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

Yun Lin
dcslny@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:


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 op1 as the returned value of the call of getAwardNum(), and operand op2 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:

\[
\text{distance}(b, t) = \begin{cases} 
op2 - \text{op1} + K & \text{if } b.c \text{ evaluates to false} \\ 0 & \text{otherwise} \end{cases}
\]

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 op1 is 0, and the valuation of op2 (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. `return 1;`
4. `}`
5. `class Student{Supervisor s; ...}`
6. `class Supervisor{CV cv; ...}`
7. `class CV{int awardNum; ...}`

Listing 1: A Branch Relevant to Object States

```
Student s = new Student();
Supervisor obj1 = new Supervisor();
CV obj2 = new CV();
int a0 = 8;
obj1.setAwardNum(a0);
obj2.setCV(obj1);
obj1.setSupervisor(obj2);
example(s);
```

(a) Initial Test $t_{ini}$

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.

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 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), 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 significantly 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 object input according (see Figure 4). The test templates are then generated through traversing the object construction 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.

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 $t_{ini}$ towards the global optimum.

According to Listing 1, the branch distance will be normalized by some function as $f(n) = \frac{n - n_{min}}{n_{max} - n_{min}}$. Thus, for the example code of Figure 2b, we have the branch distance as $3 - 0 = 3$, which will be further normalized to $\frac{3}{3} = 1.0$. In this paper, we propose a systematic and practical approach to synthesize object inputs by analyzing static control and data flow to significantly 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 object input accordingly (see Figure 4). The test templates are then generated through traversing the object construction 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.
Listing 2: Target Method `checkRules()`

We have 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 implement and evaluate this approach in the context of Java, the approach 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, compared to `EvoSuite`, `EvoObj` achieved superior performance improvement on branch coverage 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` array 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 transitive relevant data fields. Figure 4...
Figure 4: Example of graph-based test template generation. In the object construction graph, the grey rectangles represents graph inputs; the red rectangles represents fields/get-field operations; the green rectangles represent call/call operations; and the blue rectangles represent array elements/get-array-element operation. 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 APPRAACH

In this section, we describe in detail (1) how object construction graphs are created, (2) how code is synthesized from these graphs, 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 $G(b) = (I, N, E)$, 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 action() 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 player’s 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 can represent both an operation and its output variable. Each leaf node $m_l$ 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. The backwards slicing process starts from the target branch $b$, and ends at instructions satisfying either of the 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 for limiting the performance overhead when conducting interprocedural analysis. That is, once there is a call chain from $m_t$ to $m_k$, for example $m_t \rightarrow m_1 \rightarrow \ldots m_k$, we let $k \leq t_{dep}$.

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 branch represented as an if_icmpeq instruction. We refer to the JVM documentation [3] for more details on the semantics of Java bytecode instructions.

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 an 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 alaload 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() method is called from the object state of type GameState. In such a call, the player’s 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

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 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.

**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,()) → 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 as input and generates a test template 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 OCG is private, we need to search for the most promising setter/getter function in the project to set/use it in the test code.

**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 `Nd = {n1, n2, ..., nm}` in OCG, and use the node-statement map to locate their corresponding statements `code(Nd) = {code(n1), code(n2), ..., code(nm)}` in the test. Each defined variable of `code(i)` (i is 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 can know
what role (e.g., caller object, method parameter, operand of assignment, etc) of each defined variables play in synthesizing a new statement.

For the example in Figure 4, the graph node \( \text{player()} \) 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 \( \text{player()} \) 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 \( \text{obj} \). First, we scan all the methods accessible by \( \text{obj} \) in its class (and superclass), and build their call graph. Given a path in the call graph \( p = (m_0, ..., m_e) \) where \( m_0 \) 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 \( p = (m_0, ..., m_e) \), we then estimate its likelihood to set the field by (1) the number of method inputs in \( m_i, N_o \), 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 \( \text{comp}(m_i) \) 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 \( B, \text{comp}(m_i) = |B'| / |B| \). As a result, we use Equation 1 to assign the score for each path, where \( |p| \) means the length of \( p \):

\[
\text{score}(p) = (N_o + 1) \times \frac{\sum_{i=1}^{\text{comp}(m_i)}}{|p|} \quad (1)
\]

\(^3\)Here, we regard array element as a special form of field of an array object.

