International Journal on Software Tools for Technology Transfer (2019) 21:249–265
https://doi.org/10.1007/s10009-019-00512-8

SPIN 2017

EDSKETCH: execution-driven sketching for Java

Jinru Hua[1] · Yushan Zhang[2] · Yuqun Zhang[2] · Sarfraz Khurshid[1]

Published online: 16 March 2019

Abstract
Sketching is a synthesis approach that allows users to provide high-level insights into a synthesis problem and let synthesis tools complete low-level details. Users write *sketches*—partial programs that have "holes" and provide test assertions as the correctness criteria. The sketching techniques fill the holes with code fragments such that the complete program satisfies all test assertions. Traditional techniques translate the sketching problem to propositional satisfiability formulas and leverage SAT solvers to generate programs with the desired functionality. While effective for a range of small well-defined domains, such translation-based approaches have a key limitation when applying to real applications: They require either translating all relevant libraries that are invoked directly or indirectly by the given sketch or creating models of those libraries, which requires much manual effort. This paper introduces *execution-driven sketching*, a novel approach for synthesizing Java programs with *on-demand* candidate generation. The key novelty of our work is to leverage runtime behavior to prune a large amount of search space. EDSKETCH explores the actual program behaviors *in the presence* of libraries and sketches small parts of real-world applications, which may use complex constructs of modern languages, such as reflection, native calls and File I/O. We further leverage a set of pruning strategies based on Java syntax to expedite the synthesis process. EDSKETCH embodies our approach in two forms: a stateful search based on the Java PathFinder model checker; and a stateless search based on re-execution inspired by the VERISOFT model checker. Experimental results show that EDSKETCH can complete some sketches that contain complex constructs in the presence of libraries, recursive procedures and advanced features like reflection. Without translating to SAT, EDSKETCH's performance compares well with the SAT-based SKETCH system for a range of small but complex data structure subjects.

Keywords Program sketching · Execution-driven synthesis · Backtracking search

1 Introduction

Program sketching [1] is a program synthesis approach [2–5] where developers write partial programs that have "holes" and let the synthesizer fill in the holes based on the given specification, which is usually provided in the format of test harnesses or reference implementations. Existing sketching approaches [1,6] translate the partial program to propositional satisfiability formulas and leverage SAT solvers to generate a program that satisfies all constraints. While these translation-based approaches have shown their effectiveness on a range of small well-defined domains, they have a key limitation: when applying to real applications with libraries or programs with complex constructs like reflection, these translation-based approaches require either translating all libraries that are invoked directly or indirectly by the given sketch or creating models of all those libraries and complex constructs, which can lead to impractical SAT problems.

To tackle this limitation, we introduce EDSKETCH, a novel approach that performs *execution-driven sketching* to synthesize Java programs using backtracking search. The key novelty of our work is to introduce an *on-demand* candidate generation technique that leverages runtime behavior to substantially prune a large amount of search space and efficiently explore the actual program behaviors in the presence of libraries. To illustrate, consider trying to sketch a

✉ Yuqun Zhang
zhangyq@sustc.edu.cn

Jinru Hua
lisahua@utexas.edu

Yushan Zhang
zhangysh@mail.sustc.edu.cn

Sarfraz Khurshid
khurshid@utexas.edu

[1] The University of Texas at Austin, Austin, TX, USA

[2] Southern University of Science and Technology, Shenzhen, China

while condition as well as the body of the while loop, if a test execution raises an exception upon evaluating a specific candidate for the while loop condition, all candidates of the while loop body are pruned from search for that choice of the candidate condition expression. If the while loop body is not executed, our approach for lazy candidate generation will not create any candidates for the while loop body, which may contain thousands of candidates. When a test fails due to either a runtime exception or a test assertion failure, the parts of the candidate program that were executed directly determine the generation of the future candidates.

As inputs, EDSKETCH takes a sketch (partial program) with holes written in Java syntax and a test suite that characterizes the correctness specification. EDSKETCH executes the test suite against the sketch and backtracks the search when it encounters a failure (runtime failure or test assertion failure) and tries the next candidate. EDSKETCH terminates when the space of candidate programs is exhausted or a complete program that satisfies all tests is found.

EDSKETCH supports four kinds of holes: expressions (e.g., field dereferences), arithmetic operators ({+, −, ×, /}), Boolean conditions (e.g., while loop), and blocks of assignment statements. To initialize the search, EDSKETCH generates candidate expressions for the holes based on the target type and variables given by the user. For instance, using up to two field dereferences, the expressions of the type Entry derived from a variable e that represents an entry in a singly linked list should be {e, e.next, e.next.next}, where the field next represents the next entry in the linked list.

EDSKETCH introduces pruning strategies to expedite the sketching process. These strategies prune redundant candidates for assignment statement blocks and condition expressions. If the program violates a pre-defined pruning rule, EDSKETCH backtracks immediately and tries the next alternative candidate. For assignment blocks, we define a set of pruning rules based on the program isomorphism and Java syntax. For conditions, we introduce a value grouping strategy that splits all condition candidates into two sets based on their values (true and false) of the current iteration, and continue splitting the two sets based on the evaluated values for each iteration.

We embody our approach in two forms of prototypes: EDSKETCH- JPF, a stateful search based on the Java PathFinder model checker [7]; and EDSKETCH- JVM, a stateless search based on re-execution inspired by the VERISOFT model checker [8].

Experimental results show that EDSKETCH's performance compares well with the SAT-based SKETCH synthesizer [1] based on a dataset of small yet complex data structures. Out of 43 sketching tasks, EDSKETCH outperforms the SKETCH synthesizer on 40 tasks. Using three recursive data structures, EDSKETCH successfully completes all of them in a second, whereas SKETCH synthesizer fails in one given the same setting as EDSKETCH. The experiments also show that our pruning strategies are able to prune an average of 35% of candidates before evaluating them against the tests. Moreover, EDSKETCH completes some sketches that require handling reflection, I/O, native calls, and external libraries.

This paper makes the following contributions:

- **Execution-driven sketching** We introduce an execution-driven approach for program sketching that lazily generates candidates *on-demand* by leveraging the runtime behavior;
- **Pruning strategies** We introduce pruning strategies derived from the Java syntax to reduce the search space of candidates that must otherwise be explored;
- **Embodiment** We embody EDSKETCH into two prototypes: one based on the stateful model checker JAVA PATHFINDER [7], and the other based on a dedicated stateless backtracking search using re-execution in the spirit of the VERISOFT model checker [8];
- **Evaluation** We compare EDSKETCH with the SAT-based SKETCH synthesizer based on a set of small but complex data structure subjects. We illustrate EDSKETCH's ability to sketch partial programs in the presence of libraries, recursive procedures and advanced features like reflection.

2 Motivating example

We use a sketching task of reversing a singly linked list to illustrate our approach. Assume that the users want to implement a reverse() method for the singly linked list. Each list has a head entry, and each entry has a next entry and a value integer as fields. To reverse the singly linked list, the users have the notion that they need a while loop to traverse the list, and some local variables are required to record the current entry, previous entry and the next entry during the list traversal. They are also able to provide unit tests that specify the desired behavior as shown in Fig. 1c. However, it is hard for the users to complete the detailed implementation for the condition and the body of the while loop.

EDSKETCH enables users to provide high-level insights and leaves the implementation details to the synthesizer. Based on the high-level understanding of the problem, the users provide three local variables with the type Entry and a while loop to traverse the list, and leave the condition of the while loop as well as its body to be synthesized by EDSKETCH. Figure 1a shows a code skeleton written by the users that contains unknown while condition at line 7 and an unknown block of assignments at line 8. To specify the correctness criteria of the program sketch, the users write a JUnit test case that contains three entries of the linked list, as shown in Fig. 1b.

(A) A program sketch written by the user

```
1. class LinkedList {
2.  Entry head;
3.  public void reverse() {
4.   Entry ln1 = head;
5.   Entry ln2 = null;
6.   Entry ln3 = null;
7.   while (EdSketch.COND(...)) {
8.    EdSketch.BLOCK(...);
9.   }    }  }
10.class Entry {
11. Entry next;
12. int value;
13. }
```

(B) A JUnit test case provided by the user

```
  @Test
1. public void testThreeEntries() {
2.  LinkedList list = new LinkedList(new int[]{1,2,3});
3.  list.reverse();
4.  assertEquals("[3,2,1]",list.toString());
5. }
```

(C) A solution generated by EDSKETCH

```
1. COND Hole:
2.    ln1 != null
3. BLOCK Hole:
4.   head = ln1;
5.   ln1 = head.next;
6.   head.next = ln2;
7.   ln2 = head;
```

Fig. 1 A sketch example for singly linked list reversal

This sketching task is not trivial: Consider four visible variables (head, ln1, ln2, ln3), their field dereferences (given a minimum number of field dereferences required for this sketch, the candidate field dereferences are head.next, ln1.next, ln2.next, ln3.next) and a default value null for non-primitive types, the candidate for each assignment can be 8×9 given that null can only be the right-hand-side expression, and thus the search space for four consecutive assignment block is 72^4 (The sketch requires at least 4 assignments). Moreover, considering a condition expression as a combination of left-hand-side and right-hand-side expression together with a relational operator (either != or == for non-primitive types), the total search space for this sketch is 4.4 billion ($72^4 \times (9 \times 9 \times 2)$).

