Empirical Study of Effectiveness of EvoSuite on the SBST 2020 Tool Competition Benchmark

Robert Sebastian Herlim[1], Shin Hong[2], Yunho Kim[3], and Moonzoo Kim[1]

[1] KAIST, Daejeon, South Korea
robert.herlim@kaist.ac.kr, moonzoo@cs.kaist.ac.kr
[2] Handong Global University, Pohang, South Korea hongshin@handong.edu
[3] Hanyang University, Seoul, South Korea yunhokim@hanyang.ac.kr

Abstract. EvoSuite is a state-of-the-art search-based software testing tool for Java programs and many researchers have applied EvoSuite to achieve high test coverage. However, due to high complexity of object-oriented programs, EvoSuite still suffers several limitations in terms of test coverage achievement. In this paper, to improve the effectiveness of EvoSuite by analyzing EvoSuite's limitations, we conducted an empirical study to identify the limitations of EvoSuite on the most recent SBST 2020 Tool Competition benchmark that consists of 70 classes selected from real-world Java projects. We have manually classified the branches of the target programs that EvoSuite could not cover and reported corresponding limitations of EvoSuite with concrete examples.

Keywords: Empirical study · EvoSuite · SBST Tool Competition

1 Introduction

Automated test case generation has been a prominent research topic for the past decade [3,6,8–10]. Among several automated test generation techniques, search-based software testing attracts researchers for its high scalability and high test coverage. EvoSuite [6] is a state-of-the-art search-based software testing tool for Java programs and many researchers have used EvoSuite to detect faults in real-world industrial cases [1,14] and achieve high test coverage [7,13].

In this paper, to improve the effectiveness (i.e., test coverage achievement) of EvoSuite by analyzing the EvoSuite's limitations, we conducted an empirical study by applying EvoSuite to the SBST 2020 Tool Competition [5] benchmark (calling it the SBST 2020 benchmark).[4] We replicated the contest setting used by EvoSuite during the competition to obtain its coverage report. From the report, we manually analyzed each branch that EvoSuite could not cover and grouped them into a few categories.

[4] SBST 2020 Tool Competition benchmark was the latest available dataset in the annual SBST Tool Competition series by the time we performed this study. The SBST 2020 benchmark consists of 70 classes selected from real-world open-source Java projects. The contest infrastructure for replicating the SBST competition series is available at `https://github.com/JUnitContest/JUGE`.

Table 1: Target subjects & EvoSuite's achieved coverage

Subject	#Classes	#Branches				Br. Coverage (%)	
		Total	Mean	Median	Stdev	Mean	Stdev
FESCAR	20	490	24.5	11	27.3	66.9	42.9
GUAVA	20	926	46.3	19	61.5	67.5	32.4
PDFBOX	20	1,070	53.5	24	75.6	54.7	36.1
SPOON	10	1,072	107.2	45	12.9	28.0	24.5

There have been several attempts to study the limitations of EvoSuite in previous works, such as categorizing the kinds of hard-to-detect faults [2] and bad quality tests [12]. To the best of our knowledge, this is the first attempt to identify the limitations of EvoSuite exclusively for categorizing reasons of not-covered branches in the SBST 2020 benchmark. Compared with the original EvoSuite's post-mortem report [11], this paper shows a clearer picture of the challenges of EvoSuite on the SBST 2020 benchmark by reporting the limitations with concrete examples. The main contributions of this paper are as follows:

1. We performed an empirical study by applying EvoSuite to the recent SBST 2020 benchmark, from which we identified several limitations that EvoSuite struggles with.
2. We extensively analyzed the branches that EvoSuite could not cover in the benchmark and classified the corresponding reasons for those branches, which have not been done previously by other empirical study papers.

The remaining sections are organized as follows: Section 2 describes the study questions and the empirical study setup. Section 3 reports the answers to the study questions. Finally, Section 4 concludes this paper with future work.

2 Empirical Study Setup

2.1 Benchmark Overview

The SBST 2020 Tool Competition [5] benchmark contains 70 different classes selected from the following four real-world open-source projects:

- **FESCAR**: an open-source distributed transaction library to support transactions in microservice.
- **GUAVA**: a common Java library developed by Google which provides collection classes.
- **PDFBOX**: a PDF processing library which provides PDF manipulation utilities, such as text extraction, splitting, merging, and document signing.
- **SPOON**: a library for Java source code analysis and transformation.

Table 1 describes the benchmark subjects and the branch coverage on each subject achieved by EvoSuite following the analysis procedure (see Section 2.2).