---

![Figure 6: An Example of Setter Selection](image)

Given a method \( m \) serving as the starting node for multiple interesting paths \( \{p_1, ..., p_n\} \), we evaluate its score \( \text{score}(m) \) as \( \max(\text{score}(p_1), ..., \text{score}(p_n)) \). 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:

- \( p_1 = \langle m_1, m_2, m_3 \rangle \), \( \text{score}(p_1) = (1 + 1) \times \frac{1 + 1 + 1}{3} = 1.67 \);
- \( p_2 = \langle m_2, m_3 \rangle \), \( \text{score}(p_2) = (1 + 1) \times \frac{1 + 1}{2} = 2 \);
- \( p_3 = \langle m_1, m_3 \rangle \), \( \text{score}(p_3) = (1 + 1) \times \frac{1 + 1}{2} = 2 \);
- \( p_4 = \langle m_3 \rangle \), \( \text{score}(p_4) = (1 + 1) \times \frac{1}{2} = 2 \);
- \( p_5 = \langle m_4 \rangle \), \( \text{score}(p_5) = (0 + 1) \times \frac{1}{2} = 1 \).

As a result, \( \text{score}(m_1) = \max(1.67, 2) = 2, \text{score}(m_2) = 2, \text{score}(m_3) = 2, \text{score}(m_4) = 1. \) Thus, we can normalize their score to the sampling probability as \( p(m_1) = p(m_2) = p(m_3) = 2/7 = 28.6\% \) while \( p(m_4) = 1/7 = 14.3\% \).

### 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 to 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 \( p_{\text{app}} \).

### 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.

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].

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

<table>
<thead>
<tr>
<th>Coverage Performance</th>
<th>Budget-100s (%)</th>
<th>Budget-150s (%)</th>
<th>Budget-200s (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DynaMOSA</td>
<td>MOSA</td>
<td>MonotonicGA</td>
</tr>
<tr>
<td>EvoObj</td>
<td>78.82</td>
<td>77.35</td>
<td>67.22</td>
</tr>
<tr>
<td>EvoSuite</td>
<td>70.35</td>
<td>69.36</td>
<td>62.15</td>
</tr>
<tr>
<td>p-value</td>
<td>&lt;10E-10</td>
<td>&lt;10E-10</td>
<td>&lt;10E-10</td>
</tr>
<tr>
<td>Effect size</td>
<td>0.33</td>
<td>0.36</td>
<td>0.36</td>
</tr>
</tbody>
</table>

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.

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

Table 2: Runtime Overhead on Different SBST Algorithms

<table>
<thead>
<tr>
<th>Approach</th>
<th>DynaMOSA (s)</th>
<th>MOSA (s)</th>
<th>MonotonicGA (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EvoObj</td>
<td>41.68</td>
<td>36.36</td>
<td>153.33</td>
</tr>
<tr>
<td>EvoSuite</td>
<td>44.92</td>
<td>44.39</td>
<td>152.43</td>
</tr>
<tr>
<td>p-value</td>
<td>&lt;10E-10</td>
<td>&lt;10E-10</td>
<td>0.13</td>
</tr>
<tr>
<td>Effect size</td>
<td>0.09</td>
<td>0.26</td>
<td>-0.02</td>
</tr>
</tbody>
</table>

Listing 5: Complicated containment relationship

```java
public boolean equals(Object obj){
    if(this == obj) return true;
    if(obj instanceof CustPayeeModRsSequence2){
        CustPayeeModRsSequence2 temp = (CustPayeeModRsSequence2)obj;
        if(_custPayeeId != null) {
            if(temp._custPayeeId == null) return false;
            if(!_custPayeeId.equals(temp._custPayeeId)) return false;
        } else if(temp._custPayeeId != null) return false;
        if(_custPayeeInfo != null){...}
    } else if(temp._custPayeeId != null) return false;
    if(_SPRefId != null){...}
    else if(_SPRefId != null)
        return false;
    ...
}
```

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 and makes the branch distance based searching of SBST algorithms more effective. This is more impressive to consider that there is 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 the test templates takes more time at runtime, which reduces the number of iterations of the evolutionary algorithm as there is a limited time budget.
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 *get field* 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 of 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 them 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 approach 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 anonymous reviewers for their valuable input to improve our work. This work was supported in part by the Minister of Education, Singapore (No. T2EP0120-0019 and No. T1-251RES1901), the National Research Foundation Singapore through its National Salary 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