EDSKETCH dynamically selects candidates for the sketch invocations (line 7 and line 8) when it executes the given test cases. When EDSKETCH first reaches the while condition hole, EDSKETCH groups all condition candidates based on their evaluated value (true and false), and non-deterministically considers two Boolean possibilities for the execution. When the value false is considered, the unknown assignment block in the body of the while loop will not be executed, and thus all candidates of the assignment block hole are ignored and EDSKETCH will not create any additional search space for the assignment block hole.

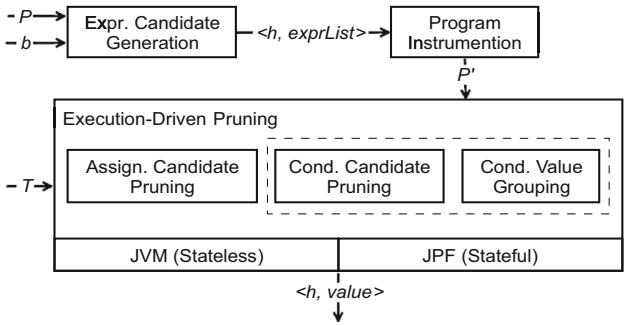

Fig. 2 EDSKETCH architecture diagram

When EDSKETCH first reaches the assignment sketch at line 8, it non-deterministically selects expressions for the right-hand-side and the left-hand-side of the assignment statement. A simple search may explore many candidates that are subsumed by other candidates which are already being explored. EDSKETCH prunes a number of such candidates based on the program isomorphism (Sect. 3.5). EDSKETCH backtracks its search if the current choice fails due to a runtime exception or a test failure. By default, EDSKETCH incrementally adds one assignment at a time until it finds the first solution or reaches the pre-defined upper bound on the number of assignments.

Figure 1c presents a solution generated by EDSKETCH based on a single test case shown in Fig. 1b. In this example, EDSKETCH finds the first solution in 9 s after exploring over 490 thousand candidates.

3 Approach

We describe our execution-driven sketching in this section. We first define the syntax of unknown holes h in Sect. 3.1. As shown in Fig. 2, EDSKETCH constructs all candidates based on the visible variables $exprList$ specified in the sketch P and the bound of field dereference b (Sect. 3.2). If the sketch contains assignment block holes, EDSKETCH instruments the program so that it can dynamically select candidates for the holes (Sect. 3.3). Section 3.4 discusses our backtrack engines for stateful prototype (EDSKETCH- JPF) and stateless prototype (EDSKETCH- JVM). Section 3.5 describes the pruning strategies we apply to sketch assignments and conditions when evaluating candidates via the test execution (T).

3.1 Partial expression syntax

Figure 3 denotes the syntax of the holes. We define two basic types of partial expressions for the sketches: expression holes and operator holes. The atomic expres-

atomic expr	e	$:= var \mid const \mid var.f$
constant	$const$	$:= null \mid true \mid false \mid k\ (0, 1, -1)$
arithmetic op	aop	$:= + \mid - \mid \times \mid / \mid \%$
relational op	rop	$:= == \mid != \mid > \mid < \mid \leqslant \mid \geqslant$
operator	op	$:= aop \mid rop \mid \mid\mid \mid \&\&$
composite expr	e	$:= e_1\ op\ e_2$
condition	$cond$	$:= e_1\ rop\ e_2$
assignment	$assign$	$:= e_1 = e_2;$
block	$block$	$:= assign1;$
		$assign2; \ldots$

Fig. 3 Syntax of partial expressions

sion holes (EdSketch.EXP) represent visible variables, constant values, and field dereferences. As to the operator holes, we define arithmetic operators $\{+, -, \times, /, \%\}$ (EdSketch.AOP) and relational operators $\{==, !=, >, <, \leqslant, \geqslant\}$ (EdSketch.ROP). EDSKETCH generates composite expressions by combining expression holes with operator holes, including arithmetic operators, relational operators and logical operators (|| and &&). Composite expressions can further combine together to generate complex expressions. For instance, we define a hole for conditions (EdSketch.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. Moreover, EDSKETCH provides the assignment block holes (EdSketch.BLOCK) to specify a list of consecutive assignments.

To specify these holes in Java syntax, EDSKETCH provides a list of method invocations that take three parameters: an object list that contains all visible variables and default values (null or 0, 1, − 1), a hole id to distinguish different holes for the same type, and the target type of the generated candidates as an optional parameter. The hole id is used to distinguish different holes, it must be unique within a type of the holes, i.e., two condition holes cannot have the same id whereas a condition hole can have the same id as a block hole. In our motivating example of Fig. 4, the user assigns an identifier 0 to the while condition and an identifier 0 to the assignment statement block sketch. If user does not specify the target type of the hole, EDSKETCH takes the first two parameters and treats the target type as another hole of object type. EDSKETCH enumerates all types that can be derived from the given object list within the bound of the field dereferences to fill in the hole of the object type. Note that SAT-based SKETCH synthesizer [9] also requires users to provide candidates for the "holes" (Fig. 6).

EDSKETCH provides a set of Java method invocations to specify different types of unknown holes. These APIs are treated as normal Java methods, thus our sketches can be directly compiled and executed using the given test suite. We highlight two APIs provided by EDSKETCH as below using the example shown in Fig. 1:

Condition Hole The complete invocation of the while condition synthesis at line 8 shown in Fig. 1 is: EdSketch.COND(new Object[] {head, ln1, ln2, ln3}, 0, Entry.class), which represents all conditional clauses whose left-hand-side and right-hand-side expressions are either the given variables or their field dereferences, combined with the relational operator holes. As the type Entry is non-primitive type, the relational operator hole consists of candidates {==, !=}.

Block Hole The complete invocation of the unknown assignment block at line 9 shown in Fig. 1 is: EdSketch. BLOCK(new Object[] {head, ln1, ln2, ln3}, 0, Entry.class), which represents a block of assignments that consist of either the given variables or their field dereferences. The number of assignments in the block is up to the pre-defined bound.

The purpose of program sketching is to bridge the gap between users' high-level insights of the expected program and the low-level implementation. Following the same spirit, EDSKETCH enables users to provide more insights by invoking a set of supportive methods.

Maximum number of field dereferences EDSKETCH generates up to one field dereference by default and makes this bound configurable. For example, the user can specify EdSketch.BLOCK(...).setFieldDeref(2) in Fig. 1 to allow up to two field dereferences. With this setting, EDSKETCH will generate field dereferences {ln1, ln1.next, ln1.next.next} derived from the ln1 in Fig. 1.

Enable default values or not We define the default value for integer and double as {0, 1, -1}, boolean as {true, false}, non-primitive types as null, and the type String as empty string. The user can exclude the default values as EdSketch.BLOCK(...).enableDefault(false) in Fig. 1.

Maximum number of assignments By default, EDSKETCH generates no more than four statements for the block holes. EDSKETCH also allows users to specify the number of assignment statements using the method set- Length(), e.g., EdSketch.BLOCK(...).setLength(5).

Note that the purpose of these bounds is to leverage users' insights to explore the search space in a more effective manner, yet there is no bound on how many holes users can introduce to the sketch. As long as the holes are covered by the test execution, EDSKETCH is able to generate candidate and validate whether there exists a solution that satisfies all tests.

3.2 Expression candidate generation

This section describes how we generate expression candidates. Shown as Algorithm 1, EDSKETCH leverages a

Algorithm 1: Expression Candidate Generation

```
   Input  : Partial program P, bound of field dereference b
   Output: Complete Program P' that pass all test cases
1  Function generateExpressions (P, b) is
2      exps[] ← ∅, exprList[] ← ∅ ;
       /* Expression candidates generation   */
3      foreach h ∈ holes(P) do
4          exps[h] ← fetchVariables();
5          len ← 0, i ← 0 ;
6          while len < b do
7              size ← exps[h].size ;
8              while i < size do
9                  exps[h] ←
                   exps[h] ∪ fields(exps[h].get(i));
10                 i + + ;
11             len + + ;
12         exprList[h] ← selectType(exps[h], type(h)) ;
13 Function getExpression (candList) is
14     if id == -1 then
15         id ← choose(0, candList.size − 1) ;
16     return candList[id] ;
17 Function getBlock (stmtList, candList) is
18     foreach stmt ∈ stmtList do
19         rhs ← getExpression(candList) ;
20         lhs ← getExpression(candList) ;
21         assign(lhs, rhs) ;
```

breadth-first iteration to generate field dereferences for all provided variables within a pre-defined bound of b (line 3-12). EDSKETCH creates all field dereferences using reflection and iteratively adds generated field dereferences to the list of candidates. After generating all candidates, EDSKETCH selects expression candidates based on the target type of the "hole" specified in the method invocation. In particular, the object *this* is also regarded as a variable yet we do not generate its field references because these fields are already included as heap-allocated variables. The implicit *length* field for the *array* type is not reflected by the *getFields()* method [10], and thus we manually insert this field if there exist candidates with the *array* type.

For each non-deterministic "hole", we put all its expression candidates in a vector called *candidate vector*. We assign each expression candidate a unique identifier, which is its index in the candidate vector. When EDSKETCH performs sketching, it dynamically selects a candidate identifier for each "hole" using non-deterministic choose() operator and executes the program based on the candidate it selects. All candidate identifiers for the holes are initialized as −1 (Algorithm 1 line 14), indicating that EDSKETCH has not selected a candidate for this "hole". Once EDSKETCH selects a candidate identifier for this hole, this candidate will be used consistently across all test cases. When EDSKETCH backtracks, it increments the candidate identifier and re-executes the program with the next candidate that corresponds to the new identifier of the hole.