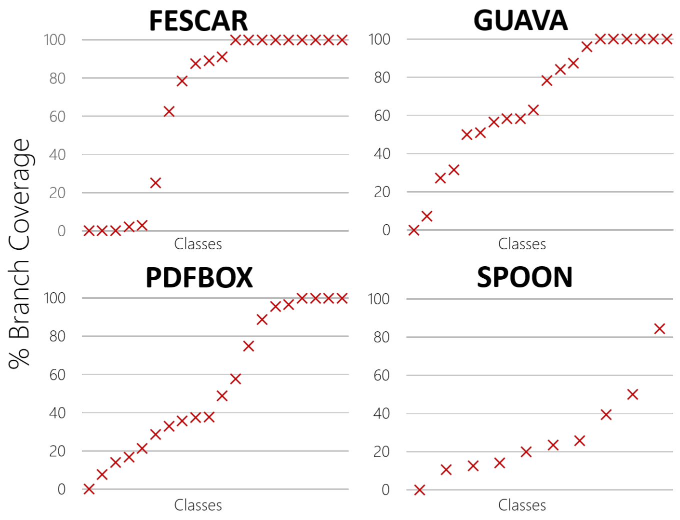

Fig. 1: EvoSuite's achieved branch coverage per class. Each data point represents the average score of EvoSuite's achieved branch coverage in a class of the respective subject.

For example, FESCAR (see the second row of the table) has 20 classes that contain total 490 branches and EvoSuite achieved 66.9% branch coverage on FESCAR on average in the experiment. Note that the benchmark does not provide any detail of fault existence in the target classes, so fault detection is beyond the scope of our study.

Figure 1 shows the scatter plot of class coverage on each subject. For example, GUAVA has four classes on which EvoSuite achieved around 60% branch coverage (see the four data points in the middle part of the GUAVA graph) and six classes on which EvoSuite achieved 100% branch coverage (see the rightmost six data points in the GUAVA graph).

2.2 Analysis Procedure

The experiments were conducted on one machine equipped with octa-core AMD Ryzen 7 1700 (up to 3.7 GHz) and 16GB RAM, running a 64-bit version of Ubuntu `16.04`. We used OpenJDK `v1.8.0` for the Java SE and Maven `v3.3.9` for the project build tool. We used the official Docker infrastructure in the version with commit hash `2ed9d22`. Except the post-processing part[5], we followed the EvoSuite's contest configurations to use EvoSuite's default configurations. Three minutes was set for the time budget, consisting 50% for the search and 50% for other remaining part. JaCoCo was used to measure the branch coverage, and by which we generated coverage report (using HTML format) to analyze the

[5] For this independent replication study, we disabled the test suite minimization step to reduce the risk of coverage loss caused by unintended test case reduction.

Table 2: Reasons/limitations of the not-covered branches

	Category	Description	Mnemonic
1	**<u>Construction problem</u>**		
1A	Construction failure	EvoSuite failed to construct CUT (Class Under Test)	C-CF
1B	Complex construction	CUT can be instantiated, but in non-trivial ways	C-CC
2	**<u>OO-related problem</u>**		
2A	Private access on method	A method protected by `private` keyword	O-PAM
2B	Inheritance instantiation	A method needs particular subclasses as argument	O-IIP
2C	`Class<?>` as an argument	A branch condition checks on special `Class<?>` type	O-CLA
2D	Inner class method invocation	A method is only callable through its inner classes	O-ICM
3	**<u>Large search space problem</u>**		
3A	Incomprehensive method testing	A method with simple arguments, but left untested	L-IMT
3B	Specific value in iterable/stream	Needs specific values in byte/string/input stream	L-ISV
3C	Key-value store pattern	Uses a dictionary (e.g., `java.util.Map`)	L-KVS
3D	Obj. state in an argument	An argument state needs to be further modified	L-OSA
3E	Obj. state in invoking object	An object state needs to be further modified	L-OSI
4	**<u>Other problem</u>**		
4A	File system access	A method performs operations on file system	FSA
4B	JVM's `System.getProperty` call	A method checks on env. variable stored in JVM	JSC
4C	Branch unreachable	Unfeasible branches by program executions	UBR

Listing 1.1: Example of *reached but not covered* branch

```
L1:void f(int x,int y) {
L2:  ...
L3:  if ( x>0 ) { /* br1 */
L4:  } else {     /* br2 */
L5:    if ( y>0 ) { /* br3 */}}}
```

not-covered branches. To limit the random variance, we performed six repeated experiment runs on each subject. We counted a branch b as not-covered if b was not covered by any of the six experiment runs.

