SketchFix: A Tool for Automated Program Repair Approach using Lazy Candidate Generation

Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid
The University of Texas at Austin, USA
{lisahua,mengshi.zhang,kaiyuanw,khurshid}@utexas.edu

ABSTRACT

Manually locating and removing bugs in faulty program is often tedious and error-prone. A common automated program repair approach called *generate-and-validate* (G&V) iteratively creates candidate fixes, compiles them, and runs these candidates against the given tests. This approach can be costly due to a large number of re-compilations and re-executions of the program. To tackle this limitation, recent work introduced the SketchFix that tightly integrates the generation and validation phases, and utilizes runtime behaviors to substantially prune a large amount of repair candidates. This tool paper describes our Java implementation of SketchFix, which is an open-source library that we released on Github. Our experimental evaluation using Defects4J benchmark shows that SketchFix can significantly reduce the number of re-compilations and re-executions compared to other approaches and work well in repairing expression manipulation at the AST node-level granularity. The demo video is at: https://youtu.be/AO-YCH8vGzQ.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Program Repair, Program Synthesis, Program Sketching

ACM Reference Format:
Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. SketchFix: A Tool for Automated Program Repair Approach using Lazy Candidate Generation. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3264600

1 INTRODUCTION

Automated program repair (APR) [11, 14, 16] has shown much promise to reduce human effort in debugging. A common repair approach is *generate-and-validate* (G&V) [8, 12, 17, 18], where candidate fixes are iteratively generated and validated against the given tests. However, traditional G&V techniques require many candidates to be re-compiled and re-executed until a candidate that passes all tests is found. The times for re-compilation and re-execution are non-trivial, especially for open source projects. Our recent work introduced SketchFix [5] that enhances the traditional G&V approach. The novelty of SketchFix is that it reduces the compilation and execution overhead by tightly integrating the generation and validation phases and using lazy candidate generation. Intuitively, SketchFix utilizes precise runtime information to create candidates as needed. To illustrate, consider trying to fix a faulty while-loop condition and the body of the loop; if a test execution raises an exception when evaluating the condition candidate of the loop, SketchFix does not consider any candidate in the while-loop body because the body is never executed. Different from traditional G&V approaches that generate thousands of concrete candidates, SketchFix does not create any concrete candidates for the parts of the program that are not reached by the test executions. This lazy candidate generation approach leverages runtime behavior to substantially prune a large part of the search space.

Another common repair approach is based on constraint solving [3, 9, 13], which uses off-the-shelf solvers to synthesize repairs based on the constraints created from faulty programs and tests. Such techniques generally reason about `boolean` or `integer` type [9, 13] and can hardly handle non-primitive-type expressions *in presence of* complex libraries (e.g., Angelix [13] cannot repair subjects from `python` and `lighttpd`). Without translation to SAT, SketchFix explores the actual runtime behavior to synthesize repairs in presence of libraries. Moreover, SketchFix can be applied to projects with unconventional structures, whereas many tools (e.g., Astor [12] and ACS [18]) cannot repair defects from the `Closure` project due to its non-standard test-cases.

This paper describes the Java implementation of SketchFix [5]. It performs fine-grained repairs at the AST node-level. Given a faulty Java program and a test suite as input, SketchFix first uses an existing spectrum-based fault localization technique called Ochiai [1] to rank suspicious statements based on the suspiciousness value. For each suspicious statement, SketchFix introduces "holes" [15] at this location based on AST node-level transformation schemas. SketchFix provides APIs to specify "holes" using Java syntax that can be directly compiled and executed against the test suite. SketchFix employs a sketch engine called EdSketch [4] to fill in the holes with backtracking search. When a test fails due to either a runtime exception or an assertion failure, the parts of the candidate program that were executed determine the generation of the future candidates. SketchFix backtracks when it encounters exceptions or test failures, and selects the next candidate until it finds a repair candidate that satisfies all tests.