3.3 Program instrumentation

To introduce non-determinism to the program and allow a backtracking search to explore the space of candidate programs, we need to instrument the sketches in the following procedures and transform them to executable programs that can be validated by the test execution.

Loops and Recursive Procedures To get rid of infinite loops during the synthesis of while loop, following the same spirit of SAT-based SKETCH synthesizer, we set up a bound for the while loop iteration and backtrack whenever the execution has exceeded the pre-defined bound, shown in Fig. 4 line 10–11. By default we set the bound EdSketch.LOOP_BOUND as 16 (the default loop bound for the SAT-based SKETCH synthesizer) and make it configurable to the end users. For recursive procedures, we use a heuristic to check whether the method itself is directly called in its body, and introduce a configurable bound EdSketch.RECUR_BOUND as 16 at the beginning of the recursive procedures, shown in Fig. 11 line 2–3. We leave the checking of indirectly recursive procedures as future work.

Assignment Block Figure 4 presents the instrumented program for the motivating example, and highlights the newly instrumented code using the "+" signal. EDSKETCH generates a for loop to sketch the assignment statement block. This for loop enumerates each statement in the list, which is the return value of the assignment statement block hole EdSketch.BLOCK (...). In this for loop, EDSKETCH assigns values to the left-hand-side expressions for each assignment using a switch statement to select the left-hand-side expression

The instrumented program for the example of Figure 1

```
1.  class LinkedList {
2.   Entry head;
4.   public void reverse(){
5.    if (head==null)   return;
6.    Entry ln1 = head;
7.    Entry ln2 = null;
8.    Entry ln3 = null;
9.    int count=0;
9.   while (EdSketch.COND(Entry.class,0)) {
10.+  if (count++>EdSketch.LOOP_BOUND)
11.+   EdSketch.backtrack();
12.+  for (Statement s: EdSketch.BLOCK(Entry.class, 0)) {
13.+   EdSketch_Assign stmt = (EdSketch_Assign) s;
14.+   Entry rhs_val = (Entry) stmt.getRHS();
15.+   switch (stmt.getLHS_id()) {
16.+    case 0: head = rhs_val;
17.+    case 1: ln1 = rhs_val;
18.+    ...
19.+    case 7: ln3.next = rhs_val;
20.+   }   } }
```

Fig. 4 The instrumented program for the example of singly linked list reversal

Algorithm 2: Execution-Driven Sketching

```
   Input  : Partial program P, test suite T
   Output: Complete Program P' that pass all test cases
1  Function sketch () is
2      do
3          try
4              exploreCurrentChoice();
5          catch BacktrackException
6              createNextChoice() ;
7      while incrementCounter();
8  Function exploreCurrentChoice() is
9      try
10         foreach test ∈ T do
11             test.run() ;
12     catch TestFailureException
13         throw BacktrackException ;
14     printSolution() ;
15     searchExit() ; // first solution found
```

based on the candidate identifier for the "hole", and assigns the right-hand-side expression to the selected left-hand-side expression. We only generate case statements for variables and field accesses that can be assigned, and will not generate case statements for this object, unmodifiable fields like array.length, and default candidates such as *null*, 0, 1, −1. If the user does not provide the bound for the number of sketching assignments, EDSKETCH will add one assignment at a time until it finds the first solution or reaches the default bound for the number of sketching assignments.

3.4 Execution-driven sketching

As shown in Algorithm 2, EDSKETCH starts sketching the partial program by directly executing the test cases (line 9–13). Whenever it encounters a runtime exception or a test failure, it backtracks and fetches for the next choice until it has explored the entire search space or finds a solution that meets the correctness criteria. Shown as Function sketch() in Algorithm 2, whenever the test execution encounters an assertion failure or an exception, EDSKETCH throws a BacktrackException, increments the candidate identifier (incrementCounter()) and re-executes the test cases based on the new candidate identifier which maps to the next unexplored candidate. EDSKETCH prints a complete program if this program satisfies all test assertions.

We build two prototypes based on two different backtrack engines: a stateful prototype based on JAVA PATH FINDER [7] and a stateless prototype based on re-execution.

Stateful Prototype with Java PathFinder JAVA PATHFINDER (JPF) [7] is a mature model checker that implements a customized JVM to efficiently store and restore program states. While JPF cannot handle Java native calls and can hardly scale up to large applications with libraries and millions lines of code, using JPF as the backend for sketching opens the future work possibility to sketch multithreading programs [11] with systematic path exploration.

Stateless Prototype Using Re-Execution Our second prototype is based on a dedicated stateless search [8] using re-execution [12]. Since this prototype executes on the standard JVM, it allows synthesis in the context of open-source projects with advanced features, such as reflection, I/O, and native calls.

3.5 Pruning strategies

Brute force search yields a huge search space of sketch candidates, whereas some candidates can be isomorphic with each other and some candidates can be detected as wrong solution based on pre-defined roles. With the purpose of pruning the search space of sketch candidates, we discuss our pruning strategies for sketching assignments and conditions in this section.

3.5.1 Assignment pruning

We define four pruning rules based on the Java syntax and program isomorphism analysis. We omit the proof of these rules as they can be easily derived from the Java syntax. These rules may prune the program based on one assignment (rule 1) or two consecutive assignments in the sketching block (rule 2-4). For the rules below, we use e_1 to represent an expression which can be either variables or field dereferences, and use v_1, v_2, v_3 to represent variables. The method id() returns the candidate identifier of this candidate.

(1) "$e_1 = e_1$". If the left-hand-side expression is equal to the right-hand-side expression, the candidate is ignored as the assignment has no effect on the current program state. For example, any candidates that have the assignment ln1.next = ln1.next will be pruned in the motivating example as shown in Fig. 1.

(2) "$v_1 = v_2; v_2 = v_1$". If the left-hand-side variable of the first assignment is the same as the right-hand-side variable of the second assignment, and the right-hand-side variable of the first assignment is the same as the left-hand-side variable of the second assignment, this candidate is omitted because the two assignments are subsumed by the assignment $v_1 = v_2$. For example, in Fig. 1, the candidate will be pruned if it has two consecutive statements ln1= ln2; ln2 = ln1.

(3) "$v_1 = v_2; v_1 = v_3$". If both left-hand-side variables in two consecutive assignments are the same and both right-hand-sides are variables, we do not need to execute this program because it is subsumed by the assignment $v_1 = v_3$. For instance, the candidate in Fig. 1 with consecutive assignments ln2 = ln4; ln2 = ln3 will be ignored, because the two assignments are equivalent to a single assignment ln2 = ln3, which has been covered by the current search.

Algorithm 3: Condition Value Grouping

```
   Input  : Expression candidates exprList, two empty sets
            trueSet and falseSet
   Output: Next condition candidate
1  Function getCondition(exprList, trueSet, falseSet) is
2      if select ← −1 then
           /* Initialize condition candidates
              */
3          condCands ← construct(exprList, primOp) ;
4          foreach c ∈ condCands do
5              if eval(c) then
6                  trueSet ← trueSet ∪ c ;
7              else
8                  falseSet ← falseSet ∪ c ;
9      else if select==0 then
           /* Split trueSet                    */
10         falseSet ← ∅;
11         foreach c ∈ trueSet do
12             if eval(c)==false then
13                 falseSet ← falseSet ∪ c ;
14                 trueSet ← trueSet \ c ;
15     else
16         trueSet ← ∅;
17         foreach c ∈ falseSet do
18             if eval(c) then
19                 trueSet ← trueSet ∪ c ;
20                 falseSet ← falseSet \ c ;
21     if trueSet is empty then
22         select = 1 ;
23     else if falseSet is empty then
24         select = 0 ;
25     else
26         select ← choose(0, 1) ;
27     return ⟨select, trueSet, falseSet⟩ ;
```

(4) "$v_3 = v_1; v_2 = v_1$" while $id(v_3) > id(v_2)$. If two consecutive assignments have the same variable at the right-hand-side, we only execute the program if the identifier of the first assignment's left-hand-side variable is smaller than that of the second assignment. We consider the consecutive assignments "$v_2 = v_1; v_3 = v_1$" and "$v_3 = v_1; v_2 = v_1$" as isomorphic solutions and we only evaluate isomorphic solutions once. For example, in Fig. 1, we will not execute the program with consecutive assignments "ln4 = ln2; ln3 = ln2", as the candidate identifier of ln4 is bigger than that of ln3 and an isomorphic program with "ln3 = ln2; ln4 = ln2" has already been explored.

We also try to apply existing synthesizing optimization strategies to EDSKETCH. Counter-example-guided inductive synthesis technique (CEGIS) [4] has shown its effectiveness on a number of SAT-based synthesizers to add the validated counter-example to the propositional satisfiability formulas and further expedite the synthesizing process. It does not directly apply to EDSKETCH as EDSKETCH is purely based on execution and no SAT translation is involved. The candidate that has been validated against the test execution will never be generated again by EDSKETCH. Yet we borrow the idea of memorizing information from previous validation results to prune the candidates before validating them with the test execution. We implement a pruning strategy to reuse the validation result of the NullPointerException, i.e., if the first several consecutive assignments lead to a NullPointerException, we will not generate any assignment blocks staring with these assignments. However, our preliminary evaluation result based on the dataset specified in Sect. 4.1 indicates that the pure execution via JVM is often much faster than memorizing and further checking the validation conditions. Therefore, we keep the simplicity of the pruning strategies and leave the improvement in these strategies as future work.

