Towards Practical Program Repair with On-Demand 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

Effective program repair techniques, which modify faulty programs to fix them with respect to given test suites, can substantially reduce the cost of manual debugging. A common repair approach is to iteratively first generate candidate programs with possible bug fixes and then validate them against the given tests until a candidate that passes all the tests is found. While this approach is conceptually simple, due to the potentially high number of candidates that need to first be generated and then be compiled and tested, existing repair techniques that embody this approach have relatively low effectiveness, especially for faults at a fine granularity.

To tackle this limitation, we introduce a novel repair technique, SketchFix, which generates candidate fixes *on demand* (as needed) during the test execution. Instead of iteratively re-compiling and re-executing each actual candidate program, SketchFix translates faulty programs to *sketches*, i.e., partial programs with "holes", and compiles each sketch once which may represent thousands of concrete candidates. With the insight that the space of candidates can be reduced substantially by utilizing the runtime behaviors of the tests, SketchFix lazily initializes the candidates of the sketches *while* validating them against the test execution.

We experimentally evaluate SketchFix on the Defects4J benchmark and the experimental results show that SketchFix works particularly well in repairing bugs with expression manipulation at the AST node-level granularity compared to other program repair techniques. Specifically, SketchFix correctly fixes 19 out of 357 defects in 23 minutes on average using the default setting. In addition, SketchFix finds the first repair with 1.6% of re-compilations (#compiled sketches/#candidates) and 3.0% of re-executions out of all repair candidates.

ACM Reference Format:
Jinru Hua, Mengshi Zhang, Kaiyuan Wang and Sarfraz Khurshid. 2018. Towards Practical Program Repair with On-Demand Candidate Generation. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180245

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180245

1 INTRODUCTION

Manually debugging faulty code is often tedious and costly, and itself error-prone. The last decade has seen much progress in the area of program repair, which has shown much promise for automating debugging to reduce its cost and increase its effectiveness [2, 11, 19, 23, 29, 32, 45, 46, 54]. A common approach for automated program repair is *generate-and-validate* [22, 25, 30, 55, 56, 58], where several candidate programs that represent potential bug fixes are iteratively generated using repair templates, and validated against the given tests until a candidate that passes all tests is found. While techniques that embody this approach have shown their effectiveness on repairing a number of defects using various search algorithms [21, 43, 55, 56], a limiting characteristic of these techniques, especially for repairing faults at a fine granularity, is that they require potentially many candidates to first be generated and then be compiled and tested. The costs of compilation and test execution are non-trivial [26, 31, 55], especially for open source projects like the Closure compiler [4], which takes around 1 minute for the compilation and the test execution. Given a typical search space of repair candidates (around 5k), the total re-compilation and re-execution time can be as long as 3 days.

To allow the exploration of large numbers of candidates, researchers have developed various techniques in previous work. For example, some techniques [7, 24, 36, 37, 40] infer constraints and synthesize repairs by translating the constraints to propositional satisfiability (SAT) formulas. Such translation-based synthesis may involve incomplete translations or create impractical problems that require creating complex models for all involved libraries. Moreover, they generally exclusively reason about `boolean` or `integer` type [24, 37] and can hardly handle manipulation of non-primitive-type expressions *in presence of* libraries or complex constructs like AST node-level type casting. Some techniques mine historical data [25, 28, 30] or analyze documents [27, 58] to rank the repair candidates. These techniques have shown their effectiveness on some classes of defects like exception handling, yet they may not be effective at repairs that require fine-grained expression manipulations at the AST node-level.

We present SketchFix, which is a novel technique for more effective generate-and-validate program repair using a perspective different from previous work. Our key insight is that the space of candidate programs can be pruned substantially by utilizing runtime information and by generating candidates *on-demand* during test validation. To illustrate, consider trying to fix a faulty condition in a while-loop as well as the body of the loop; if a test execution raises an exception upon evaluating a specific candidate while-loop condition, all candidates of the while-loop body are pruned from search for that choice of the candidate condition expression. In fact, our approach for *lazy candidate generation* will not create any

candidates for the while-loop body (which may contain thousands of patches) if the while-loop body is not executed. When a test fails due to either a runtime exception or a test assertion failure, the *parts* of the candidate program that were directly executed determine the generation of the future candidates. Instead of the traditional approach of iteratively generating and validating each repair candidate, we tightly integrate the generation and validation of candidates by effectively utilizing runtime behaviors of the test executions to prune a large part of the search space, which must be explored otherwise.

At the AST node-level, SketchFix performs a systematic reduction of program repair to program synthesis [10, 18, 41, 51] by translating a faulty Java program to sketches [16, 50], which will be completed by a synthesizer [13] with respect to the given test suite. Given a faulty Java program and a test suite as input, SketchFix introduces holes to suspicious statements based on the AST node-level transformation schemas. To fill in these holes, SketchFix employs a practical sketch engine called EdSketch [13] to synthesize sketches with backtracking search. Whenever EdSketch encounters runtime exceptions or test failures, it backtracks immediately and fetches for the next choice until the space of candidates is exhausted or a complete program that satisfies all tests is found. This complete program is regarded as a repair for the original faulty program.

SketchFix defines transformation schemas at a fine granularity and prioritizes first the schemas that introduce smaller perturbations to the original program. Recent techniques present the insight that patches that are semantically closer to the original programs are more likely to be correct from the perspective of the developers [5, 24]. Our ranking strategy is in line with this insight and aims to mitigate the *overfitting* issue [49] in automated program repair.

We evaluate SketchFix using Defects4J [20]—a dataset that has been widely used to evaluate automated program repair techniques. With default setting, SketchFix correctly fixes 19 out of 357 bugs in 23 minutes on average. It additionally fixes 15 faults with alternative settings, such as applying more transformation schemas, etc. Furthermore, we compare SketchFix with other repair techniques and illustrate favored defect classes for different automated program repair approaches. With on-demand 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. Lastly, we demonstrate that our find-grained transformation schemas are able to generate high-quality patches by introducing AST node-level edits to the original programs.

In summary, we make the following contributions:

- **On-Demand Candidate Generation for Program Repair.** Utilizing runtime behaviors, we lazily generate *on-demand* candidates during the test execution for more practical program repair. This integration of the generation and the validation phases substantially prunes the search space of the repair candidates.
- **AST node-level Program Repair** We design a set of AST node-level transformation schemas to repair faulty programs at a fine granularity. This strategy is effective for generating high-quality patches that are semantically closer to the original programs.
- **Practical Reduction of Program Repair to Synthesis.** Without inferring constraints or creation of constraint solving problems for SAT/SMT solvers, we transform the faulty subjects to sketches and synthesize code to complete the sketches using a backtracking search-based sketch system. Our evaluation shows that this reduction from program repair to program synthesis helps build effective and well-founded repair techniques.