Note that we investigated the *reached but not covered* branches only. For Listing 1.1, with an input `(x,y)=(1,1)` for `f()`, the branch `br1` in L3 is reached and covered, `br2` in L4 is reached but not covered, and `br3` in L5 is not reached (and, thus, not covered). In other words, we did not analyze the branches that were not reached by any test input because the reason for not covering such unreached branches can be complex to classify. For example, the reason of why `br3` was not covered with `(x,y)=(1,1)` does not only depend on L5, but its all predecessor conditions such as L2 and so on. We could hypothesize that the `br3` coverage failure was caused by "argument `x` was always > 0" (the same cause as failure in `br2`), but it may not be true because another input where `(x,y)=(0,0)` would neither make `br3` to be covered.

From now on, we use the term *not-covered* to represent *reached but not covered* branches in this paper. We manually analyzed total 359 not-covered branches in the SBST 2020 benchmark. The raw data is accessible through `https://bit.ly/evoStudySSBSE2021RENE`.

2.3 Study Questions

After manually analyzing *reached but not covered* branches by EvoSuite, we classified the limitations of EvoSuite on the SBST 2020 benchmark into four groups (i.e., construction problem, OO-related problem, large search space problem, and other problem) of the 14 categories as shown in Table 2. For the empirical study, we made the following four study questions:

SQ 1 Object Construction Problem. How much do the object construction problems impact EvoSuite's branch covereage?

SQ 2 OO-related Problem. How much do the OO-related problems impact EvoSuite's branch coverage?

SQ 3 Large Search Space Problem. How much do the large search space problems impact EvoSuite's branch coverage?

SQ 4 Other Problem. How much do the environment-related problem and unreachable branches impact EvoSuite's branch coverage?

Second, we tried to identify *common major* limitations of EvoSuite on the SBST 2020 benchmark. The term *common* is used on the limitations that are observed on all four target subjects of the benchmark. Such common problems are particularly interesting because they may be general limitations that apply to not only SBST 2020 benchmark but also other programs. We define *common* and *major* limitations as follows:

– A category becomes a *common* problem if its occurrences can be found across all four subjects in the benchmark.
– A category becomes a *major* problem (in one subject) if it appears in at least 25% of the classes with not-covered branches (i.e., classes with < 100% branch coverage).

SQ 5 Common Major Problems. By running EvoSuite on the SBST 2020 Tool Competition benchmark, is there any common major problem found across all target subjects?

3 Empirical Study Results

Figure 2 shows the distribution of the limitation categories of the branches that EvoSuite could not cover (see Table 2). For example, the 11 branches of FESCAR and the 20 branches of GUAVA were not covered by EvoSuite due to the construction failure (C-CF) and the complex construction (C-CC) respectively (see the bar of light blue color (FESCAR) and the bar of green color (GUAVA) of the two leftmost bars (C-CF and C-CC) in Figure 2).

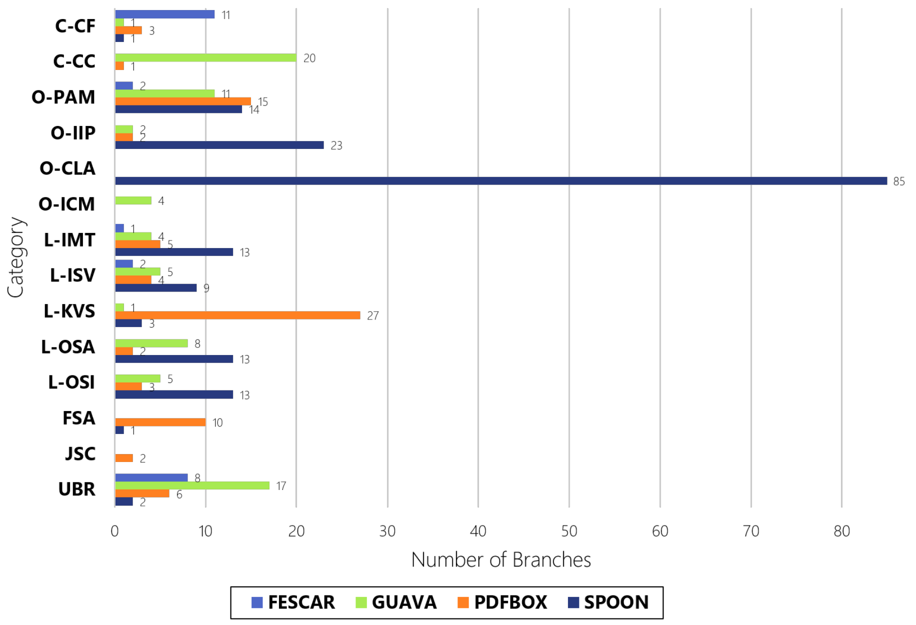

Fig. 2: Distribution of the limitation categories of not-covered branches

3.1 SQ1: Object Construction Problem

1A Construction failure (C-CF). The C-CF category indicates that EvoSuite fails to instantiate an instance of CUT (class under test). Several interesting causes of C-CF in the benchmark are as follows:

1. **Constructor needs complex arguments**:
   For example, consider a constructor [6] of `PDVisibleSignDesigner` (`PDFBOX-130`) shown in Listing 1.2. The constructor needs: (1) a `PDDocument`-typed `document` with p pages ($p > 0$); (2) an `InputStream`-typed `imageStream` transformed from any valid `BufferedImage`; and (3) an integer `page` where `page` $\in (0, p]$. Violating any of the constraints (e.g., `page` $= 0$, `null imageStream` or empty `InputStream` such as returned by `new ByteArrayInputStream(new byte[5])`) would result an exception raised during the execution. Due to these constraints, we found EvoSuite could not construct a `PDVisibleSignDesigner` instance during the runs.
   Another example is `FilteredEntryMultimap` (`GUAVA-47`), where its constructor needs a `Predicate`-typed instance. We found in eight out of 12 test cases where EvoSuite used a `BloomFilter` instance as an argument of a `Predicate`-typed instance, in which six out of the eight test cases failed (i.e., raising exceptions) during the `BloomFilter` construction. This indicates that generating a valid `Predicate` instance is highly complex for EvoSuite. We observed

[6] We observed that other six `PDVisibleSignDesigner` constructors have a similar construction failure problem as described in this paper.

Listing 1.2: Construction failure in PDFBOX-130

```
L1: public PDVisibleSignDesigner(PDDocument document,
L2:   InputStream imageStream, int page) throws IOException {
L3:   readImageStream(imageStream);
L4:   calculatePageSize(document, page); }
```

that such failures in constructing `BloomFilter` induced failure constructions of `FilteredEntryMultimap`, which causes testing could not reach any other member methods that needs at least one `FilteredEntryMultimap` instance as target object.

2. **Dependency on another C-CF class**:
   For example, the constructor of `FilteredMultimapValues` (`GUAVA-240`) needs a `FilteredMultimap`-typed argument. Every subclass of `FilteredMultimap` needs a `Predicate` instance (a complex object as described above), causing EvoSuite to fail at constructing a `FilteredMultimapValues` instance. As a result, all seven related member methods were not covered.
3. **Accessibility conflict on constructor's arguments**:
   For example, the constructor of `PDTrueTypeFontEmbedder` (`PDFBOX-235`) receives a `TrueTypeFont`-typed instance as a parameter. EvoSuite could not generate a `PDTrueTypeFontEmbedder` instance since `TrueTypeFont` is package-private and located in a different package.
4. **Missing class from the classloader**:
   The classloader could not find a constructor parameter type (e.g., `FESCAR-6`, `FESCAR-8`, `FESCAR-15`, `FESCAR-41`, `SPOON-155`), which induced an exception (i.e., `NoClassDefFoundError`). [7]

1B Complex construction (C-CC). Compared to the C-CF category where object creation entirely fails, this C-CC category applies to the CUTs with successful instantiation, but of only *simple* objects. Several interesting causes of C-CC in the benchmark are as follows:

1. **Implicit object construction convention**:
   For example, the only way to construct `ImmutableEnumSet` (`GUAVA-206`) is by invoking the *static factory* method `asImmutable` as shown in Listing 1.3. However, `asImmutable` may not generate a `ImmutableEnumSet` instance (but `ImmutableSet`-typed instead) if the given `set` argument's size is ≤ 1 (L3–L5). To mitigate this case, EvoSuite should infer how to obtain the correct `ImmutableEnumSet` instance by invoking `asImmutable` with an `EnumSet` instance of > 1 elements (L6).
2. **Dependency to other unknown class**:
   Another example is `Graphs` (`GUAVA-22`), whose methods perform graph operations (e.g., transposing, finding reachable nodes) on `Graph` instances. For

[7] We suspect that these limitations were caused by a bug in EvoSuite since those missing classes had been correctly placed in the same directory and package as the CUT.

Listing 1.3: Complex construction problem in GUAVA-206

```
L1: static ImmutableSet asImmutable(EnumSet set) {
L2:   switch (set.size()) {
L3:     case 0: return ImmutableSet.of();
L4:     case 1: return ImmutableSet.of(
L5:       Iterables.getOnlyElement(set));
L6:     default: return new ImmutableEnumSet(set); } }
```

Listing 1.4: Inheritance instantiation problem in SPOON-105

```
L1: // To cover: pass a CtCatch instance to the first argument
L2: SourcePosition buildPositionCtElement(CtElement e, ...) {
L3:   if (e instanceof CtCatch) { // then branch was not covered
L4:     return SourcePosition.NOPOSITION;
L5:   } ... }
```

this case, we found EvoSuite generated method call sequences using *empty* `Graph` instances. That is because the standard construction (i.e., through available constructors) produces empty `Graph` objects by default. To construct more diverse `Graph` objects (e.g., adding nodes/edges, cyclic/acyclic, directed/undirected), EvoSuite has to use a builder class. Since the relationship between the builder class and CUT to construct more complex `Graph` instances is unknown to EvoSuite, EvoSuite failed to cover branches relevant to diverse `Graph` instances.