SketchFix defines transformation schemas at a fine granularity and prioritizes schemas that introduce smaller perturbations to the original programs. Recent studies [9, 14] propose the insight

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264600

```
// Input: the path for source code and tests
// d4j.dir.src.classes=src
// d4j.dir.src.tests=test
// Human-written patch
public Vector2D intersection(...) {
 Vector2D v2D = line1.intersection(line2);
  // throw NullPointerException if v2D is null
+ if (v2D == null)
+   return null; ...}
// A sketch automatically generated by SketchFix
public Vector2D intersection(...) {
 Vector2D v2D = line1.intersection(line2);
 if (SketchFix.COND(...);
  return (Vector2D) SketchFix.EXP(...); ...}
// Output: Synthesized repair
// SketchFix.COND:  v2D == null
// SketchFix.EXP: v2D
```

Figure 1: The input and output of SketchFix

that patches that are semantically closer to the original programs are more likely to be correct. Our ranking strategy is consistent with this insight, which aims to mitigate the overfitting issue [14]. SketchFix is available at https://github.com/sketchFix.

We evaluated SketchFix using Defects4J [7] benchmark. With the default setting, SketchFix correctly fixes 19 out of 357 bugs in 23 minutes on average. With lazy candidate generation, SketchFix requires only 1.6% of re-compilations (#compiled sketches/#candidates) and 3% of re-executions out of all repair candidates when it finds the first repair. Even if SketchFix exhaustively explores the entire search space, it only compiles 7% of all candidates (#sketches/#candidates), whereas without lazy candidate generation, all 100% of candidates must be compiled.

2 EXAMPLE

We describe SketchFix through a defect from the `Apache Math` project. Figure 1 presents the input and output of SketchFix, a human-written patch, and a sketch generated by SketchFix that leads to a correct fix. Note that the end user is not aware of the intermediate results (sketches). We list the sketch to illustrate the notion of lazy candidate generation.

Given the path of source code and tests, SketchFix uses the Ochiai [1] fault localization approach to identify a list of suspicious statement. For each returned suspicious statement, SketchFix applies AST node-level transformations to generate sketches. For each sketch, SketchFix replaces the original source file with the sketch, compiles it, and executes the sketch against the given tests. This process is *automated* and no other human effort is required.

In Figure 1, SketchFix transforms the faulty program to a sketch based on two AST node-level transformation schemas: a condition schema that introduces a new if-condition and a return schema that inserts a return statement. Different from other repair techniques [8, 17] that generate a list of concrete candidates and compile them iteratively, SketchFix uses a sketch with holes to encode all candidates of the if-return repair template. Thus the sketch is compiled only once yet it represents hundreds of concrete candidates.

SketchFix executes the given test suite against the compiled sketch. When the test execution reaches the hole `SketchFix.COND(...)`, instead of considering hundreds of concrete candidates such as `v2D != null` and `line1 != line2`, SketchFix only considers two boolean values (`true` and `false`) and selects either `true` or `false` to

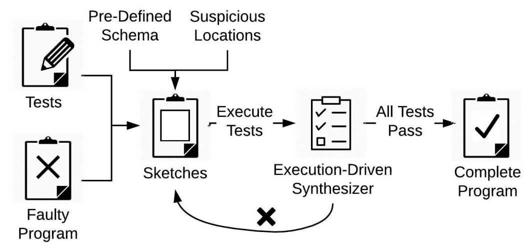

Figure 2: Workflow of SketchFix

fill in the condition hole. If SketchFix selects `false` for the if condition, it does not initialize any candidate in the `return` expression because the test execution does not reach the hole `SketchFix.EXP(...)` inside the if-condition body. In contrast, traditional G&V repair techniques can hardly utilize the runtime information that the condition is evaluated to be false. As a result, they must compile and execute all candidates for the body of the if-condition that are not reached by the test execution. In this example, selecting `false` leads to a test failure. SketchFix backtracks and selects the next candidate, which sets the if-condition to `true`. When the test execution reaches the hole `SketchFix.EXP(...)`, SketchFix lazily generates candidates for this hole and selects a candidate to fill in the expression hole. In this example, SketchFix selects the visible variable `v2D` to fill in the hole and this candidate passes all tests.