2 MOTIVATING EXAMPLE

(A) Part of the human-written patch to fix the Chart14 defect

```
1.public class CategoryPlot extends Plot...{...
2. public boolean removeDomainMarker (...,boolean notify) {
3.  ArrayList markers;
4.  if (...) {...} else {
5.+ if (markers == null)
6.+  return false;
7. ...} }
```

(B) A sketch generated by SketchFix and synthesized solutions

```
1.public class CategoryPlot extends Plot...{...
2. public boolean removeDomainMarker (...,boolean notify) {
3.  ArrayList markers;
4.  if (...) {...} else {
5.  if(SketchFix.COND(ArrayList.class,new Object[]{markers,..}))
6.   return (Boolean) SketchFix.EXP(Boolean.class,
      new Object[]{markers,..};
7.  ...} }
  // Synthesized solution:
  // SketchFix.COND: markers==null,...
  // SketchFix.EXP: false,...
```

Figure 1: An Illustrative Example for a Fault and the Repair Generated by SketchFix

To illustrate the large search space of repair candidates, we present a part of a defect derived from the JFreeChart project [17]. Figure 1 (A) presents a human-written patch to fix this part of the bug that omits the null pointer checking for an ArrayList object (markers). The class CategoryPlot contains 49 fields and 5 local variables (54 candidates in total). Given a suspicious location, if an automated repair tool wants to insert an if-condition and a return statement to fix the bug, without considering field dereferences derived from the variables and visible fields inherited from parent classes, the space of candidates for the if-condition alone can be more than 5k. We define conditions as left and right hand side expressions combined with a relational operator (either "==" or "!=") for non-primitive types, thus the space is $54^2 \times 2$. The return expression has another 15 candidates with the `boolean` type including the default boolean values (`true` and `false`). Given an average compilation and test-execution time of 15 seconds for the JFreeChart project, it takes more than 15 days to validate all 87k candidates.

To effectively explore this large search space of repair candidates, SketchFix translates the faulty program to sketches with holes and synthesizes sketches with on-demand candidate generation. Given

a suspicious location, SketchFix applies AST node-level transformation schemas to generate a set of sketches, and each sketch can represent thousands of concrete repair candidates. Figure 1 (B) illustrates a sketch generated by SketchFix. To generate this sketch, SketchFix applies two schemas at the suspicious location, one schema introduces an if-condition (*if-condition schema*) and another introduces a return statement (*return-statement schema*). To represent an unknown condition hole, i.e, if (??), SketchFix defines an API SketchFix.COND(...) in Java syntax, which returns a non-deterministic boolean value (either true or false) during the test execution. This API takes two parameters: the hole's target type (i.e, ArrayList), and an array of visible variables as candidates to fill in the hole. The hole's target type is defined as the type of the variables used to fill in the hole. SketchFix enumerates all types derived from visible variables and generates one sketch for each target type. At line 6, the *return-statement schema* inserts a return statement with a non-deterministic expression SketchFix.EXP(...), i.e, return ??, whose target type is the return type of the method.

The sketch shown in Figure 1 (B) will be compiled only once, yet it represents 87k candidate patches. SketchFix directly executes the given test suite after the compilation. When the test execution first reaches the hole SketchFix.COND(...), instead of considering thousands of concrete candidates like "markers!=null", SketchFix only considers two `boolean` values and non-deterministically selects either true or false to fill in the condition hole. If SketchFix selects false for the if condition, it will not initialize any candidates for the return expression because the test execution does not reach the hole SketchFix.EXP(...) inside the if block. In this example, choosing the value false for the if condition leads to a test failure at runtime. SketchFix backtracks immediately and selects the next choice which is the value true for the if-condition. SketchFix generates candidates for the expression SketchFix.EXP(...) when the test execution reaches the hole at the first time. Given 15 candidates of the expression hole, SketchFix selects one candidate at a time during the test execution until a candidate that satisfies all tests is found. In this example, SketchFix finds the first solution in 40 seconds after compiling the sketch once and executing the tests twice.

3 APPROACH

In this section, we describe how we translate faulty programs to sketches using AST node-level transformation schemas (Section 3.1). These sketches are further synthesized by a practical sketch engine with on-demand candidate generation (Section 3.2).

3.1 AST Node-Level Transformation

We perform a systematic reduction of program repair to program synthesis by translating faulty programs to sketches at a fine granularity.

Syntax of Partial Expressions (Holes). Figure 2 denotes the syntax of holes. We define two basic types of non-deterministic holes for sketches: expression holes and operator holes. The atomic expression holes (`SketchFix.EXP()`) represent visible variables, constant values and field dereferences. As to the operator holes, we define arithmetic operators $\{+, -, \times, /, \%\}$ (`SketchFix.AOP()`), relational operators $\{==, !=, >, <, \leqslant, \geqslant\}$ (`Sketch Fix.ROP()`) and logical operators $\{||, \&\&\}$ (`SketchFix.BOP()`) that combine multiple clauses. SketchFix generates composite expressions by combining expression holes with operator holes or combining array variables with the index of array elements. Composite holes can further combine together. For instance, we define a hole for conditions (`SketchFix.COND()`) as two expression holes at left and right hand side combined with a relational operator. Both sides of expression holes in the condition can be replaced by infix expressions (e.g., `a+b`) with arithmetic operators, and the condition holes can further be combined together with logical operators to support multiple clauses. To specify these holes in Java syntax, SketchFix provides a list of method invocations and these invocations take two parameters: the target type of the hole and a list of objects.

atomic expr	e :=	$var \mid const \mid var.f$
constant	$const$:=	$null \mid true \mid false \mid k$
arithmetic op	aop :=	$+ \mid - \mid \times \mid / \mid \%$
relational op	rop :=	$== \mid != \mid > \mid < \mid \leqslant \mid \geqslant$
logical op	lop :=	$\&\& \mid \mid\mid$
composite expr	e :=	$e_1\ op\ e_2$ or $array[e_{int}]$

Figure 2: Syntax of Partial Expressions

$$M_{exp} = \frac{p[\ell] \vdash e_t}{e_t \mapsto \omega_t}$$

$$M_{op} = \frac{p[\ell] \vdash op}{op \mapsto \delta}$$

$$M_{par} = \frac{p[\ell] \vdash f(par),\ f(par) \vdash f'(par \cup e_t)}{f(par) \mapsto f'(par \cup \omega_t)}$$

$$M_{con} = \frac{p[\ell] \vdash \texttt{if}\ (c)}{c \mapsto c\ \texttt{lop}\ (\omega_t\ \texttt{rop}\ \omega'_t)}$$

$$M_{if} = \frac{p(\ell) \vdash (v, t)}{p(\ell) \mapsto \texttt{if}\ (\omega_t\ \texttt{rop}\ \omega'_t)\ p(\ell)}$$

$$M_{rtn} = \frac{p(\ell) \vdash (v, t)}{p(\ell) \mapsto \texttt{return}\ \omega_t\ p(\ell)}$$

Figure 3: Program Transformation Schemas

Base on the syntax of the holes, we define six AST node-level transformation schemas (Figure 3) that take a faulty program p as the input and produce sketches with holes.

Expression Transformation M_{exp}: Given a suspicious statement ℓ in the faulty program p, if it contains any variables, constant values, or field dereferences with the type t, M_{exp} transforms this expression e_t to a partial expression ω_t that represents any expression candidates with the same type t. That is, the original expression will be replaced by a hole `SketchFix.EXP(t, new Object[]{v1,v2,...})` while the object list contains all visible variables, parameters and fields.