3.2 SQ2: OO-related Problem

2A Private access on method (O-PAM):
We observed that 42 methods (from 13 different classes) could not be tested due to the private access issue. For example, almost 50% (nine out of 20) of the `PositionBuilder`'s (`SPOON-105`) private methods were not covered.

2B Inheritance instantiation problem (O-IIP):
The O-IIP category is mostly encountered in the form of inheritance-checking conditions, caused by the `instanceof` operator in Java. An example of O-IIP category is shown in Listing 1.4. In the example, the return statement at L4 was not covered because EvoSuite could not satisfy the condition at L3. For example, SPOON suffered from O-IIP severely (i.e., 23 branches in five out of the ten SPOON classes were affected).

2C `Class<?>` as a method argument (O-CLA):
Java provides `java.lang.Class` to represent any class or interface in the application, which is commonly utilized by the factory classes. Having a `Class<?>` parameter enlarges the search space because all classes in the classpath become candidates for the method's arguments. For example, `DefaultCoreFactory` (`SPOON-65`) creates a `CtElement` instance based on the supplied argument `klass`, as shown in Listing 1.5. We found that EvoSuite failed to cover any of the 83 (= 3+80) *then* branches that can be taken if the corresponding *equality* checking branch passes (e.g., L2, L4, L6).

Listing 1.5: `Class<?>` as method argument in SPOON-65

```
public CtElement create(Class<? extends CtElement> klass) {
  if (klass.equals(CtAnnotationFieldAccess.class))
    return createAnnotationFieldAccess();
  if (klass.equals(CtArrayRead.class))
    return createArrayRead();
  if (klass.equals(CtArrayWrite.class))
    return createArrayWrite();
/* ... 80 more similar if-statments */ }
```

Listing 1.6: Inner class method invocation requirement in GUAVA-102

```
L1: // To cover: use NodeIterator's remove API
L2: public class LinkedListMultimap<K, V> {
L3:   private void removeNode(Node<K, V> node) {
L4:     if (node.previous != null) { /* not covered */ } ... }
L5:   ...
L6:   // NodeIterator is an inner class of LinkedListMultimap
L7:   private class NodeIterator {
L8:     public void remove() { ...
L9:       removeNode(current);
L10:    ... }
L11:  } ... }
```

2D Inner class method invocation requirement (O-ICM):
Although inner classes are located inside the CUT, EvoSuite did not generate a method call sequence using methods from the inner classes. However, some of the CUT's methods can only be invocable though the inner classes of the CUT. [8] Listing 1.6 shows an example of O-ICM category in `LinkedListMultimap` (`GUAVA-102`). In `LinkedListMultimap`, a private method `removeNode` (L3) was never invoked because it was invocable only through the `remove` API (L8) of `LinkedListMultimap`'s iterator inner classes, such as `NodeIterator` (L7). Our coverage report showed that although EvoSuite had constructed `NodeIterator` instances (e.g., by generating `valueIterator` call), no further method invocation was performed on the resulted iterator instances. Therefore, `removeNode` was not covered.

3.3 SQ3: Large Search Space Problem

3A Incomprehensive method testing (L-IMT):
For example, EvoSuite did *not* generate a call sequence for a member method `createSerializedForm` in `SparseImmutableTable` (`GUAVA-129`), although the method is declared as public and takes *no* argument. We observed L-IMT also occurred in the *caching* pattern, whose example is shown in Listing 1.7. This is a common pattern to prevent multiple creations of expensive objects (L5) by keeping previously-created reference in the class' field (L6). Through this pattern, the next invocation of `toString` does not invoke `computeToString` if the result

[8] We guess that EvoSuite may consider that generating method call sequences using the methods of the CUT's inner classes would not increase the coverage of CUT since the CUT's inner classes are separate classes from the CUT.

Listing 1.7: Caching pattern inside class in GUAVA-212

```
L1: // To cover: invoke toString() twice.
L2: public String toString() {
L3:   String result = toString;
L4:   if (result == null) {
L5:     result = computeToString();
L6:     toString = result;
L7:   } // The else branch was not covered
L8:   return result; }
```

has been stored in the `toString` member field. The benchmark contains several methods adopting this caching pattern, such as in `MediaType` (`GUAVA-212`), `Graphs` (`GUAVA-22`), and `PDType3Font` (`PDFBOX-265`). EvoSuite rarely performed repeated invocations on such methods, which left the *else* branches (L4 in Listing 1.7) not covered.