3 IMPLEMENTATION

Figure 2 shows the workflow of SketchFix. Given a faulty program and a test suite, SketchFix first identifies a list of suspicious statements sorted by the suspiciousness value based on Ochiai fault localization technique. For each suspicious location, SketchFix applies pre-defined AST node-level transformation schemas (Section 3.1) to create sketches. These sketches are directly compiled and executed against the test suite. Once the execution triggers test failures or runtime exceptions, SketchFix backtracks, selects the next candidate to fill the hole, and executes the new candidate (Section 3.2) against the tests.

3.1 AST Node-Level Transformation

SketchFix performs a systematic reduction of program repair to program synthesis by translating faulty programs to sketches at a fine granularity. The API provided by SketchFix mainly take three parameters: an object list that contains all visible variables and default values (`null` or `0`), a hole id to distinguish different holes for the same type, and the target type of the generated candidates. For instance, the expression hole in Figure 1 is specified as `SketchFix.EXP(new Object[]{v2D,line1,...,null},0,Vector2D.class)`. The target type of the hole is derived from the return type of the method based on Java syntax. Different types of holes can have the same id yet this id must be unique across the same type of holes. For example, the condition hole in Figure 1 is `SketchFix.COND(new Object[]{v2D,line1,...,null}, 0)`, and SketchFix will not use the hole id 0 to specify another condition hole. If the target type of hole is unknown, SketchFix takes the first two parameters and treats the target type as another hole to synthesize.

We use JavaParser [6] to automatically transform the faulty program to sketches. To handle defects that require multiple holes to fix, such as the null pointer checking, SketchFix applies transformation schemas incrementally at the same suspicious location. Due to the large search space of repair candidates, SketchFix creates no more than two schemas at the same location by default.

Transformation Schema. We define six AST node-level transformation schemas that take a suspicious location as input and produce sketches with holes.

Expression Transformer (EXP): If the faulty statement contains any AST node of variables, constant values, or field dereferences, the node is transformed to a hole `SketchFix.EXP(...)`, which returns an object. This object is casted to the corresponding type.

Operator Transformer (AOP): If the faulty line contains a binary expression with arithmetic operator (+, −, ×, /), this binary expression is transformed to a hole `SketchFix.AOP(...)`.

Overloading Transformer (PAR): If the faulty statement contains a method invocation that has an overloading method, SketchFix maps parameter types and generates expression holes to represent parameters in different types.

Condition Transformer (COND): This schema appends a new clause to the faulty condition, e.g., `if (cond &&SketchFix. COND(...))`. The new clause is represented as left and right hand side expressions combined with a relational operator. If the expressions are of non-primitive types, SketchFix applies relational operators "==" and "! =" to construct the clause, while for primitive types, it applies all 6 operators ($==, !=, >, <, \leqslant, \geqslant$). The new clause is appended to the existing condition *cond* with logical operators ("&&" and "||").

If-condition transformer (IF): SketchFix introduces an if-condition before the faulty statement with a condition hole.

Return-statement transformer (RTN): SketchFix inserts a `return` statement before the faulty statement. If the return type of the current method is `void`, SketchFix simply inserts an empty `return` statement, otherwise, SketchFix inserts an expression hole based on the method's return type.

Ranking Strategies. Intuitively, the synthesis cost increases if more holes are introduced to the sketch. We define the cost of transformation schemas as the number of atomic holes (expression holes and operator holes) introduced by the schemas. We prioritize the schemas with lower synthesis cost. For instance, we favor expression (EXP) and operator (AOP) schemas over the condition schema (COND) because the condition schema inserts a relational operator hole and two expression holes at the left and right hand side of the relational operator. This strategy is consistent with the heuristic from the existing literature [9, 14]: repair candidates that semantically closer to the original programs are relatively easier to comprehend by the developers.