Operator Transformation M_{op}: Given a suspicious statement, if it contains any arithmetic, relational or logical operators defined in the syntax of holes (Figure 2), M_{op} transforms the operator to an operator hole δ that denotes any operators of the same operator type. For instance, given a clause $a > b$ in the faulty program, M_{op} will replace this clause with a partial expression a ?? b where

Algorithm 1: Static Transformation for Sketch Generation

```
   Input  : Faulty program p, Fault locations L, Schemas M
   Output : List of sketches Q
1  Function transformSketch (p, L, M) is
2      Q ← ∅;
3      foreach ℓ ∈ L do
           /* apply one schema                                  */
4          foreach σ ∈ M do
5              Q ← Q ∪ σ(p, ℓ) ;
           /* apply two schemas                                 */
6          i ← 0;
7          while i < M.size do
8              ω ← M[i](p, ℓ) ;
9              j ← i ;
10             while j < M.size do
11                 Q ← Q ∪ M[j](ω, ℓ), j + + ;
12             i + +;
```

the operator hole ?? can be any of {==, ! =, >, <, ⩽, ⩾}. The corresponding generated sketch is `SketchFix.ROP(Integer.class, new Object[]{a,b})` if `a` and `b` are of `Integer` type.

Overloading Transformation M_{par}: Given a suspicious statement, if it contains a method invocation f that has a overloading method f', SketchFix tries to map parameter types for f and f' using an approach similar to the edit distance [9]. SketchFix only considers overloading methods f' whose edit distances in terms of the parameter types are no more than two. SketchFix generates holes ω_t for f' to represent parameters in different types.

Condition Transformation M_{con}: Given a suspicious condition expression c, M_{con} introduces a new clause for the condition. The new clause is represented as ω_t `rop` ω'_t where ω_t and ω'_t represent any expression candidates with the type `t`. The hole `rop` represents relational operators. If `t` is a non-primitive type, SketchFix applies relational operators "==" and "! =" to construct the clause, whereas for primitive types, it applies all 6 relational operators. The new clause is appended to the existing boolean expression c with logical operators ("&&" and "||").

If-condition transformation M_{if}: Given a suspicious location ℓ, SketchFix introduces an if-condition before ℓ with a condition "hole". If SketchFix selects the target type `t`, the schema will insert an if-condition as `if (SketchFix.COND (t, new Object[]{v1, v2,...})`, where the target type `t` is derived from visible variables v at the location ℓ.

Return-statement transformation M_{rtn}: Given a suspicious location ℓ, SketchFix inserts a `return` statement before ℓ. If the return type of the current method is `void`, SketchFix simply inserts an empty `return` statement, otherwise, SketchFix inserts a hole ω_t based on the method's return type, i.e., `return (t) SketchFix.EXP(t, new Object[]{v1, v2,...})`.

To handle defects that require multiple holes to fix, such as the omission of null-pointer-checking in Figure 1, we apply transformation schemas incrementally at the suspicious statements. Algorithm 1 shows the static transformation approach that applies no more than two schemas on the faulty program p. The loop at line 4 applies each applicable schema on the suspicious location ℓ to create sketches. The nested loop starting from line 7 enumerates all combinations of two schemas and applies these two schemas at the same location ℓ. Given the large search space of repair candidates in open source projects, by default SketchFix applies no more than two transformation schemas at the same location. We leave the discussion of applying more schemas in Section 4.2.2.

Intuitively, the synthesis cost increases with more holes in 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* (M_{expr}) and *operator* (M_{op}) manipulation over the *condition* transformation schema (M_{con}) because the M_{con} inserts a relational operator hole and two expression holes at the left and right hand side of the operator. This strategy is in line with existing literatures [5, 24] based on the heuristic that repair candidates semantically closer to the original programs are relatively easier to comprehend and are more likely to be accepted by the developers.

Existing automated repair techniques have utilized various supportive resources to improve the repair efficacy, such as historical data [25, 30], documents [27, 58], anti-patterns [53] and test generation [57]. Yet we preserve SketchFix's practicality and leave the improvement of fix patterns [28, 30, 58] as future work.

3.2 Practical Sketch Synthesis

After transforming the original faulty program to sketches based on the schemas, SketchFix executes test cases to synthesize sketches with on-demand candidate generation. SketchFix will not generate concrete candidates for a hole until the test execution reaches the hole. The candidates are created based on the runtime information. For instance, we will not generate field dereferences for *null* variables (refer to [13]). Algorithm 2 describes the procedure of lazy candidate generation and the sketching using backtracking search.

On-Demand Candidate Generation. When the test execution first reaches a hole (line 2), SketchFix initializes the hole's expression candidates based on the given visible variable list. For the expression hole `SketchFix.EXP(...)`, the method `candidateGen()` at line 3 collects visible variables, constant values and field dereferences (refer to EdSketch [13]), and returns a candidate vector (`hole.candidates`). Each candidate is assigned a unique identifier, which is its index in the vector. Each hole's candidate identifier is initialized as -1, indicating that SketchFix has not selected a candidate for this hole (Algorithm 2 line 4). During the test execution, the sketch engine non-deterministically selects an identifier using `choose()` operator (line 5), and the candidate with the corresponding identifier is used to fill in the hole. The execution continues with this selection until it encounters a runtime exception or a test failure, leading to a backtrack to the next choice of candidates, i.e., incrementing the candidate identifier and selecting the next candidate at runtime.

Synthesis with Backtracking Search. Algorithm 2 illustrates the idea of synthesizing sketches with backtracking search, which is embodied in the sketch engine EdSketch [13]. Instead of inferring constraints and translating them to SAT for the repair synthesis, EdSketch starts sketching by directly executing the given tests. Whenever a runtime exception or a test failure occurs, EdSketch backtracks by throwing a `BacktrackException`, which will enforce

Algorithm 2: On-Demand Candidate Generation based on EdSketch [13]

```
   Input  : Sketches P, test suite T
   Output : Complete Program P' that pass all test cases
1  Function synthesizeHole (hole) is
2      if hole.candidates==null then
           /* First Access                                   */
3          hole.candidates ← candidateGen(hole);
4      if hole.id == -1 then
           /* First Access                                   */
5          hole.id ← choose(0, hole.candidates.size-1) ;
6      return hole.candidates[hole.id] ;
7  Function sketch () is
8      do
9          try
10             exploreCurrentChoice();
11         catch BacktrackException
12             createNextChoice() ;
13     while incrementCounter();
14 Function exploreCurrentChoice() is
15     try
16         foreach test ∈ T do
17             test.run() ;
18     catch TestFailureException
19         throw BacktrackException;
20     printSolution() ;
21     searchExit(); /* if only needs the first solution     */
```

the program to re-execute from the beginning (line 19). For each re-execution, the sketch engine increments the counter for the non-deterministic `choose()` operator and selects the next candidate (line 12) to fill the hole. The synthesizing process terminates when the space of candidate programs is exhausted or a complete program that satisfies all tests is found. EdSketch embeds a range of pruning strategies to effectively explore the space of candidates. For instance, based on the program symmetry, the condition clause `a==b` is equivalent to `b==a`, thus only one clause is considered.