3.5.2 Condition pruning

To sketch condition expressions including if conditions and while loop conditions, EDSKETCH first generates all condition candidates and splits these candidates into two groups based on their evaluated values (true and false). Algorithm 3 outlines how EDSKETCH sketches a condition expression.

Condition Candidate Generation During the first access of the condition sketch, EDSKETCH generates all condition candidates by combining expression candidates with relational operators rop. We define two relational operators $\{==, !=\}$ for non-primitive types and six condition operators for primitive types $\{==, !=, >, <, >=, <=\}$. The primitive type operators are also applied to corresponding wrapper classes such as Integer. We only need to consider the combination $e_1\ rop\ e_2$ where the candidate identifier of e_1 is smaller than that of e_2 ($id(e_1) < id(e_2)$) based on the program symmetry. Assume that e_1 and e_2 are two non-primitive expression candidates, and the e_1's candidate identifier is smaller than e_2, we only need to consider the condition candidates $e_1 == e_2$ and $e_1 != e_2$ because $e_2 == e_1$ and $e_2 != e_1$ are equivalent to the previous two candidates. If we have five expression candidates with the primitive type int, EDSKETCH will generate 60 ($6 \times (4 + 3 + 2 + 1)$) condition candidates by combining each candidate with the ones that have bigger identifiers using six condition operators. We also include two constant Boolean value true and false for completeness ($e_1 == e_1$ and $e_1 != e_1$).

Condition Value Grouping The generated condition candidates are further split to two sets based on their evaluated values, shown as line 9 to 20 in Algorithm 3. If it is not the first access, EDSKETCH will re-evaluate each candidate in the set and split the candidate set based on the evaluated value of each condition candidate in the new execution. If EDSKETCH selects boolean value true in the previous execution, the trueSet will be split in the current execution, and vice versa. For example, the condition candidate ln1 != null in the motivating example may be true in an iteration, and its value may change to false in the next iteration. Therefore,

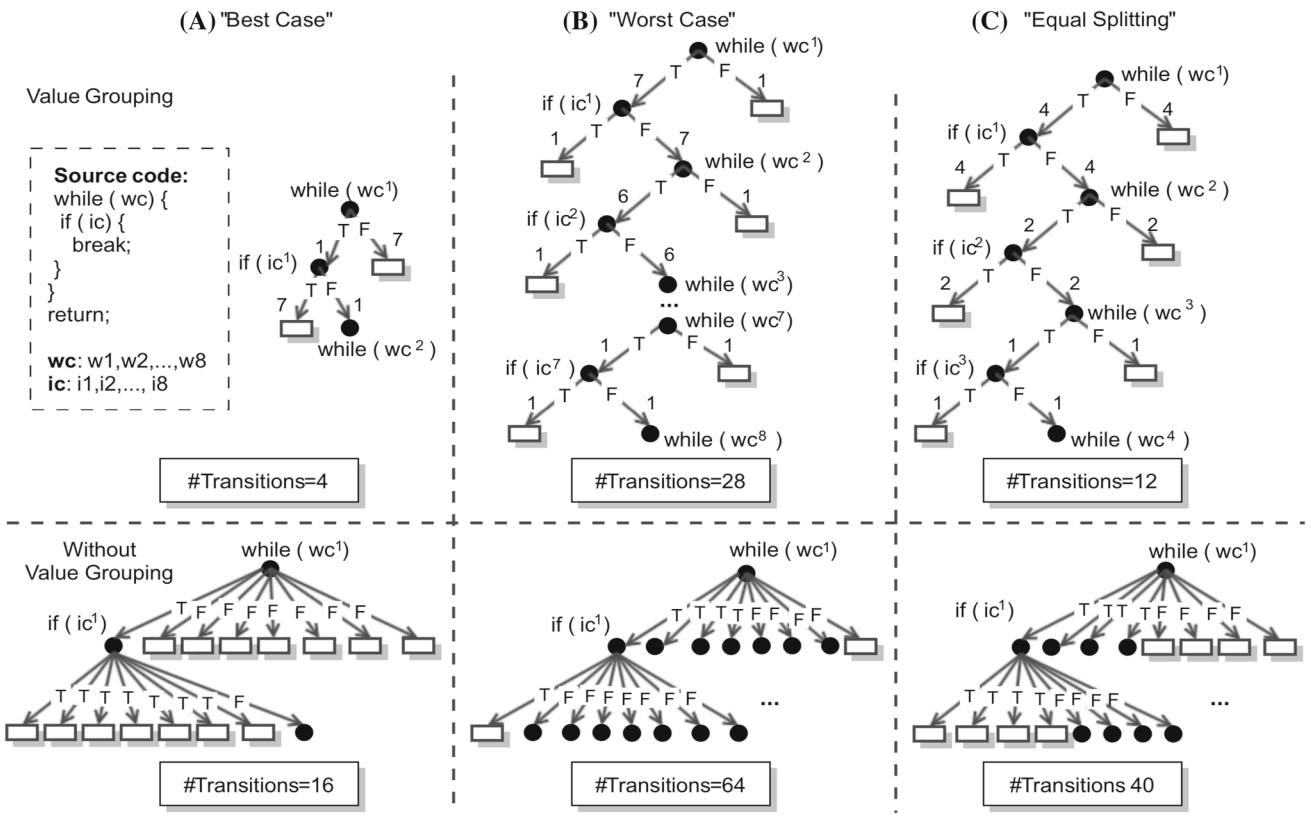

Fig. 5 EDSKETCH pruning example

the condition candidate ln1 != null will be put in trueSet in the first iteration and will be moved to falseSet in the next iteration.

EDSKETCH chooses a Boolean value at the end of each invocation based on the size of two candidate sets, shown as line 21 to 27 in Algorithm 3. If there is no candidate that is evaluated to be true, EDSKETCH will select false represented as 1 at line 22 in Algorithm 3. And if the set of candidates which are evaluated to be false is empty, EDSKETCH will select true represented as 0. The selected Boolean value is returned from the getCondition method together with two candidate sets. If the chosen value does not satisfy test assertions, EDSKETCH will backtrack to the previous choice, and select a different value based on the non-deterministic choose() operator.

To illustrate the efficacy of our condition value grouping strategy, we present three cases with and without value grouping and compare the number of transitions in each case. As shown in Fig. 5, assume the while condition has eight candidates ($wc \in \{w_1, w_2, \ldots, w_8\}$) and the if condition also has eight candidates ($ic \in \{i_1, i_2, \ldots, i_8\}$):

(1) "Best Case" In this case, we assume that there is only one candidate that is evaluated to be true for the while loop (wc^1) and false for the if condition (ic^1). wc^1 indicates the while loop condition in the first iteration. In this case, no further choices will be created because the program terminates after executing this candidate; we term this case as the "best case" as shown in Fig. 5a. Shown as edges from one node to another, the "best case" requires 4 transitions with value grouping and 16 transitions without value grouping.

(2) "Worst Case" We assume that there is only one candidate for which the condition evaluates to false for the while loop (wc^1) and true for the if condition (ic^1) when it is executed the first time. EDSKETCH will keep creating non-deterministic choices until it exhausts all choices; we term this case as the "worst case". As shown in Fig. 5b, EDSKETCH has 28 transitions with value grouping strategy, while the traditional approach without value grouping requires 64 transitions.

(3) "Equal Splitting" Lastly, we consider a case where candidates are equally split in each iteration. As shown in Fig. 5c, EDSKETCH with value grouping strategy requires 12 transitions while traditional approach requires 40 transitions.

Our example illustrates that our value grouping strategy effectively reduces the number of transitions in three cases.

4 Evaluation

To investigate EDSKETCH's efficacy of sketching partial programs, we curate two evaluation datasets: the first dataset consists of small but complex data structures including recur-

sive procedures, these subjects have been used to evaluate SAT-based sketch synthesizers in prior works [1,4,13,14], the second dataset contains partial programs in the presence of libraries and advanced features like reflection.

We address the following research questions in the evaluation:

- How effective is EDSKETCH to sketch small but complex subjects compared to the SAT-based synthesizer?
- How do the pruning strategies affect the search space of sketching?
- Can EDSKETCH complete synthesis tasks in the presence of libraries in open-source projects and advanced language features such as reflections?

4.1 Sketching tasks with data structures

To study EDSKETCH's efficacy of sketching small but complex data structures, we select 10 subjects from java.util source code and algorithm book [15] as reference implementation. Based on the reference implementation, we transform all if and while conditions to condition holes, and all expression assignments as assignment holes [16]. We do not evaluate expression holes separately because they are part of the condition and assignment holes.

As shown in Table 1, the 10 subjects are: Binary Search Tree Insertion (BSTAS and BSTCD), Finding Median (MEDAS and MEDCD), Red-Black Tree Insertion (RBTAS and RBTCD), Singly Linked List Reversal (LLREV), Doubly Linked List Add First (DLLAF), Doubly LinkedList Add Last (DLLAL), Red-Black Tree Removal (RBTRM), Fibonacci numbers using recursion (FIB), Singly Linked List Insertion using recursion (LLINS) and Binary Search Tree Insertion using recursion (BSTIS). We evaluate the performance and pruning strategies of sketching assignments and conditions separately considering different pruning strategies to these two types of sketching. We use the first 10 subjects to compare the performance of EDSKETCH- JPF, EDSKETCH-JVM and SAT-based SKETCH synthesizer and use the last 3 subjects to illustrate EDSKETCH's effectiveness to sketch recursive procedures.