With the intuition that variables declared closer to the hole are more likely to be used [9], we rank variables based on their proximity to the hole location, i.e., the number of statements between the hole and the variable declaration. For conditional holes whose target types are unknown, we explore target types based on the types of variables declarations in descending order of their closeness to the hole location. For instance, SketchFix prioritizes the target type `Vector2D` for the condition hole in Figure 1 because the closest defined variable (`v2D`) is of this type.

```
JUnitCore core = new JUnitCore();
Result result = null;
Class target = Class.forName(Defects4J.triggerTest);
do {
 try {
  result = core.run(target);
  if (result.wasSuccessful()) {
   System.out.println("Solution:"+SketchFix.getString());
   break;
 } catch (Exception e) { }
} while (SketchExecutor.incrementCounter()); }}
```

Figure 3: Test driver used by SketchFix

3.2 Lazy Candidate Generation

When the test execution first reaches a hole, SketchFix initializes candidates of the hole based on the given visible variables and default values. Each candidate is assigned a unique identifier, which is its index in the list. Each hole's candidate identifier is initialized as -1, indicating that this hole has not been initialized. When the execution first reaches a hole whose identifier is -1, SketchFix selects an identifier starting from 0 to represent the candidate used to fill in the hole. In Figure 1, the identifier 0 maps to the value `false` for the condition hole. The execution continues with this choice of candidates until the execution encounters a runtime exception or a test failure, leading to a backtrack with an increment of the candidate identifier (Figure 3 `incrementCounter()`), which dynamically selects the next candidate. In Figure 1, the identifier 1 maps to the value `true` for the condition hole. If there exist multiple holes, the method `incrementCounter` will increment a hole identifier at a time.

The process terminates when a repair that passes all tests is found or the space of candidate programs is exhausted, i.e., all candidate identifiers have reaches their maximum values – the sizes of the candidate lists. In this case, the method `incrementCounter` returns `false` and the program exits the while loop. Note that checking the test result (`result.wasSuccessful()`) can be generalized to other frameworks apart from JUnit, e.g. TestNG. Therefore, SketchFix can perform repair for the subjects that do not use JUnit tests.

4 EXPERIMENTS

We evaluate SketchFix on the Defects4J benchmark [7], which consists of 357 defects from 5 open source Java projects.

To identify suspicious statements for the defects, we use the ASM bytecode analysis framework [2] to capture the coverage of failing and passing test executions. SketchFix uses an existing spectrum-based fault localization technique called Ochiai [1] to rank suspicious statements. Existing empirical study [19] illustrates that Ochiai is effective on localizing defects in object-oriented programs. It has been applied to all four repair techniques [3, 10, 12, 18] that we use in the comparison. If multiple statements have the same suspiciousness score, we order them randomly.

We first compare SketchFix's repair efficacy with other repair techniques –Astor [12], Nopol [3], ACS [18] and HDRepair [10] that have been evaluated against the Defects4J benchmark. Due to the space limit, Figure 4 only presents part of the repair result through manual inspection that contains the defects fixed by SketchFix. A full comparison can be found at [5]. We check three conditions to identify if the repair is semantically equivalent to the

No.	SF	A	N	C	H	No.	SF	A	N	C	H
CH1	✓	?	×	×	✓	L6	✓	×	×	×	✓
CH8	✓	×	×	×	✓	L51	?	×	?	×	✓
CH9	✓	×	×	×	×	L55	✓	?	✓	×	×
CH11	✓	×	×	×	×	L59	✓	×	×	×	✓
CH13	?	?	?	×	×	T4	?	?	×	×	×
CH20	✓	×	×	×	×	M5	✓	✓	×	✓	✓
CH24	✓	×	×	×	×	M33	✓	×	?	×	×
CH26	?	?	?	×	×	M50	✓	✓	✓	×	✓
C14	✓	×	×	×	✓	M59	✓	×	×	×	×
C62	✓	×	×	×	✓	M70	✓	✓	×	×	✓
C70	?	×	×	×	✓	M73	?	✓	?	×	×
C73	?	×	×	×	✓	M82	✓	?	?	✓	✓
C126	✓	×	×	×	✓	M85	✓	?	?	×	×

Figure 4: Manual Assessment Result of Patches Generated by SketchFix and Other Repair Approaches. SF represents SketchFix, A represents Astor [12], N represents Nopol [3], C represents ACS [18], and H represents HDRepair [10]. ✓ represents correct fix, ? represents plausible fix, and × represents not generating fix.

human-written patch: 1) the repair is at the same location; 2) the repair is with the same type of repair, i.e., expression or operator manipulation; 3) the runtime value of the candidate for the hole is the same as the value of the expression developer used to fix the defect. For example, in Figure 1, we treat `return v2D` as semantically equivalent to `return null` in the null pointer checking. SketchFix generates 19 correct repairs and 7 plausible ones, i.e., repairs that pass all tests but fail in manual inspection. This result compares well with other four repair techniques.