Ranking Candidates Based on The Locality. With the notion that variables declared closer to the hole are more likely to be used [24, 58], we rank variables based on their proximity to the hole location, i.e., the number of statements between the hole and the variable declaration. We rank the target types of the hole with a similar heuristic based on the number of statements between the hole and variable declarations of this type in ascending order. For instance, in our motivating example (Figure 1(B)), the condition hole at line 5 favors the type `ArrayList` because the closest variable is with this type.

In summary, we discuss how we systematically reduce the problem of program repair to program synthesis at the fine granularity. We utilize the runtime behaviors of tests to prune a large amount of search space with on-demand candidate generation.

4 EVALUATION

We evaluate SketchFix on the Defects4J benchmark [20], which consists of 357 real defects from 5 open source Java applications. Defects4J contains various types of defects from open source projects to eliminate the bias of the dataset [39] in terms of the defects types and the abundance of these defect classes [34, 39]. In Section 4.2, we compare SketchFix's repair efficacy and favored defect types with other automated repair techniques. In Section 4.3, we investigate if the on-demand candidate generation could effectively reduce the search space of repair candidates. We finally discuss how our AST node-level transformation influences the patch quality in Section 4.4.

Table 1: Overview of the Evaluation Dataset and the Repair Results of SketchFix

Project	LoC	Tests	#Bug	#Fix	FL Time(m)	Sketch Time(s)	Repair Time(m)
Chart	96k	2.2k	26	6/2	1.1	9.9	44.2
Closure	90k	7.8k	133	3/2	7.2	10.8	3.6
Lang	22k	2.2k	65	3/1	0.4	1.5	26.7
Math	85k	3.6k	106	7/1	1.8	3.7	4.0
Time	28k	4.1k	27	0/1	0.4	14.2	27.3
Total			357	19/7			

#Fix shows the number of correct fixes and plausible fixes (repairs that pass tests but fail in manual inspection).

We address the following research questions in this section:

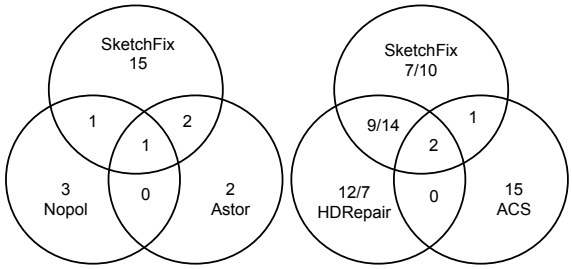
- What's the repair efficacy of SketchFix compared to other repair techniques?
- Does the on-demand candidate generation technique reduce the search space of repair candidates?
- How does our fine-grained transformation affect the quality of the generated repairs?

4.1 Experiment Setting

Defects4J is a collection of reproducible bugs with triggering tests verified. SketchFix regards the test suite of each subject as the correctness property and tries to synthesize a new program such that all tests could pass.

To identify suspicious statements for the defects, we use the ASM bytecode analysis framework [3] together with JavaAgent [14] to capture the test coverage of both failing and passing test executions. SketchFix uses an existing spectrum-based fault localization technique called Ochiai [1] to rank potential faulty statements based on their suspiciousness. Existing empirical studies [52, 59] illustrate that Ochiai is more effective on localizing the faults in object-oriented programs than other techniques. Ochiai has been applied to numerous repair techniques [7, 25, 35, 37, 58], including all four repair techniques [7, 25, 35, 58] we use in the comparison. We rank the suspicious statements based on the Ochiai [1] suspiciousness value for each statement covered by failing executions and select the top 50 suspicious statements by default. If multiple statements have the same suspiciousness score, we order them randomly. We leave the discussion for the number of selected statements in Section 4.2.2.

Table 1 presents the basic information of the subjects used in the evaluation, including the lines of code (*L*oC), the number of test cases (*T*est), the number of defects collected in the benchmark



HDRepair [25] assumes that the faulty methods are known in advance. SketchFix fixes 8 more defects with this assumption, 5 of them are also fixed by HDRepair. We report two results without and with the assumption.

Figure 4: Comparison of Correct Patches Generated by SketchFix and Other Repair Techniques

(*#Bug*), and the average performance time to identify faulty statements (*FL Time*). To translate faulty programs to sketches, we implement transformation schemas using JavaParser [15], a tool that parses and transforms Java programs. The column *Sketch Time* denotes the average performance time to generate program sketches. SketchFix explores the search space of repair candidates for each program sketch until the space of candidates is exhausted or we find a pre-defined number of repairs that pass all tests. Currently we terminate after finding the first repair yet we set the number of output repairs as configurable. We report the average performance time SketchFix takes to find the first repair (column *Repair Time*) following the spirit of existing literatures [29, 37]. We execute only the tests that reach the holes following the spirit of recent work on test pruning [38]. We manually check all synthesized repairs against human-written patches from the version history, and report the number of correct repairs and plausible ones that pass all tests but fail in manual inspection in column *#Fix*.

All performance experiments are conducted on a platform with 4-core Intel Core i7-6700 CPU (3.40 GHz) and 16 Gigabyte RAM on Ubuntu Linux 16.04.

4.2 Repair Efficacy

In this section, we answer the first research question by comparing SketchFix's repair efficacy with other repair techniques—Astor [35], Nopol [7], ACS [58] and HDRepair [25]. We select these four publicly available repair techniques that have been evaluated against the Defects4J benchmark and ignore others which either focus on another language [29, 40, 56] or are not publicly available [22, 24]. All four techniques use Ochiai [1] to identify the suspicious statements. Note that HDRepair assumes that the faulty methods are known in advance and performs Ochiai to identify the faulty statements in the given methods. We thus add this assumption when conducting the comparison with HDRepair. Due to the difference in experiment platforms, we can hardly reproduce all experiments for these four techniques, we thus mainly refer to the generated repairs provided by the authors and refer to their results of the manual inspection.