To reach full branch coverage, we use 7 test cases from [17] for Find Median subjects and 4 test cases for the first 4 Fibonacci numbers in the subject of FIB. For the rest of subjects, we use KORAT [18] to generate bounded exhaustive test suites. KORAT is a test generation tool that uses given constraints to guide the generation of bounded suites. We use bounded exhaustive test suite up to three nodes for binary search tree, singly linked list, and doubly linked list, and test suite up to four nodes for red–black tree. We sort the test cases based on the number of nodes and execute EDSKETCH with test cases in ascending order.

Table 1 lists the average search space for the first five assignments or conditions. The subjects can have more than five partial expressions (RBTCD has 7 condition holes) or only 4 non-deterministic holes, whose search space of the fifth expression is marked as N/A. For example, the reference implementation of the Singly Linked List Reversal (LLREV) has 4 assignments (Fig. 1), and the search space of candidates for 3 assignments (262.4 K in Table 1) is calculated as the average search space of candidates for all combinations of 3 out of 4 assignments (i.e., the search space of candidates for the assignments {1, 2, 3}, {1, 2, 4}, {2, 3, 4}). Given 4 variables (head, ln1, ln2, ln3 in Fig. 1), one

Table 1 Evaluation subjects

	Type	Name	Tests	1	2	3	4	5
1	A	BSTAS	8	196	38.4 K	7.5 M	1.5 B	289 B
2	A	MEDAS	7	16	256	4.1 K	66 K	N/A
3	A	LLREV	4	64	4.1 K	262.4 K	16.7 M	1.1 B
4	A	RBTAS	15	676	457 K	309 M	209 B	N/A
5	A	DLLAF	4	196	38.4 K	7.5 M	1.5 B	N/A
6	A	DLLAL	4	196	38 K	7.5 M	1.5 B	N/A
7	C	BSTCD	8	392	190.5 K	74.7 M	29.3 B	14.2 T
8	C	RBTRM	15	74	5.5 K	7.4 M	10.0 B	N/A
9	C	MEDCD	7	96	9.2 K	884.7 K	84.9 M	8.2 B
10	C	RBTCD	15	74	5.5 K	7.4 M	10.0 B	13.5 T
11		FIB	4	1COND, 1AOP			24	
12		LLINS	4	3EXP			108	
13		BSTIS	8	2COND, 5EXP			24	

Column *Type* represents the sketching type: *A* represents assignments and *C* represents conditions. If the subject has only four statements, the search space for the fifth statement is marked as N/A. The last three subjects are recursive procedures. *1COND* represents one condition hole, *1AOP* represents one arithmetic hole, *3EXP* represents three expression holes

(A) A sketch written in Sketch syntax [40]

```
1.void reverse(LinkedList l) {
2. Entry ln1 = l.head;
3. Entry ln2 = null;
4. Entry ln3 = null;
5. while({|(l.head|ln1|ln2|ln3)(.next)?(==|!=)
   (l.head|ln1|ln2|ln3)(.next)?|null|}){
6.   minrepeat {
   {|(l.head|ln1|ln2|ln3)(.next)?|}= {|(l.head|ln1
    |ln2|ln3)(.next)?|};
7.  } } }
```

(B) Test harnesses written in Sketch Language

```
1.harness void test() {
2. l = newList(); ...// insert 1,2,3
3. reverse(l);
4. assert l.head.value == 3;
5. assert l.head.next.value == 2;
6. assert l.head.next.nextvalue == 1;}
```

Fig. 6 Singly linked list reversal written in Sketch syntax [1]

field dereference (head.next, ln1.next, ln2.next, ln3.next) without considering the default value null, the number of candidates for one assignment are 8×8. The number of candidates for three assignments (say assignment {1, 2, 3}) are $64^3 = 262.4K$, and the average candidate search space for 3 assignments are also 262.4K. Similarly, the search space of candidates for condition holes in the findMedian method will be 96 considering six operators (==, ! =, >, <, >=, <=) and 4 candidates at both sides of the condition, thus synthesizing four conditions can have 96^4 candidates. Using these subjects, we first compare the sketching performance of EDSKETCH with SAT-based Sketch synthesizer, and then illustrate the efficacy of our pruning strategies.

Comparison with Sketch Synthesizer We compare EDSKETCH with SKETCH synthesizer [1], a state-of-the-art SAT-based synthesizer that has been evaluated as a benchmark for other synthesizers [4,19]. We choose SKETCH synthesizer because it can sketch assignments, conditions and expressions similar to EDSKETCH, whereas other synthesizers focus on API sequences and cannot sketch assignments blocks (e.g., CODEHINT [20], SYPET [21] and EDSYNTH [22].

We manually transform the Java subjects and test suites to SKETCH language, which is a type-based language similar to Java. Figure 6 illustrates the Singly linked list reversal example written in SKETCH language. To make sure the subjects written in SKETCH language are semantically equivalent to the subjects for EDSKETCH (Fig. 1), we provide the same variables, field dereferences and relational operators for the subjects written the SKETCH language. Note that the keyword minrepeat in SKETCH language shares the same logic as EDSKETCH: adding one statement at a time until a solution is found.

We execute EDSKETCH- JVM, EDSKETCH- JPF, and SKETCH synthesizer on the subjects using the same test suites and report the time when they find the first solution that satisfies all test cases. All performance experiments are conducted on a MacBook Pro with 2.2 GHz Intel Core i7 processor and 16 GB of 1600 MHz DDR3 memory running OS X version 10.12.1.

Figure 7 represents the sketching performance time of EDSKETCH- JVM, EDSKETCH- JPF, and SKETCH synthesizer for sketching different numbers of assignments. The x-axis shows the number of statements under sketching, and the y-axis represents the average performance time for sketching tasks with specific number of assignments. The y-axis in for the last 4 subjects in Fig. 7 is transformed with $log2$ scale for better display. The green line with triangle represents the performance of EDSKETCH- JVM, the red line with circle represents EDSKETCH- JPF, and the blue line with square represents Sketch synthesizer.

Figure 7 indicates that EDSKETCH is able to sketch small but complex data structures with a better performance compared to SAT-based inductive synthesizer in 20 out of 22 sketch tasks. For example, for the subject LLREV, EDSKETCH- JVM sketches the first correct solution with 5 assignments in 3.1 s while Sketch synthesizer takes 11.7 s for the same task. EDSKETCH- JVM is slower than Sketch synthesizer in 2 experiments: DLLAF with four assignments (22.4 vs. 1.9 s) and the subject DLLAL with 4 assignments (25.8 vs. 2.8 s). The average performance time for EDSKETCH- JVM is 16.2 s while the Sketch synthesizer is 25.1 s.

Figure 8 presents the performance of three tools for sketching conditions, including if conditions and while conditions. For instance, for the subject RBTCD, EDSKETCH- JVM spends an average of 0.06 s sketching 6 conditions while SKETCH synthesizer spends 58.9 s. It might because the transformation of red-black tree implementation to boolean formulas is not trivial. and it takes a long time for the SAT solver to find a solution. EDSKETCH- JVM is faster than SKETCH synthesizer in 19 out of 21 experiments. EDSKETCH- JVM is slower than SKETCH synthesizer in the subject BSTCD with three and four conditions (1.7 vs. 0.8 s and 10.5 vs. 1.7 s).

Previous work [19] conjectured that the backtracking solver purely based on concrete execution will be much slower than SAT-based solver in exploring large state space due to the highly optimized heuristics used by modern SAT engines. However, our experiments show that EDSKETCH with our pruning strategies for sketching assignments and