3B Specific value in iterable/stream (L-ISV):
The L-ISV category requires some specific input byte sequences to satisfy the branch condition. Several L-ISV examples are as follows:

- The `decode` of `JPXFilter` (`PDFBOX-220`) needs a valid JPEG2000-formatted `InputStream` as an argument. The `decode` calls another method `readJPX`, where `readJPX` will throw an `IOException` if the given `InputStream` in the method argument is not JPEG2000-formatted. In this case, EvoSuite failed to supply the valid `InputStream`, leaving other seven branches in `readJPX` method not reached.
- The `getEndOfComment` of `PositionBuilder` (`SPOON-105`) needs a `char[]` buffer as an argument. The method searches the end-of-comment token (i.e., `'*/'`) in the `char[]` buffer. EvoSuite failed to generate a valid test case to cover the equality checking branch within the `getEndOfComment` method.

3C Key-value store pattern (L-KVS):
The L-KVS category is related to the use of key-value data structure (e.g., `Map`) in branch conditions. Such conditions increase the complexity as they require a correct guess in three dimensions: *key*, *value*, and the *key-value* pair combination. For example, consider `Predictor` (`PDFBOX-117`) as shown in Listing 1.8. In this example, branch in L6 was not covered because `decodeParams` never had an entry for key = `COSName.PREDICTOR` as requested at L5-L6. Note that, to put an item to `COSDictionary`, EvoSuite has to select a key from 517 available `COSName`. We found that L-KVS majorly impacted PDFBOX (i.e., 10 out of 20 classes (50%) suffered L-KVS as shown in Figure 3)

3D Object state problem in argument (L-OSA):
The L-OSA category requires further alteration of the state of the object passed as a method argument. L-OSA examples are as follows:

- The `visitCtClass` of `ImportScannerImpl` (`SPOON-169`):
  As illustrated in Listing 1.9, the body statement in L5 was not covered

Listing 1.8: Key-value store problem in PDFBOX-117

```
L1: // To cover: pass decodeParams argument containing entry
L2: //   { key = COSName.PREDICTOR, value = COSNumber > 1 }
L3: static OutputStream wrapPredictor(OutputStream out,
L4:   COSDictionary decodeParams) {
L5:   int predictor = decodeParams.getInt(COSName.PREDICTOR);
L6:   if (predictor > 1) { /* not covered*/ }
L7:   else { return out; } }
```

Listing 1.9: Argument state problem in SPOON-169

```
L1: // To cover: invoke ctClass.setTypeMembers()
L2: public <T> void visitCtClass(CtClass<T> ctClass) {
L3:   addClassImport(ctClass.getReference());
L4:   for (CtTypeMember t : ctClass.getTypeMembers()) {
L5:     ... // This block was not covered
L6:   } super.visitCtClass(ctClass); }
```

because `getTypeMembers` (L4) called only returned empty iterables. To cover the not-covered branch, `setTypeMembers` invocation on `ctClass` argument was necessary prior to the `visitCtClass` method call (to set the `typeMembers` field of `ctClass` argument). However, EvoSuite did not invoke a such call.

- The `setCount` of `TreeMultiset` (`GUAVA-39`):
  As illustrated in Listing 1.10, the `setCount` method has a *then* branch (L7) which was not covered because EvoSuite did not pass an integer value greater than 0 as an argument to `setCount` to satisfy the branch condition.

3E Object state problem in invoking object (L-OSI):
The L-OSI category requires to alter the CUT's object state further for not-covered branches. Several L-OSI examples are as follows:

- The `addClassImport` of `ImportScannerImpl` (`SPOON-169`) as shown in Listing 1.11. The `addClassImport` has three `if` statements (L4, L5, L6) whose conditions check whether the member field `targetType` is not equal to `null`. All three conditions were not satisfied because the `targetType` field was never got assigned to non-`null` value by EvoSuite. To assign the `targetType` field with non-`null` value, EvoSuite should invoke the `computeImports` prior to the `addClassImport` method call since the `targetType` initialization happens only in the `computeImport` call.
- The `isEmpty` of `LinkedListMultimap` (`GUAVA-102`) checks whether the linked list is empty by the `head == null` conditional expression. We found EvoSuite applied only empty lists to the `isEmpty` method in all attempts, which caused the negated branch of `head == null` (i.e., `head != null`) remain not covered.