Table 2: Manual Assessment Result of Patches Generated by SketchFix and Other Repair Approaches

No.	SF	A	N	C	H	No.	SF	A	N	C	H
CH1	✓	?	×	×	✓	M5	✓	✓	×	✓	✓
CH3	×	?	?	×	×	M8	×	?	×	×	×
CH5	×	?	×	×	×	M22	×	×	×	×	✓
CH7	×	?	×	×	×	M25	×	×	×	✓	×
CH8	✓	×	×	×	✓	M28	×	?	×	×	×
CH9	✓	×	×	×	×	M32	×	?	?	×	×
CH11	✓	×	×	×	×	M33	✓	×	?	×	×
CH13	?	?	?	×	×	M34	×	×	×	×	✓
CH14	×	×	×	✓	×	M35	×	×	×	✓	×
CH15	×	?	×	✓	×	M40	×	?	?	×	×
CH19	×	×	×	×	×	M42	×	×	?	×	×
CH20	✓	×	×	×	×	M49	×	?	?	×	×
CH21	×	×	?	×	×	M50	✓	✓	✓	×	✓
CH24	✓	×	×	×	×	M53	×	✓	×	×	✓
CH25	×	?	?	×	×	M57	×	×	?	×	×
CH26	?*	?	?	×	×	M58	×	×	?	×	×
C1	×*	×	×	×	×	M59	✓	×	×	×	×
C10	×	×	×	×	✓	M61	×	×	×	✓	×
C11	×*	×	×	×	×	M69	×	×	?	×	×
C14	✓	×	×	×	✓	M70	✓	✓	×	×	✓
C51	×	×	×	×	✓	M71	×	?	?	×	×
C62	✓	×	×	×	✓	M73	?	✓	?	×	×
C70	?*	×	×	×	✓	M78	×	?	?	×	×
C73	?*	×	×	×	✓	M80	×	?	?	×	×
C126	✓	×	×	×	✓	M81	×	?	?	×	×
L6	✓	×	×	×	✓	M82	✓	?	?	✓	✓
L7	×	×	?	✓	×	M84	×	×	×	×	×
L10	×*	×	×	×	✓	M85	✓	?	?	✓	×
L24	×	×	×	✓	×	M87	×	×	?	×	×
L35	×	×	×	✓	×	M88	×	×	?	×	×
L39	×	×	×	×	×	M89	×	×	×	✓	×
L43	×	×	×	×	✓	M90	×	×	×	✓	×
L44	×	×	✓	×	×	M93	×	×	×	✓	×
L46	×	×	?	×	×	M95	×	?	×	×	×
L51	?*	×	?	×	✓	M97	×	×	?	×	×
L53	×	×	?	×	×	M99	×	×	×	✓	×
L55	✓	?	✓	×	×	M104	×	×	?	×	×
L57	×	×	×	×	✓	M105	×	×	?	×	×
L58	×	×	✓	×	×	T4	?	?	×	×	×
L59	✓	×	×	×	✓	T11	×	?	?	×	×
M2	×	?	×	×	×	T15	×	×	×	✓	×
M3	×	×	×	✓	×	T19	×*	×	×	×	✓
M4	×	×	×	✓	×						

SF represents SketchFix, A represents Astor [35], N represents Nopol [7], C represents ACS [58], and H represents HDRepair [25]. ✓represents correct fix, ? represents plausible fix, and × represents not generating fix. We use the * to mark the new generated fixes by providing the faulty methods when comparing with HDRepair. We highlight the correct repairs generated by SketchFix.

(A) A bug fix for Chart24 generated by SketchFix

```
//Human-Written patch for the defect
private int upperBound, lowerBound;
public Paint getPaint(double value) {
 double v=Math.max(value,this.lowerBound);
- int g=(int)((value-this.lowerBound)/...;
+ int g=(int)((v-this.lowerBound)/...; ...}
// A sketch generated by SketchFix and the synthesis result
public Paint getPaint(double value) {
 double v=Math.max(value,this.lowerBound);
 int g=(int)(((Integer)SketchFix.EXP(int.class,
   new Object[]{v,value,...})   - this.lowerBound)/...;}
 // Synthesized solution: SketchFix.EXP: v
```

(B) A bug fix for Math73 generated by Astor

```
//Human-Written patch for the defect Math73
  if (yInitial*yMax<0) return solve(f,initial,yInitial,...);
+ if (yMin*yMax>0) throw new llegalArgumentException(...);
   return solve(f, min, yMin,...);
//bug fix generated by Astor with API replacement
  if (yInitial*yMax<0) return solve(f,initial,yInitial,...);
-  return solve(f, min, yMin, max,...);
+  return solve(f, min, max);
```

(C) A bug fix for Lang58 generated by Nopol

```
//Human-Written patch for the defect
//Omit complex clauses for simplicity
- if (dec == null && exp == null && (A && B) || C) {..}
+ if (dec == null && exp == null && (A || D)) {..}
//bug fix generated by Nopol that is regarded as correct
- if (dec == null && exp == null && (A && B) || C) {..}
+ if ((dec == null) && (exp == null)) {..}
```

(D) A bug fix for Time15 generated by ACS

```
 //Human-Written patch for the defect and ACS's result is
 // semantically identical to this.
+ if (val1 == Long.MIN_VALUE) {
+   throw new ArithmeticException("...");}
```

(E) A bug fix for Math34 generated by HDRepair

```
 //Human-Written patch for the defect and HDRepair's result
 // is identical to this.
public Iterator<Chromosome> iterator() {
+ return chromosomes.iterator();
- return getChromosomes().iterator(); }
```

Figure 5: Comparison of Favored Defect Classes for SketchFix and Other Repair Techniques

Figure 4 denotes Venn Diagrams for the defects that can be correctly repaired by SketchFix and other techniques. Given that SketchFix, Astor and Nopol do not require supportive information such as documents (ACS [58]) or historical data (HDRepair [25]), we separate the comparison into two Venn Diagrams. Table 2 presents the repair result through manual inspection.

4.2.1 Defect Classes favored by the repair techniques . Different repair techniques focus on different types of bugs [39] (e.g., Nopol only deals with condition faults). We are not intended to perform an apple-to-apple comparison with these repair techniques, instead, we highlight favored defect classes of SketchFix and other techniques with examples.

SketchFix. Figure 5 (A) presents a defect that can be fixed by SketchFix. In this example, SketchFix applies the M_{exp} schema to a type-casting expression at the AST node-level. It synthesizes a variable `v` for the expression hole to replace the original variable `value`. Note that the constraint-solving-based repair techniques [36, 37, 40] in general only modify expressions in conditions or the right-side of assignments with `boolean` or `integer` types, other techniques that leverage statement-level mutation [35] or supportive information [25, 58] can hardly repair the defect because this AST node-level replacement has a rare chance to be duplicated in existing program context, history [25] or documents [58]. Table 3 reports the schema types that finally generate repairs for each defect. SketchFix correctly generates repairs for 19 subjects and around half of them (9 subjects) yield expression manipulation. The manipulation of operators and variable types (parameters for overloading methods) correctly fixes another 6 defects, and the rest schemas or their combinations fix another 4 subjects. For the 6 defects that can only be fixed by SketchFix with respect to other four repair techniques (Table 2), 4 of them yield expression manipulation, one of them is variable type manipulation in overloading methods, and the last one applies an insertion of the if-condition.

Astor. Astor is a program repair library that contains three repair modes based on the genetic algorithm, statement/condition removal, and operator mutation. Figure 5 (B) shows a defect that can be fixed by Astor. It replaces the `return` expression with another method invocation. The new return-statement exists in the current program context, and the Astor correctly generates this repair based on the statement-level search. SketchFix generates the same repair by transforming the original invocation to an overloading method with holes at the AST node-level.

Nopol. Nopol is a repair technique that focuses on the condition faults by modifying existing if-conditions or adding pre-conditions. Figure 5 (C) shows a defect and a bug fix generated by Nopol. The generated patch removes multiple clauses and is reported as correct in the literature [33]. SketchFix fails to repair this intricate condition defect that requires changes on multiple clauses.