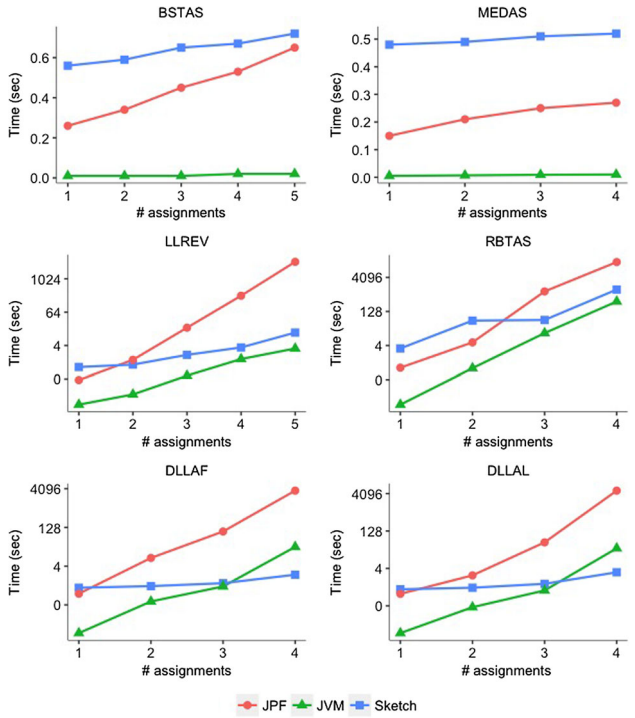

Fig. 7 Compare the performance of sketching assignments

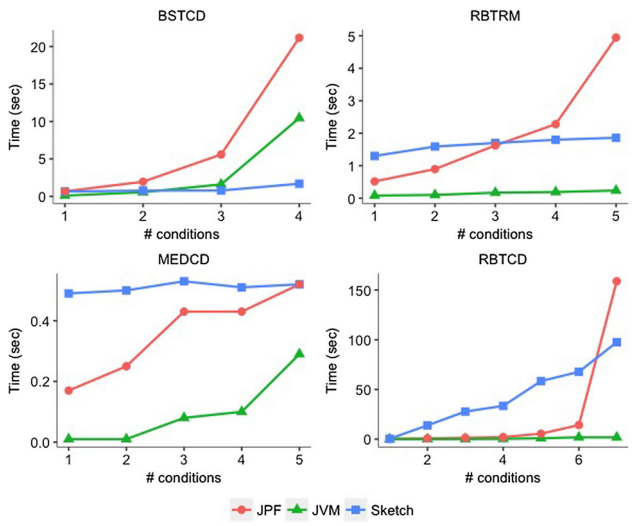

Fig. 8 Compare the performance of sketching conditions

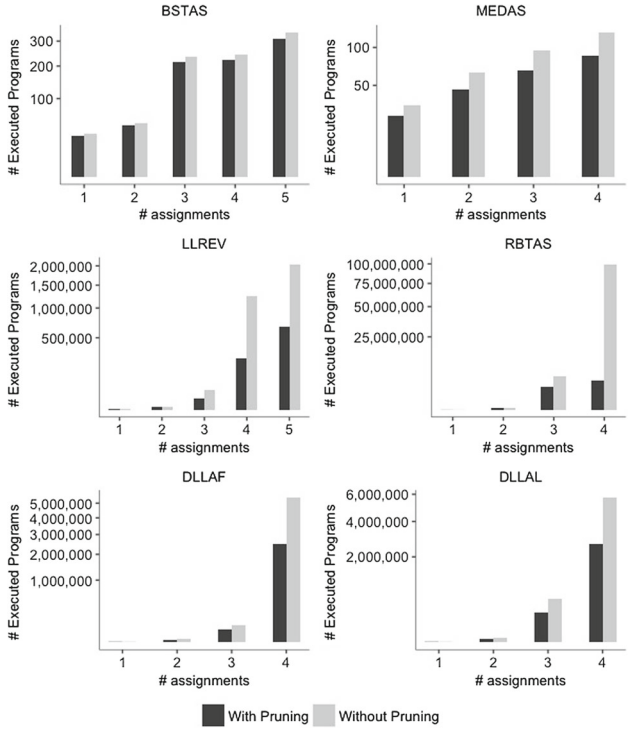

Fig. 9 Compare the pruning efficacy for sketching assignments

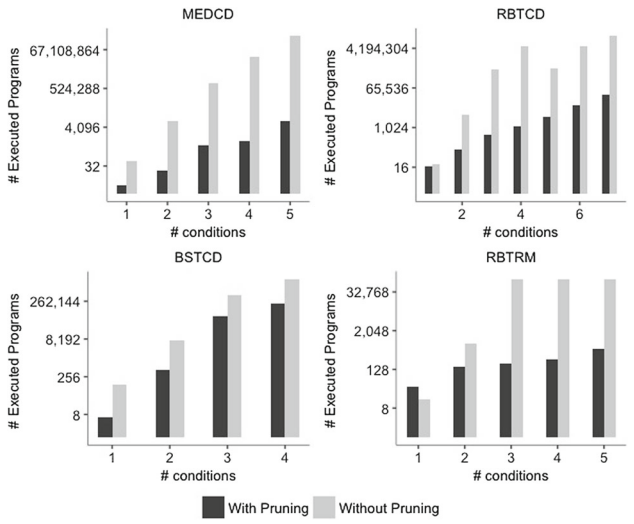

Fig. 10 Compare the pruning efficacy for sketching conditions

conditions is comparable or even sometimes faster than the SAT-based synthesizer.

Comparison of two prototypes In our experiments, stateful search using JAVA PATHFINDER (JPF) is always slower than the dedicated stateless search. JPF is a general purpose model checker that implements a custom JVM to handle all of Java bytecode, including multithreading programs. JPF provides an off-the-shelf backtrack engine, which is a very convenient way to implement a solution for the sketching problem, albeit with sub-optimal performance, it suffers from the performance cost on saving and restoring previous program state. Yet using JPF as the backend for sketching opens the future work possibility to sketch concurrent programs [11] with systematic path exploration. We leave the sketching for multithreading programs as future work.

Efficacy of Pruning Strategies To evaluate whether our pruning strategies can effectively reduce the number of candidates before executing given test suites, we report the number

of executed programs with and without pruning strategies when EDSKETCH finds the first solution that satisfies all test cases.

Figure 9 presents the number of executed programs with and without pruning rules for sketching assignment subjects. The x-axis represents the number of assignments and the y-axis represents the average number of executed programs for sketching a certain number of assignments. The last 4 subjects in Fig. 9 are transformed to square-foot scale for better display. The black bars represent the number of executed programs with pruning rules and the gray ones represent the number of executed programs without pruning rules. As shown in Fig. 9, our pruning rules can effectively prune 7% to 70% with an average of 21% candidates before executing the test suite. For instance, the pruning rules discard 68.4% of candidates for sketching four assignments in the subject LLREV, that is, only around one third of the candidates are actually executed with the test suite and more than two thirds of them are pruned before being executed. The subjects BSTAS and MEDAS have a lower pruning rate (the percentage of candidates being pruned out of all candidates) compared to other subjects, because the sketched assignments in these two subjects are scattered in different if–else conditions and only the rule 1 of the pruning rules can apply to these two subjects.

Figure 10 presents the number of executed programs with and without condition pruning strategy for sketching condition subjects. The y-axis for all subjects in Fig. 10 are transformed to $log2$ scale for better display. Our value grouping strategy can effectively prune an average of 56% of candidates before executing test suites. Note that for sketching one condition in the subject RBTRM, 35 programs are executed on average with value grouping strategy, while only 15 programs are evaluated without value grouping strategy. This result indicates that the value grouping strategy might not always bring in saving when the search space is small, which is different from the pruning rules for assignments.

In summary, EDSKETCH effectively prunes an average of 21% of candidates for assignment sketching and more than half candidates for condition sketching.

Sketching Recursive Procedures Sketching recursive procedures is known to be a challenging problem in program synthesis [13,14,23]. A handful of previous works utilize heuristic search [13], reusable templates [14] and formal specifications [23] to synthesize recursive programs. SKETCH synthesizer has been used as a performance benchmark for these works. We follow the same spirit [13] to compare EDSKETCH with SKETCH synthesizer using recursive tasks borrowed from prior works [13,14,23]: Fibonacci number (integer program), link list insertion (list) and binary search tree insertion (tree).

(A) A sketch of generating fibonacci numbers

```
1. public int fib_sketch(int x) {
2.  if (count++ > EdSketch.RECUR_BOUND)
3.   EdSketch.backtrack();
4.  if (EdSketch.COND(0, new Object[]{x, 1}))
5.   return x;
6. return EdSketch.AOP(0, int.class, new Object[]{
     fib(x-1), fib(x-2)});
7.  }
```

(B) Expected program for the sketch

```
1. public int fib_expect(int x) {
2.  if (x<=1)
3.   return x;
4. return fib(x-1)+fib(x-2);
5.  }
```

(C) Test cases written in JUnit Format

```
@Test
1. public void test() {
2.  assertEquals(fib_sketch(0), fib_expect(0));
3.  assertEquals(fib_sketch(1), fib_expect(1));
4.  assertEquals(fib_sketch(2), fib_expect(2));
5.  assertEquals(fib_sketch(3), fib_expect(3));
6. }
```

Fig. 11 A sketching example of Fibonacci numbers using recursion

Figure 11 presents the subjects of generating Fibonacci numbers using recursion. Similar to prior works on synthesizing recursive procedures [13,14,23], we create a sketch with a condition hole and a hole for the arithmetic operators ({+, -, ×, /}). We use four JUnit test cases to check the first four Fibonacci numbers starting from 0. As described in Sect. 3.3, EDSKETCH instruments the recursive procedure with a bound of the recursive execution to avoid infinite recursion (Fig. 11a). EDSKETCH completes the sketching task in 1 s. Yet using semantic-equivalent sketches and test harnesses, SKETCH synthesizers throws an out-of-memory exception with 2GB memory limit with the same default setting.

For the other two recursive examples for linked list insertion and binary search tree insertion, we introduce an average of one condition holes and four expression holes based on the reference implementation [13,14,23]. Leveraging the test generation tool KORAT [18], we generate a bounded exhaustive test suite up to 3 nodes and convert the generated tests to JUnit format. Both EDSKETCH and SKETCH synthesizer successfully complete the recursive synthesis task within 1 s. Our results demonstrate the EDSKETCH's efficiency at

(A) repOK() method for BST with Reflection

```
1.public boolean repOK() throws Exception {
2. if (root == null)  return size == 0;
3. Set<Node> visited = new HashSet<Node>();
4. visited.add(root);
5. LinkedList<Node> workList=new LinkedList<Node>();
6. workList.add(root);
7. while (!workList.isEmpty()) {
8.  Node cur= workList.removeFirst();
9.  Node exp=(Node)EdSketch.EXP(...);
10. //expect current.left
11. if(Node.class.getField("left").get(exp)!=null){
12.  if (!visited.add(current.left))  return false;
13.  workList.add(current.left);
14. } ...//omit the rest } }
```

(B) Calculate the sum of two integers read from a file

```
1. public int getSum(String file) {
1.  Scanner scan = new Scanner(new File(file));
2.  int a = scan.nextInt();
3.  int b = scan.nextInt();
    //expect a+b
4.  return (int)EdSketch.EXP(...)+(int)EdSketch.EXP(...);
5. }
```

(C) Swap two integers and concatenate them using JNI

```
1. public native String nativeToString(int x, int y);
2. public String swap(int x, int y) {
3.  int tmp = x;
    //expect x = y, y = tmp
4.  EdSketch.BLOCK(...);
5.  String str = nativeToString(x,y);
6.  return str;
7. }
```

Fig. 12 Sketching tasks with advanced features

synthesizing a broad range of programs including recursive procedures.