3.4 SQ4: Other Problem

4A File system access (FSA):
The FSA category relates to attempts to access files in the file system. EvoSuite already provides a Virtual File System (VFS) [4] to handle such file accesses

Listing 1.10: Argument state problem in GUAVA-39

```
L1: // To cover: argument newCount > 0
L2: public boolean setCount(@Nullable E element, int oldCount,
L3:   int newCount) { ...
L4:   AvlNode<E> root = rootReference.get();
L5:   if (root == null) {
L6:     if (oldCount == 0) {
L7:       if (newCount > 0) { /* not covered block */ }
L8:       return true;
L9:     } else { return false; } } ... }
```

Listing 1.11: Invoking object state problem in SPOON-169

```
L1: // To cover: invoke computeImports prior to addClassImport
L2: protected boolean addClassImport(CtTypeReference<?> ref) {
L3:   ...
L4:   if (targetType != null && ...) { /* not covered */ } ...
L5:   if (targetType != null && ...) { /* not covered */ } ...
L6:   if (targetType != null) { ... /* not covered */ } ... }
L7:
L8: public void computeImports(CtElement element) { ...
L9:   targetType = ... /* targetType was set to non-null here */
L10: ... }
```

during testing. However, there are still cases where VFS itself is insufficient, for example when the target program expects files with certain extension, file format, or possibly existing OS-related files. Several FSA examples in the benchmark are as follows:

- `FileSystemFontProvider` (`PDFBOX-8`) as shown in Listing 1.12. During its construction, `FileSystemFontProvider` performs a scan (L3) for existing font files in the file system. The test failed to find font files in the file system, causing `files` (L1) and `fonts` (L3) became empty lists. Thus, the `for` loop in L4 and other six branches within the same class became not covered.
- `MavenLauncher` (`SPOON-32`) requires to read a valid Maven's `pom.xml` file, whose path specified in its constructor's argument. If a valid `pom.xml` file does not exist in the file system, the execution will raise an `SpoonException` so the test will not cover certain branches.

4B JVM's `System.getProperty` call (JSC):
JRE allows JVM to store values through `System.setProperty`. However, EvoSuite provides *no* mechanism to update those values. For an example of `FileSystemFontProvider` (`PDFBOX-8`) in Listing 1.13, the program queried the `"pdfbox.fontcache"` (L2) and `"user.home"` (L4) property. But the values of `path` in L3 and L5 were always `null` since EvoSuite did not update those values.

4C Branch unreachable (UBR):
The UBR category captures all branches that are infeasible to cover by any execution paths. We found that 33 not-covered branches belonged to UBR, which corresponded to around 9% of the not-covered branches that we manually analyzed. For an example in the Listing 1.14, `getDeclaration` in `CtLocalVariable-`

Listing 1.12: File system access in PDFBOX-8

```
L1: List<File> files = new ArrayList<File>();
L2: FontFileFinder fontFileFinder = new FontFileFinder();
L3: List<URI> fonts = fontFileFinder.find();
L4: for (URI font : fonts) {
L5:   files.add(new File(font)); // not covered
L6: } ...
```

Listing 1.13: JVM's `System.getProperty` problem in PDFBOX-8

```
L1: private File getDiskCacheFile() {
L2:   String path = System.getProperty("pdfbox.fontcache");
L3:   if (path == null || ...) { // else was not covered
L4:     path = System.getProperty("user.home");
L5:     if (path == null || ...) { // else was not covered
L6:       path = System.getProperty("java.io.tmpdir"); } }
L7:   return new File(path, ".pdfbox.cache"); }
```

`ReferenceImpl` (`SPOON-20`) has a `null`-checking branch (L3) on the return value of `getFactory` (L2). The `getFactory` (L5-L7) never returns a `null` value, causing the *then* branch in L3 not to be covered.

3.5 SQ5: Common Major Problems

Figure 3 shows the distribution of the limitation categories (described in Table 2) of the not-covered branches per subject in the SBST 2020 benchmark. Note that Figure 3 shows data aggregated per class (i.e., multiple branches with the same category within the same class is counted as one) to find a common major problem in SQ 5.

We found that EvoSuite has *no* common major problem across all four target subjects. However, each subject has its own major problem (see the number of the classes of each subject (Table 1) and Figure 3). For example, the *key-value store pattern* (L-KVS) was a major problem in PDFBOX only (i.e., 10 out of 20 classes of PDFBOX suffered L-KVS. But, only one class of GUAVA (and SPOON) suffered L-KVS and FESCAR had no class that suffered L-KVS). Similarly, *construction failure* (C-CF) was a major problem in FESCAR only.