ACS. ACS leverages document analysis to rank patches and it advances other techniques on fixing the omission error of the exception handling (*if-throw*). Figure 5 (D) presents an example of the repair generated by ACS which is semantically equivalent to the human-written patch. SketchFix is not designed to handle exception omission and we leave this as future work.

HDRepair. HDRepair prioritizes patches based on the mined repair models from historical data. It prefers the repairs that require the method invocation replacement. Figure 5 (E) shows a repair generated by HDRepair that replaces an expression with a method invocation. SketchFix can be extended to support API synthesis with a similar idea of the on-demand candidate generation [60].

Our experiment indicates that SketchFix performs particularly well in the defect classes that require manipulations of expressions and variable types. Recent empirical studies on repair models [28,

Table 3: SketchFix Repair Result for Each Subject

#		Fix	Type	Sk.	Sp.	cSk.	Exe	Cor	T(m)
1	CH1	C	M_{rop}	1.7k	301.2k	209	2.8k	2.9k	208.5
2	CH8	C	M_{exp}	607	8.7k	83	663	663	32.9
3	CH9	C	M_{con}	744	6.9k	678	100	100	5.6
4	CH11	C	M_{exp}	249	2.4k	25	26	31	7.2
5	CH13	P	M_{par}	813	79.8k	115	914	-	-
6	CH20	C	M_{exp}	137	2.9k	127	206	206	14.9
7	CH24	C	M_{exp}	17	378	1	4	4	0.5
8	CH26	P	M_{if}, M_{rtn}	77	1.6k	21	105	-	-
9	C14	C	M_{exp}	2.6k	46.1k	144	4	36	40.7
10	C62	C	M_{rop}	490	4.9k	58	72	92	8.8
11	C70	P	M_{par}	1.2k	18.9k	108	28	-	-
12	C73	P	M_{rop}	476	5.2k	59	40	-	-
13	C126	C	M_{con}	462	5.6k	32	8	64	7.3
14	L6	C	M_{exp}	255	3.0k	1	51	424	2.9
15	L51	P	M_{if}, M_{rtn}	222	2.4k	43	28	-	-
16	L55	C	M_{con}	108	1.6k	76	62	71	86.2
17	L59	C	M_{exp}	188	1.8k	48	5	5	0.1
18	M5	C	M_{exp}	1	72	1	1	1	0.1
19	M33	C	M_{par}	1.1k	17.8k	446	221	631	20.2
20	M50	C	M_{con}	655	6.0k	43	22	92	0.9
21	M59	C	M_{exp}	259	4.0k	36	7	7	0.4
22	M70	C	M_{par}	139	2.8k	60	8	8	0.1
23	M73	P	M_{if}, M_{rtn}	383	6.2k	76	59	-	-
24	M82	C	M_{rop}	500	6.0k	68	36	803	12.1
25	M85	C	M_{rop}	407	5.2k	78	590	590	23.1
26	T4	P	M_{par}	555	23.4k	40	274	-	-

Fix represents whether it is a correct fix (*C*) or a plausible fix (*P*). *Type* denotes the schema types that yield the repair. *Sk.* shows the number of generated sketches. *Sp.* presents the total search space of candidates. *cSk.* is the number of compiled sketches when SketchFix generates first repair. *Exe* represents the number of candidates SketchFix explores when it generates the first repair passing all tests. *C*or represents that number when SketchFix generates the first correct repair based on the manual inspection. *T(m)* reports the performance time to synthesize a correct fix.

34, 61] show the abundance of such defects. Yet SketchFix is not designed to handle statement-level mutation or exception handling.

4.2.2 Search Space Extension. In this section, we extend the search space of repair candidates to consider more suspicious statements and transformation schemas.

Suspicious Locations. By default, we consider top 50 suspicious statements as we notice that SketchFix cannot fix many defects with an increase of bound whereas the cost of the program repair increases significantly. If we increase bound to 100, SketchFix correctly fixes 2 more defects: Chart26 and Lang10. HDRepair [25] fixes defects with given faulty methods. With the same condition, SketchFix correctly fixes 6 more defects: Closure1, Closure11, Closure70, Closure73, Lang51 and Time19. This result also indicates that concise fault localization techniques can greatly improve the efficacy of program repair techniques.

Multiple Transformation Schemas. As a trade-off between the cost of experiments and the repair efficacy, SketchFix applies no more than two transformation schemas at a suspicious location. The number of generated sketches and corresponding candidates grow quickly with more transformation schemas. SketchFix fixes 4 more bugs with 3 schemas at a given location (Closure130, Lang58, Math53 and Time4).

Repair at Multiple Locations. We introduce an incremental approach to fix defects at multiple locations based on the reduction of the failing test cases [12]. If there exists multiple failing test cases, we localize the defect with one failing test case and all passing test cases, let SketchFix generate sketches and evaluate if any synthesized solution can eliminate one or more failing test cases. If a synthesized version reduces the number of the failing test cases, SketchFix applies this repair and keeps on considering other failing test cases. This incremental repair technique is based on the assumption that at least one failing test case can be fixed by synthesizing a sketch and failing test cases are independent with each other such that the fix of one test case will not affect others. Although the validity of this assumption remains unsettled, SketchFix correctly fixes another 3 defects (Chart14, Math4, and Math22) based on this assumption.

4.2.3 Threats to Validity. Although Defects4J contains hundreds of defects from multiple open-source projects, this defect benchmark may unavoidably yield the bias of the evaluation datasets [39] with respect to the defect types and the abundance of the defect classes [34, 39]. We compare our result with other four repair techniques that have been evaluated with the same benchmark and explain the favored defect classes for each technique. Based on the human-written patches from the version history, we assess the correctness of the generated repairs. However, this assessment may still be biased.

4.3 Search Space Reduction

In this section, we answer the second research question on the efficacy of our on-demand candidate generation.

SketchFix is able to generate repairs for the subjects shown in Table 3. With the default setting, these repairs include correct fixes that pass the manual inspection and the plausible ones that pass all tests but are regarded as semantically-different from the developer-written patches. The column *Fix* denotes whether the repair is correct (*C*) or plausible (*P*). The column *Type* represents the transformation schema types that are used to fix the bugs. If more than one schema combination can generate repairs that satisfy all test assertions, we report the first schema combination. The column *S*k. shows the total number of sketches generated by SketchFix considering all selected locations, and the column *S*p. lists the total search space of the candidates that the sketches represent. For instance, if SketchFix introduces a non- deterministic condition clause for a primitive type, assume that there are 10 expression candidates at both left and right hand side, the search space of candidates for this sketch is 600 ($6 \times 10 \times 10$) as two expressions are combined with a relative operator which has 6 options (==, ! = , >, <, $\leqslant$, $\geqslant$). The column *cSk.* presents the number of compiled sketches when SketchFix generates the first repairs that pass all tests. The column *Exe.* shows the number of candidates that have

been executed against the test suite when SketchFix generates the first repairs satisfying all tests. And the column *Cor.* represents the number when SketchFix finds the first correct repairs. The column *T(m)* illustrates the total performance time in minutes when SketchFix finds the first correct fix, including the time for the compilation and the test execution.