4.2 Sketching with libraries and advanced features

EDSKETCH is able to explore the actual runtime behaviors in the presence of libraries and advanced features such as reflections, File I/O and native calls. We only use EDSKETCH- JVM to complete the sketch tasks in this section. The SAT-based SKETCH synthesizer and other translation-based synthesizer can hardly tackle these tasks because it's not trivial to translate third-party libraries, reflections and native calls to propositional satisfiability formulas and leverage SAT/SMT solvers to complete the sketches.

Sketching Tasks with Advanced Features EDSKETCH evaluates code with concrete program execution and hence can sketch real-world Java code with advanced features. We illustrate EDSKETCH's ability to sketch program with reflection, I/O and native call using the prototype EDSKETCH- JVM.

Figure 12a presents a program sketch of the repOK() method for Binary Search Tree derived from KORAT's evaluation dataset [18]. At line 11, we try to sketch the object whose left field will be used in the if condition. We provide a bounded exhaustive test suite for up to 3 nodes and execute EDSKETCH on the given test suite from KORAT's evaluation dataset. EDSKETCH completes this task with the variable expression cur. This example also indicates that EDSKETCH can be used in a wider scope of sketching tasks, such as sketching a formal specification written in Java.

Figure 12b shows a getSum() method that reads two variables from a file and outputs their sum. The task sketches the infix expression $a + b$ for the return statement, and EDSKETCH completes this task with three test cases (0, 0), (0, 1), and (2, 1).

Figure 12c presents a sketch task with native calls. EDSKETCH sketches an incomplete swap() method for two integers and returns the string concatenation for the swapped integers in a native method nativeToString() using three test cases (0, 0), (0, 1), and (2, 1).

Sketching Tasks in Presence of Libraries We evaluate EDSKETCH's efficacy of sketching tasks in the presence of libraries on 10 tasks derived from human-written patches and use the original test cases from the open-source projects as the synthesis criteria. These patches are randomly sampled from five open-source projects: JFreeChart (Chart [24]), Closure compiler (Clo- sure [25]), Apache Commons-Lang (Lang [26]), Apache Commons-Math (Math [27]) and JodaTime (Time [28]), which have been used to evaluate a number of prior works [29–31].

Table 2 presents these 10 sketching tasks. Column *Commit* represents the commit number we refer to when generating the sketches. Part of the human-written patches are shown in the next column. The last column represents the sketches written in EDSKETCH syntax to simulate the original patches written by the developers.

For instance, the No. 7 task is derived from the commit c9d786 of the open-source project Apache com- mons-lang (Lang [26]). The version we use has 234 files, 55 thousand lines of Java code, and more than 10 external libraries including maven plug-ins. Derived from the RandomStringUtils class, the sketch task involves an if condition and an assignment. Each invocation of this sketch involves more than 20 methods and 10 JUnit test cases. EDSKETCH generates a complete program that is identical to the original human-written patch in 8.8 s.

4.3 Discussion and threats to validity

In this section, we discuss some alternative setting for EDSKETCH which may affect its sketching efficacy and discuss the threats to validity of our experiments.

Table 2 Sketching tasks in presence of libraries from open-source projects

ID	Commit	Part of the human-written patches	Program sketch written in EDSKETCH syntax
1	Chart	CategoryDataset dataset = this.plot.getDataset(index);	CategoryDataset dataset = this.plot.getDataset(index);
	2265	− if (dataset==null)	
		+ if (dataset!=null)	if (EdSketch.COND(...)) { ... }
		return result;	return result;
2	Chart	− if (endIndex < 0)	
	1083	+ if (endIndex< 0|| endIndex<startIndex)	if (endIndex< 0|| EdSketch.COND(...))
		emptyRange = true;	emptyRange = true;
3	Chart	PathIterator iterator1 = p1.getPathIterator(null);	PathIterator iterator1 = p1.getPathIterator(null);
	1025	− PathIterator iterator2 = p1.getPathIterator(null);	PathIterator iterator2 = ((GeneralPath) EdSketch.EXP
		+ PathIterator iterator2 = p2.getPathIterator(null);	(...)).getPathIterator(null);
4		if (!NodeUtil.isObjectLitKey(n, n.getParent())) {	if (!NodeUtil.isObjectLitKey(n, n.getParent())) {
	Closure	ensureTyped(t, n, STRING_TYPE);	ensureTyped(t, n, STRING_TYPE);
	59a30b	+ else	else
		+ typeable = false;	EdSketch.BLOCK(...);
5	Closure	− if (flags.process_closure_primitives)	
	369282	− options.closurePass = true;	
		+ options.closurePass = flags.closure_primitives;	EdSketch.BLOCK(...);
6		private String getRemainingJSDocLine() {	private String getRemainingJSDocLine() {
	Closure	String result = stream.getRemainingJSDocLine();	String result = stream.getRemainingJSDocLine();
	4fbbc4	+ unreadToken = NO_UNREAD_TOKEN;	EdSketch.BLOCK(...);
		return result;}	return result;}
7		if (start == 0 && end == 0) {	if (start == 0 && end == 0) {
	Lang	+ if (chars != null)	if (EdSketch.COND(...))
	c9d786	+ end = chars.length;	EdSketch.BLOCK(...);
		+ } else {...}	} else {...}
8	Lang	+ if (runningState==STATE_RUNNING) {	if (EdSketch.COND(...)) {
	3ef8a7	stopTime = System.currentTimeMillis();	stopTime = System.currentTimeMillis();
		}	}
9		− qTy(residuals);	
	Math	+ qTy(qtf);	qty((double[]) EdSketch.EXP(...));
	7dadc2	+ tmpVec = objective;	
		+ objective = oldObj;	EdSketch.BLOCK(...);
		+ oldObj = tmpVec;	
10	Time	if (iFieldType == SECONDS_MILLIS || dp > 0) {	if (iFieldType == SECONDS_MILLIS || dp > 0) {
	5d08a1	+ if (valueLong < 0 &&	if (EdSketch.COND(...) && EdSketch.COND(...)) {
		valueLong > − DateTimeConstants.MILLIS_PER_SECOND) {	

We first investigate the order of the test cases and its influence on EDSKETCH's sketching efficacy. We sort the bounded exhaustive test cases based on the number of nodes, execute EDSKETCH with test cases in ascending order and descendent order, and compare the performance time of finding the first solution for all subjects. The result illustrates that the performance is almost the same with test cases reordered (5.1 vs. 5.7 s, Wilcoxon test $p > 0.05$). It might be because the subjects we select are relatively small with a small number of test cases and the time to evaluate test cases is negligible, and thus the prioritization of test cases has little influence on the total performance.

We then investigate the order of selecting left-hand-side and right-hand side expressions for sketching assignments. We have two options to sketch an assignment: select left-hand-side expression first and then select right-hand side expression, or vice versa. We compare the sketching performance with these two options on sketching assignment subjects. Based on our subjects, we find that sketching right-hand side first performs faster than the other option,

especially for the experiments with large search space. For instance, EDSKETCH spends 22.4 s sketching four assignments for the subject DLLAF by selecting right-hand side first, whereas it takes 112.5 s with the other option. Yet this difference is not significant for the experiments with small search space (sketching less than 4 assignments) based on Wilcoxon tests ($p > 0.05$). We report the performance time by selecting right-hand side first in our performance evaluation.

To further investigate the efficacy of our pruning strategies, we execute EDSKETCH to find all solutions that satisfy the test suite and compare the pruning rate, i.e., the percentage of candidates that have been pruned out of all candidates. The result shows that the pruning rate for the first solution is similar to that of all solutions (35% vs 37%), indicating that the efficacy of our pruning strategies is consistent in finding the first solution and all solutions based on our subjects.

External Validity We only compare EDSKETCH with a SAT-based counter-example-guided inductive synthesizer with a small but complex dataset. Our performance comparison results may not extend to other SAT-based synthesizers in a different dataset. Yet Sketch synthesizer has been used as a benchmark for a number of counter-example-guided inductive (CEGIS) synthesizers [4] with reasonable performance. EDSKETCH sketches expressions, conditions and assignments, our idea of execution-driven sketching can be extended to sketch method invocations [22], which has been shown as comparable with the state-of-the-art synthesizers for API sequences, such as SYPET and CODEHINT.

Construct Validity In our performance comparison experiment, we manually transform the Java program to Sketch language, which may have inadvertently introduced behavioral differences. We list all subjects and tests at [16] for cross-validation.

Internal Validity To sketch partial programs, we use bounded exhaustive test suites generated by KORAT [18] based on formal specification, yet these test cases might still lead to plausible sketching results, i.e., passing all test cases but is incorrect from the perspective of users. In our experiment, we manually inspect the first generated solution for each subject to validate their correctness.