4 Conclusion and Future Work

To improve the effectiveness of EvoSuite by analyzing the EvoSuite's limitations, this paper presents the limitations of EvoSuite through an empirical study on the latest SBST 2020 benchmark. Through the manual analysis of the 359 reached-but-not-covered branches, we classified the four groups of the limitations of EvoSuite (i.e., construction problems, OO-related problems, large search space problems, and so on (Table 2)). We reported all observed limitations of EvoSuite on the SBST 2020 benchmark with concrete examples so that researchers and practitioners can address such limitations more clearly. For future work, we plan

Listing 1.14: Unreachable branch in SPOON-20

```
public CtLocalVariable<T> getDeclaration() {
  final Factory factory = getFactory();
  if (factory == null) { return null; /* not covered */ }
... }
public Factory getFactory() {
  if (this.factory == null) { return DEFAULT_FACTORY; }
  return factory; }
```

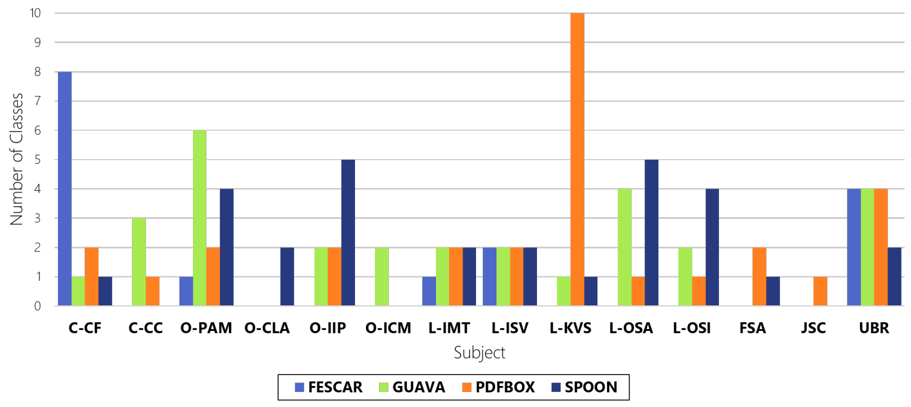

Fig. 3: Limitation category distribution aggregated per class

to apply EvoSuite to more target benchmarks and study the effect of allocating higher search budget and using different fitness functions towards coverage attainment.

Acknowledgments. We would like to thank Ahcheong Lee for his effort to discuss the experiment results. This research work is supported by Basic Research Program (NRF-2020R1C1C1013512, NRF-2020R1C1C1013996, NRF-2021R1A2C2009384) of National Research Foundation (NRF) of South Korea and NRF grant funded by the Korea government (MSIT) (2021R1A5A1021944).

References

1. Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J.: An industrial evaluation of unit test generation: Finding real faults in a financial application. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). pp. 263–272 (2017). https://doi.org/10.1109/ICSE-SEIP.2017.27
2. Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J.: An industrial evaluation of unit test generation: Finding real faults in a financial application. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). pp. 263–272 (2017). https://doi.org/10.1109/ICSE-SEIP.2017.27

3. Arcuri, A.: Restful api automated test case generation. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). pp. 9–20 (2017). https://doi.org/10.1109/QRS.2017.11
4. Arcuri, A., Fraser, G., Galeotti, J.P.: Automated unit test generation for classes with environment dependencies. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 79–90. ASE '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642986, https://doi.org/10.1145/2642937.2642986
5. Devroey, X., Panichella, S., Gambi, A.: Java unit testing tool competition: Eighth round. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. p. 545–548. ICSEW'20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387940.3392265, https://doi.org/10.1145/3387940.3392265
6. Fraser, G., Arcuri, A.: EvoSuite: Automatic test suite generation for object-oriented software. pp. 416–419. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, New York, NY, USA (2011)
7. Fraser, G., Arcuri, A.: 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. Empirical Softw. Engg. **20**(3), 611–639 (Jun 2015). https://doi.org/10.1007/s10664-013-9288-2, https://doi.org/10.1007/s10664-013-9288-2
8. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated unit test generation really help software testers? a controlled empirical study. ACM Trans. Softw. Eng. Methodol. **24**(4) (Sep 2015). https://doi.org/10.1145/2699688, https://doi.org/10.1145/2699688
9. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: 29th International Conference on Software Engineering (ICSE'07). pp. 75–84 (2007). https://doi.org/10.1109/ICSE.2007.37
10. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Transactions on Software Engineering **44**(2), 122–158 (2018). https://doi.org/10.1109/TSE.2017.2663435
11. Panichella, A., Campos, J., Fraser, G.: Evosuite at the sbst 2020 tool competition. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. p. 549–552. ICSEW'20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387940.3392266, https://doi.org/10.1145/3387940.3392266
12. Panichella, A., Panichella, S., Fraser, G., Sawant, A.A., Hellendoorn, V.J.: Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 523–533 (2020). https://doi.org/10.1109/ICSME46990.2020.00056
13. Rojas, J.M., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. Software Testing, Verification and Reliability **26**, n/a–n/a (03 2016). https://doi.org/10.1002/stvr.1601
14. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: 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). pp. 201–211 (2015). https://doi.org/10.1109/ASE.2015.86