With the on-demand candidate generation, every sketch will be compiled once which may represent thousands of candidates. Up to the time that SketchFix finds the first repair, it compiles 1.6% (avg. #compiled sketches/#space). If SketchFix wants to exhaustively search the entire space of repair candidates, it only compiles 7% (avg. #sketches/#space) of all candidates. SketchFix utilizes runtime information to further prune the candidates that actually need to be verified by the test execution. For example, SketchFix will not generate candidates for the condition body if the condition is evaluated to be `false`. The experiment shows that SketchFix only executes 3% of candidates (avg. #Gen/#Space) when it finds the first repairs that pass all tests.

On average, SketchFix spends 9 minutes generating sketches with fault localization and 23 minutes generating the first repairs that satisfy all test assertions. To compare our performance with other repair techniques, we refer to the performance time reported in the literatures [7, 33, 35, 58] as we are not able to reproduce all experiments of other techniques due to the various experiment platforms. Astor and Nopol spend 30-40 minutes on average for each repair on a powerful machine. HDRepair spends an average of 20 minutes generating patches with given methods. Without reporting the performance time for document analysis, ACS's patch generation time (max. 28minutes, median 5.5minutes) is not fairly comparable with our repair technique. Without pre-processing historical data or document analysis, our practical repair technique compares well with other repair techniques on performance. We omit the comparison on the number of executed candidates across different repair techniques, because different techniques use different templates and the choice of templates determines the search space of the candidates.

4.4 The Quality of Generated Repairs

We answer the last research question in this section by investigating if our AST node-level transformation helps generate high-quality patches. Shown as Figure 6, we illustrate the quality of the repairs generated by SketchFix using an example.

Based on the AST node-level transformation schema, SketchFix transforms a method invocation to its overloading method, and correctly repairs the defect by replacing the variable `maxUlps` (`integer`) to `epsilon` (`double`). This fine-grained transformation is in line with the notion that the repairs which introduce smaller perturbations to the original programs are more likely to be correct from the perspective of the developers. With the same notion, SketchFix favors expression manipulations for the overloading method (M_{par}) rather than the if-condition insertion (M_{if}) as the insertion of a new if-condition will introduce more atomic holes than the expression manipulation. SketchFix generates sketches shown as Figure 6 (B) and synthesizes this sketch as expected.

The repair techniques Astor, ACS, and HDRepair fail to generate repairs for this bug. Nopol fixes this defect by adding a new `if` block. Although both solutions might be semantically equivalent,

(A) A human-written patch for the defect Math33

```
public class Precision {
 public static int compareTo(double x,double y,double eps)..
 public static int compareTo(double x,double y,int maxUlps)..
 }
 /* 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){...
}
```

(B) A sketch generated by SketchFix and a synthesized solution

```
 /* 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
```

(C) A plausible repair generated by Nopol

```
 protected void dropPhase1Objective() {
  if (Precision.compareTo(entry, 0d, maxUlps) > 0) {
+  if (numSlackVariables<constraints.size()) {...}}
```

Figure 6: Patches Generated by SketchFix and Nopol for the defect Math33

we believe that the repair generated from the fine-grained transformation introduces a smaller perturbation to the original faulty program and is conceptually easier to comprehend.

Apart from the fine-grained transformation, our candidate ranking strategy based on the locality also helps generating high-quality patches. Figure 5 (A) shows an expression manipulation that replaces the variable `value` with `v`, which is the maximum value of the variable `value` and the field `lowerBound`. Both `v` and `lowerBound` satisfy all test assertions, yet SketchFix favors the variable `v` because it is defined at the previous statement of the hole whereas `lowerBound` is a field in the class. With the insight that the variables defined closer to the holes are more likely to be used, our practical candidate ranking strategy is in line with other repair techniques that aim to improve patch quality, such as anti-patterns [53], program distance [5] and syntactic similarity [24].

5 RELATED WORK

This section compares SketchFix with other approaches on program repair and program synthesis.

Generate-and-Validate Repair. Generate-and-Validate repair techniques apply a set of transformations to generate a set of candidates and validate each candidate against the given test suite. To efficiently explore the immense search space of repair candidates in large-scale applications, these automated program repair tools leverage genetic algorithms [56], random search [43], semantic

search [21] and adaptive search [55] to find a repair that let all test cases pass. Kali [44] applies code removal and Debroy et al. [6] employ a set of mutation operators (e.g., arithmetic operators "+" and "×") to fix faults. Astor [35] is a program repair library that includes the re-implementations for GenProg [56], Kali [44], and mutation-based repair [6] to fix Java code.

To prioritize repairs, PAR manually learns repair patterns from the human-written patches [22]. Other G&V techniques automatically mine repair models from historical data [25, 30]. CodePhage [46] eliminates defects by transferring correct code across applications. Genesis [28] also automatically infers repair models based on the human-written patches. Some recent works leverage document analysis [27, 58], anti-patterns [53], test generation [57], and location selection with test pruning [38] to enhance repair.

SketchFix differs from these G&V approaches in that it tightly integrates the generation and validation phase of the repair candidates and only generates on-demand ones during the test execution. This integration substantially reduces the search space of the repair candidates and effectively prunes large amount of re-compilation and re-execution of the potential repairs. Moreover, a number of approaches that aim to improve the patch quality [53, 57] are complementary to ours. In particular, we can use mined repair models [25, 28, 30] and document analysis [58] to prioritize candidates and support more transformation schemas at a fine granularity.

Constraint-Solving Repair. Program repair techniques such as Nopol [7] and Angelix [37] dynamically collect path conditions and infer constraints based on the passing and failing test executions [36, 40]. These constraints are further translated to SAT for SAT/SMT solvers to synthesize a repair that satisfies all inferred specifications. SPR [29] infers a target value for the faulty condition based on the test suite. MintHint uses the symbolic execution and statistical correlation analysis for the patch ranking. S3 [24] extends the semantics-based family like SemFix [40] and Angelix [37] to incorporate a set of ranking criteria such as the variation of the execution traces similar to Qlose [5].

By inferring constraints based on the execution and translating these constraints to SAT, these repair techniques may yield incomplete translation [24, 37] or impractical problem of creating models for all relevant libraries [36, 40]. In general, they exclusively reason about `boolean` or `integer` types in conditions or the right side of assignment. They may suffer from the limitations of the symbolic execution engines to extract constraints. E.g., Angelix cannot execute libraries such as `python`, `lighttpd` and `fbc` due to the limitation of KLEE. SketchFix is substantially different from these translation-based techniques as its sketch backend [13] explores the actual program behaviors *in presence of* libraries and does not translate to SAT/SMT. SketchFix thus works well in manipulating expressions and handles a whole range of variable types and transformation locations, e.g., SketchFix fixes a bug by replacing a variable in a cast expression at the AST node-level. In addition, SketchFix is not confined to the structure of the subjects as long as the subjects can be executed. Therefore, it can be applied to the projects like `Closure compiler` whose test cases are organized using scripts rather than standard JUnit test cases. This non-conventional structure has obstructed some repair techniques like Astor.