Compared to other repair techniques, SketchFix works particularly well in manipulating expressions and variable types. For instance, Figure 5 presents a human-written patch that uses different parameters with different types (`integer` vs. `double`). SketchFix correctly fixes this bug with the overloading transformation schema. In contrast, the constraint-solving-based repair techniques [3, 13] in general only modify expressions in conditions or the right hand side of assignments with `boolean` or `integer` types. These techniques can hardly fix defects that require manipulations of expression and variable types. Shown as Figure 5, the constraint-based tool Nopol generates a plausible repair by inserting a new if-statement. This example also illustrates that compared to Nopol, SketchFix introduces smaller AST node-level change to the original program and this repair is more likely to be accepted by the user.

With lazy candidate generation, every sketch will be compiled once which may represent thousands of candidates. When SketchFix finds the first repair, it compiles 1.6% (avg. #compiled sketches/#space). Even if SketchFix exhaustively searches the entire space of repair candidates, it only compiles 7% (avg. #sketches/#space) of all candidates, which must all be compiled without lazy candidate generation. The experiment shows that SketchFix only executes 3% of candidates (avg. #Gen/#Space) when it finds the first patch that passes all tests. On average, SketchFix spends 9 minutes to locating faults and generating sketches, and 23 minutes to generating the first repairs that satisfy all test assertions. The performance of our tool compares well with other repair techniques.

```
 /* Human-Written patch for SimplexTableau.java  */
private final int maxUlps;
private final double epsilon;
protected void dropPhase1Objective() {
- if (Precision.compareTo(entry,0d,maxUlps)>0){...
+ if (Precision.compareTo(entry,0d,epsilon)>0){... }
 /* SimplexTableau.java as sketch */
protected void dropPhase1Objective() {
 if (Precision.compareTo(entry,0d,(Double)SketchFix.EXP(
  Double.class,new Object[]{..,epsilon,maxUlps,..}))>0)...
 // Synthesized solution: SketchFix.EXP: epsilon
// A plausible repair generated by Nopol
protected void dropPhase1Objective() {
if (Precision.compareTo(entry, 0d, maxUlps) > 0) {
+ if (numSlackVariables<constraints.size()) {...}}
```

Figure 5: Patches Generated by SketchFix and Nopol

5 CONCLUSION

This paper described SketchFix, an automated program repair tool with lazy candidate generation. The key insight of SketchFix is to utilize runtime information to substantially prune the space of candidates. It transforms the faulty program to sketches with holes at the fine-grained granularity. SketchFix can be applied to a wide class of programs with various code structures in presence of libraries. With a tight integration of the generation and validation phase, SketchFix substantially pruned a large amount of candidate fixes based on our experiments. Our result also indicates that SketchFix works well in repairing defects with non-primitive-type expression manipulation at the AST node-level.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1319688, CCF-1704790, and CCF-1718903.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. [n. d.]. On the accuracy of spectrum-based fault localization. In *TAIC PART*.

[2] ASM Java bytecode manipulation framework 2017. http://asm.ow2.org/

[3] Favio Demarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA*.

[4] Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: execution-driven sketching for Java. In *SPIN*.

[5] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards Practical Program Repair with On-Demand Candidate Generation. In *ICSE*.

[6] JavaParser 2017. http://javaparser.org. Accessed: 2017-07-30.

[7] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*.

[8] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*.

[9] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *FSE*.

[10] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER*.

[11] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *FSE*.

[12] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *ISSTA*.

[13] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE*.

[14] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *FSE*.

[15] Armando Solar-Lezama. 2013. Program sketching. *STTT* (2013).

[16] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. Automated Model Repair for Alloy. In *ASE*.

[17] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*.

[18] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE*.

[19] Jifeng Xuan and Martin Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *ICSME*.