3160.

[19] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.

[20] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2018. Search-based Test Data Generation for SQL Queries. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. 1220–1230.

[21] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108.

[22] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 196–206.

[23] Giovanni Denaro, Alessandro Margara, Mauro Pezze, and Mattia Vivanti. 2015. Dynamic data flow testing of object oriented systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 947–958.

[24] Michael P Fay and Michael A Proschan. 2010. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys* 4 (2010), 1.

[25] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[26] Gordon Fraser and Andrea Arcuri. 2012. The seed is strong: Seeding strategies in search-based software testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 121–130.

[27] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Softw. Eng.* 39, 2 (Feb. 2013), 276–291.

[28] Gordon Fraser, Andrea Arcuri, and Phil McMinn. 2015. A memetic algorithm for whole test suite generation. *Journal of Systems and Software* 103 (2015), 311–327.

[29] Gordon Fraser and Andreas Zeller. 2011. Exploiting common object usage in test case generation. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 80–89.

[30] M. Harman. 2007. The Current State and Future of Search Based Software Engineering. In *Future of Software Engineering (FOSE '07)*. 342–357.

[31] M. Harman and J. Clark. 2004. Metrics are fitness functions too. In *10th International Symposium on Software Metrics, 2004. Proceedings.* 58–69.

[32] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3–16.

[33] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. 2012. Empirical Software Engineering and Verification. Chapter Search Based Software Engineering: Techniques, Taxonomy, Tutorial, 1–59.

[34] Kobi Inkumsah and Tao Xie. 2008. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 297–306.

[35] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. 2010. OCAT: object capture-based automated testing. In *Proceedings of the 19th international symposium on Software testing and analysis*. 159–170.

[36] Y. Jia and M. Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. 249–258.

[37] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang NA Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (2018), 74–93.

[38] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 533–544.

[39] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering* 33, 4 (2007), 225–237.

[40] Z. Li, M. Harman, and R. M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering* 33, 4 (2007), 225–237.

[41] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering fitness gradients for interprocedural Boolean flags in search-based testing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 440–451.

[42] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. 2018. Break the dead end of dynamic slicing: Localizing data and control omission bug. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 509–519.

[43] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 393–403.

[44] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. 2020. {FANS}: Fuzzing Android Native System Services via Automated Interface Analysis. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[45] Kulani Mahadewa, Yanjun Zhang, Guangdong Bai, Lei Bu, Zhiqiang Zuo, Dileepa Fernando, Zhenkai Liang, and Jin Song Dong. 2021. Identifying Privacy Weaknesses from Multi-party Trigger-action Integration Platforms. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA.

[46] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156.

[47] W. Miller and D. L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *IEEE Trans. Softw. Eng.* 2, 3 (May 1976), 223–226.

[48] Duy Tai Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Minh Quang Tran. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. 1–12.

[49] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[50] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[51] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[52] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[53] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. 1999. Test-data generation using genetic algorithms. *Software testing, verification and reliability* 9, 4 (1999), 263–282.

[54] Long H Pham, Quang Loc Le, Quoc-Sang Phan, and Jun Sun. 2019. Concolic testing heap-manipulating programs. In *International Symposium on Formal Methods*. Springer, 442–461.

[55] Long H Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2019. Enhancing symbolic execution of heap-based programs with separation logic for test input generation. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 209–227.

[56] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *Proceedings of the 7th International Symposium on Search-Based Software Engineering (SSBSE '15)*. Springer, 93–108.

[57] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401.

[58] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893.

[59] Urko Rueda, Tanja EJ Vos, and ISWB Prasetya. 2015. Unit testing tool competition–round three. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*. IEEE, 19–24.

[60] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. 2014. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering* 41, 3 (2014), 294–313.

[61] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.

[62] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. 2015. Random or genetic algorithm search for object-oriented test suite generation?. In *Proceedings of the 2015 annual conference on genetic and evolutionary computation*. 1367–1374.

[63] Sina Shamshiri, José Miguel Rojas, Luca Gazzola, Gordon Fraser, Phil McMinn, Leonardo Mariani, and Andrea Arcuri. 2018. Random or evolutionary search for object-oriented test suite generation? *Software Testing, Verification and Reliability* 28, 4 (2018), e1660.

[64] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sérgio Lopes de Souza. 2017. A systematic review on search based mutation testing. *Information and Software Technology* 81 (2017), 19–35.

[65] Anupama Surendran and Philip Samuel. 2017. Evolution or Revolution: The Critical Need in Genetic Algorithm Based Testing. *Artif. Intell. Rev.* 48, 3 (Oct. 2017), 349–395.

[66] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Zhendong Su. 2011. Synthesizing method sequences for high-coverage testing. *ACM SIGPLAN Notices* 46, 10 (2011), 189–206.

[67] Paolo Tonella. 2004. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 119–128.

[68] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. 2006. TimeAware Test Suite Prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (Portland, Maine, USA) *(ISSTA '06)*. 1–12.

[69] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jin Song Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering* (2019).

[70] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 999–1010.

[71] Haijun Wang, Xiaofei Xie, Shang-Wei Lin, Yun Lin, Yuekang Li, Shengchao Qin, Yang Liu, and Ting Liu. 2019. Locating vulnerabilities in binaries via memory layout recovering. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 718–728.

[72] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*. 291–302.

[73] Stefan Wappler and Joachim Wegener. 2006. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. 1925–1932.

[74] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information & Software Technology* 43, 14 (2001), 841–854.

[75] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 765–777.

[76] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. 2018. Machine translation-based bug localization technique for bridging lexical gap. *Information and Software Technology* 99 (2018), 58–61.

[77] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology* 105 (2019), 17–29.

[78] Xiong Xu, Ziming Zhu, and Li Jiao. 2017. An Adaptive Fitness Function Based on Branch Hardness for Search Based Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Berlin, Germany) *(GECCO '17)*. 1335–1342.

[79] Guolong Zheng, Quang Loc Le, ThanhVu Nguyen, and Quoc-Sang Phan. 2018. Automatic data structure repair using separation logic. *ACM SIGSOFT Software Engineering Notes* (2018).

[80] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. 2019. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 772–784.