Automated program repair based on the formal specifications has had numerous success. AutoFix-E [54] is able to repair complex data structure for Eiffel program, but it relies on human-written contracts to generate fixes. Gopinath et al. [11] use pre- and post-conditions written in Alloy specification language to identify defects and repair the program, and were the first to conjecture the reduction of program repair to program sketching [12] from the best of our knowledge. Singh et al. [47] use specification to generate feedback for students' faulty python program. Kneuss et al. [23] generate test inputs automatically from the specifications to fix faulty Scala programs using deductive program synthesis. Yet formal specifications are usually not available for projects written in imperative languages.

Domain-Specific Repair. ClearView [42] eliminates security vulnerability with inferred invariants. Demsky et al. [8] use Daikon to infer invariants for data-structure repair. Vejovis [19] infers possible string replacement for DOM-related faults in JavaScript code through string constraint solver. Similar to Vejovis, PHPRepair [45] uses string constraint-solving to fix PHP programs that generate HTML. These approaches share similar spirit with constraint-solving repair and thus may encounter similar limitations.

Program Synthesis. Program synthesis has shown its promise on synthesizing code in small well-defined domains such as bit-vector logic [18] and data structures [48]. In particular, sketch-based synthesis [50] asks programmers to write a program skeleton containing holes, and uses counter-example-guided inductive synthesis to complete the holes. However, these approaches require to transform constraints to SAT formulas, which might be impractical in large-scale projects with external libraries. EdSketch [13] is a practical execution-driven sketching engine that synthesizes sketches with backtracking search, yet it requires users to provide partial programs as the inputs. Recent work [60] enhanced EdSketch to support method sequence generation for API completion. Different from EdSketch and other program synthesis techniques [10, 41, 51], we translate existing faulty programs to sketches with the purpose of repairing the original faulty programs.

6 CONCLUSION

This paper introduced an on-demand repair technique, SketchFix, which tightly integrates the traditionally separate phases of generation and validation of candidate programs that represent potential fixes to consider for the given faulty program. Utilizing runtime information, this integration effectively prunes a large portion of the search space for the candidate fixes. The technique reduces program repair to program synthesis by transforming faulty programs to sketches at the AST node-level granularity. SketchFix employs an execution-driven sketching engine to complete the sketches using a backtracking search. The experimental results show that SketchFix works well in manipulating expressions, and the fine-grained transformation schemas it employs allow it to generate high-quality patches.

ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-1319688, CCF-1704790, and CCF-1718903). We thank Mukul Prasad and Yuqun Zhang for discussions and comments.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing - Practice and Research Techniques, 5th International Academic and Industrial Conference, TAIC PART 2007.*

[2] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing event race errors by controlling nondeterminism. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017.* 289–299.

[3] ASM Java bytecode manipulation and analysis framework 2017. (2017). http://asm.ow2.org/ Accessed: 07-30-2017.

[4] Closure Compiler 2017. https://github.com/google/closure-compiler. (2017). Accessed: 2018-02-10.

[5] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II.* 383–401.

[6] Vidroha Debroy and W. Eric Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *3rd IEEE International Conference on Software Testing, Verification and Validation, ICST 2010.* 65–74.

[7] Favio Demarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014.*

[8] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. 2006. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis, ISSTA 2006.* 233–244.

[9] Edit Distance 2017. (2017). https://en.wikipedia.org/wiki/Edit_distance Accessed: 07-30-2017.

[10] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. 2014. CodeHint: dynamic and interactive synthesis of code snippets. In *36th International Conference on Software Engineering, ICSE 2014.* 653–663.

[11] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-Based Program Repair Using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011.* 173–188.

[12] Jinru Hua and Sarfraz Khurshid. 2016. A Sketching-Based Approach for Debugging Using Test Cases. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016.* 463–478.

[13] Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: execution-driven sketching for Java. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017.* 162–171.

[14] Java programming language agents 2017. (2017). https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html Accessed: 07-30-2017.

[15] JavaParser Transformation Tool 2017. http://javaparser.org. (2017). Accessed: 2017-07-30.

[16] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: sketching for Java. In *23th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE 2015.* 934–937.

[17] JFreeChart Project 2017. (2017). http://www.jfree.org/jfreechart/ Accessed: 07-30-2017.

[18] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *32th International Conference on Software Engineering, ICSE 2010.* 215–224.

[19] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Ali Mesbah. 2014. Vejovis: suggesting fixes for JavaScript faults. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014.* 837–847.

[20] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA 2014.* 437–440.

[21] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015.* 295–306.

[22] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE 2013.* 802–811.

[23] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. 2015. Deductive Program Repair. In *Computer Aided Verification - 25th International Conference, CAV.*

[24] 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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 593–604.

[25] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1.* 213–224.

[26] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software Quality Journal* 21, 3 (2013), 421–443.

[27] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically Generating Bug Fixes from Bug Reports. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013.* 282–291.

[28] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 727–739.

[29] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 23th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE 2015.* 166–178.

[30] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016.* 298–312.

[31] Fan Long and Martin C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016.* 702–713.

[32] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2017. Automated repair of layout cross browser issues using search-based techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017.* 249–260.

[33] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964.

[34] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205.

[35] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *International Symposium on Software Testing and Analysis, ISSTA 2016.* 441–444.

[36] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *37th International Conference on Software Engineering, ICSE 2015.* 448–458.

[37] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *38th International Conference on Software Engineering, ICSE 2016.*

[38] Benjamin Mehne, Hiroaki Yoshida, Mukul Prasad, Koushik Sen, Divya Gopinath, and Sarfraz Khurshid. 2018. Accelerating Search-based Program Repair. In *11th IEEE Conference on Software Testing, Validation and Verification (ICST).* To appear.

[39] Martin Monperrus. 2014. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *36th International Conference on Software Engineering, ICSE 2014.* 234–242.

[40] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE 2013.* 772–781.

[41] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014.* 43.

[42] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2009. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009.* 87–102.

[43] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *36th International Conference on Software Engineering, ICSE '14.* 254–265.

[44] Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis, ISSTA 2015.* 24–36.

[45] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. 2012. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland.* 277–287.

[46] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* 43–54.

[47] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013.* 15–26.

[48] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing data structure manipulations from storyboards. In *19th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2011*. 289–299.

[49] 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 *23th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE 2015*. 532–543.

[50] Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495.

[51] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*. 313–326.

[52] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *International Symposium on Software Testing and Analysis, ISSTA 2013*. 314–324.

[53] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 727–738.

[54] Yi Wei, Yu Pei, Carlo A. Furia, Lucas Serpa Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis, ISSTA 2010*. 61–72.

[55] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*. 356–366.

[56] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 364–374.

[57] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 226–236.

[58] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 416–426.

[59] Jifeng Xuan and Martin Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2014*. 191–200.

[60] Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Test-Execution-Driven Sketching for Complex APIs. In *11th IEEE Conference on Software Testing, Validation and Verification (ICST)*. To appear.

[61] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 740–751.