5 Related work

Sketch-based synthesis Program synthesis has had numerous successes on synthesizing code in small well-defined domains such as bit-vector logic [32] and data structures [33]. They transform partial programs [1], input-output examples [5] or oracles [32] to decision procedures and leverage SAT/SMT solvers to complete the procedures based on the given specification. These techniques are very efficient in certain domains that have been fully modeled [34]. Sketch-based synthesis [1] is a sub-problem of program synthesis that asks programmers to write a draft program containing missing expressions, and uses counter-example-guided inductive synthesis to complete the holes. JSKETCH brings sketch-based synthesis to Java [6]. Given a partial Java program written in the sketch syntax, JSKETCH translates the Java program to SKETCH synthesizer and transfers the synthesizer's result back to executable Java code. Yet it only supports a limited subset of Java libraries and does not support access control and exceptions. JSKETCH is confined to the limitations of translation-based sketching approaches. EDSKETCH does not translate program sketches and test suites to SAT formulas. It directly executes test cases thus can sketch real-world Java applications that involve advanced language features like reflection. While a primary focus of sketching has been on imperative and functional languages, ASKETCH [35] introduces a test-driven approach for sketching declarative models in ALLOY [36]—a relational first-order logic with transitive closure.

API sequence synthesis Another group of synthesis tools [20,21,37] try to synthesize a composition of API calls, where each method takes some arguments and returns a non-void value. PROSPECTOR [37] and CODEHINT [20] synthesize Java APIs and evaluate code at runtime, thus in theory they could also apply to programs with libraries and advanced features such as reflection. EDSKETCH focuses on synthesizing conditions and blocks of assignments, whereas PROSPECTOR [37], CODEHINT [20] and SYPET [21] focuses on API chain completion. Our idea of execution-driven sketching can be extended to synthesize sequences of API method invocations [22], which has been shown as comparable to SYPET [21] and CODEHINT [20].

Angelic Programming Similar to our approach, angelic programming [19] leverages the non-deterministic backtrack algorithm [38]. Barman et al. [19] embed the angelic choice construct into the Scala programming language and build a parallel backtracking solver to explore the scalability of their backtracking solver. Without any pruning strategies, this approach scales up much faster compared to the SAT-based SKETCH synthesizer. We illustrate that our pruning strategy is efficient and the performance of EDSKETCH on our dataset is comparable or even faster than a SAT-based synthesizer. We also demonstrate that our approach can be extended to real-world Java code that involves reflection and external libraries.

Code completion Code completion refers to the generation of small code snippets. Perelman et al. [39] infer partial expressions using type-directed completion and INSYNTH [34] handles high-order functions and polymorphism using

theorem proving. Yet both are confined to single-statement synthesis. SLANG [40] predicts probabilities of API calls using statistical models based on machine learning. STRATHCONA [41] assists developers for relevant API invocations from similar program contexts. Different from code completion tools based on probabilistic models, EDSKETCH ensures that the generated solution satisfies all test assertions.

Program Repair Program repair [31,42,43] for automated debugging addresses a similar technical problem as program synthesis. Our program isomorphism analysis is similar to AE [44], which determines semantic equivalence with respect to test cases. AE considers duplicate statements and dead code elimination in terms of variable dependency. Our condition value grouping is similar to SPR's notion of abstract condition values [45], which only considers Boolean value true and false as abstract symbols instead of considering all condition candidates. ANGELIX [46] leverages symbolic analysis and constraint solving to generate repairs for real-world C programs. The idea of finding an angelic value is similar to our approach, but it cannot generate new statements while EDSKETCH is able to sketch multiple assignments. Hua et.al [47] reduce the problem of program repair to program sketching by translating the faulty program into a sketch of a correct program, and leverages SAT solvers to generate a complete program that satisfies all test assertions. SKETCHFIX [31] applies execution-driven sketching technique to program repair and automatically generates sketches to fix problematic conditions and expressions. Instead of fixing existing programs, EDSKETCH is designed in the purpose of sketching partial expressions, conditions, and a block of assignment statements.

6 Conclusion

This paper presents a novel *execution-driven sketching* approach that synthesizes Java programs using backtracking search. Our key insight is to introduce effective pruning strategies to reduce the search space for solutions and explore the *actual* program behaviors by executing the given test suite. EDSKETCH can synthesize Java code that may use complex constructs of imperative languages, such as reflection and native calls, and can be applied to recursive procedures and large-scale projects in the presence of libraries. Our experiments show that our approach is comparable or even sometimes faster than a SAT-based synthesizer on our dataset. We believe our execution-driven approach holds a key to practical and scalable solutions to a wide-class of synthesis problems.

Acknowledgements This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-1319688 and CNS-1239498), Shenzhen Peacock Plan (Grant No. KQT D2016112514355531), the Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. JCYJ2017081 7110848086). We thank Mukul Prasad, Allison Sullivan and Kaiyuan Wang for discussion and comments.

References

1. Solar-Lezama, A.: Program sketching. STTT **15**(5–6), 475–495 (2013)
2. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17–23, 2010, pp. 313–326 (2010)
3. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5–10, 2010, pp. 316–329 (2010)
4. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 1–8 (2013)
5. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input–output examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015, pp. 229–239 (2015)
6. Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: JSketch: sketching for Java. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30–September 4, 2015, pp. 934–937 (2015)
7. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. In: ASE 2000, pp. 3–12 (2000)
8. Godefroid, P.: Model checking for programming languages using verisoft. In: POPL 1997, pp. 174–186 (1997)
9. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21–25, 2006, pp. 404–415 (2006)
10. http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html (2017). Accessed 30 Jan 2017
11. Ujma, M., Shafiei, N.: jpf-concurrent: An extension of java pathfinder for java.util.concurrent. CoRR arXiv:1205.0042 (2012)
12. Elkarablieh, B., Khurshid, S.: Juzi: a tool for repairing complex data structures. In: ICSE 2008, pp. 855–858 (2008)
13. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Computer Aided Verification—25th International Conference, CAV 2013, Saint Petersburg, Russia, 13–19 July 2013. Proceedings, pp. 934–950 (2013)
14. Inala, J.P., Polikarpova, N., Qiu, X., Lerner, B.S., Solar-Lezama, A.: Synthesis of recursive ADT transformations from reusable templates. In: Tools and Algorithms for the Construction and Analysis of Systems—23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings, Part I, pp. 247–263 (2017)
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
16. https://github.com/SketchFix/EdSketch-Evaluation (2019). Accessed 1 Jan 2019

17. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: ASE 2005, pp. 273–282 (2005)
18. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. ISSTA **2002**, 123–133 (2002)
19. Bodík, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., Rodarmor, C.: Programming with angelic nondeterminism. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, 17–23 January 2010, pp. 339–352 (2010)
20. Galenson, J., Reames, P., Bodík, R., Hartmann, B., Sen, K.: Codehint: dynamic and interactive synthesis of code snippets. In: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India, May 31–June 07, 2014, pp. 653–663 (2014)
21. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex APIs. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017, pp. 599–612 (2017)
22. Yang, Z., Hua, J., Wang, K., Khurshid, S.: EdSynth: Synthesizing API sequences with conditionals and loops. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Vasteras, Sweden, April 9–13, 2018. IEEE Computer Society (2018)
23. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013, pp. 407–426 (2013)
24. https://sourceforge.net/p/jfreechart/code/HEAD/tree/ (2018). Accessed 30 April 2018
25. https://github.com/google/closure-library (2018). Accessed 30 April 2018
26. https://github.com/apache/commons-lang (2018). Accessed 30 April 2018
27. https://github.com/apache/commons-math (2018). Accessed 30 April 2018
28. https://github.com/JodaOrg/joda-time (2018). Accessed 30 April 2018
29. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: FSE 2014, pp. 654–665 (2014)
30. Just, R., Jalali, D., Ernst, M.D.: Defects4J: a database of existing faults to enable controlled testing studies for java programs. In: ISSTA '14, San Jose, CA, USA, July 21–26, 2014, pp. 437–440 (2014)
31. Hua, J., Zhang, M., Wang, K., Khurshid, S.: Towards practical program repair with on-demand candidate generation. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27–June 3, 2018. ACM (2018)
32. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010, pp. 215–224 (2010)
33. Singh, R., Solar-Lezama, A.: Synthesizing data structure manipulations from storyboards. In: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5–9, 2011, pp. 289–299 (2011)
34. Gvero, T., Kuncak, V., Piskac, R.: Interactive synthesis of code snippets. In: Computer Aided Verification—23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, pp. 418–423 (2011)
35. Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: ASketch: a sketching framework for alloy. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, pp. 916–919 (2018)
36. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
37. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12–15, 2005, pp. 48–61 (2005)
38. Floyd, R.W.: Nondeterministic algorithms. J. ACM **14**, 4 (1967)
39. Perelman, D., Gulwani, S., Ball, T., Grossman, D.: Type-directed completion of partial expressions. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China, June 11–16, 2012, pp. 275–286 (2012)
40. Raychev, V., Vechev, M.T., Yahav, E.: Code completion with statistical language models. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom, June 09–11, 2014, pp. 419–428 (2014)
41. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: 27th International Conference on Software Engineering (ICSE 2005), 15–21 May 2005, St. Louis, Missouri, USA, pp. 117–125 (2005)
42. Malik, M.Z., Ghori, K., Elkarablieh, B., Khurshid, S.: A case for automated debugging using data structure repair. In: ASE, pp. 620–624 (2009)
43. Gopinath, D., Malik, M.Z., Khurshid, S.: Specification-based program repair using SAT. In: TACAS 2011, pp. 173–188 (2011)
44. Weimer, W., Fry, Z.P., Forrest, S.: Leveraging program equivalence for adaptive program repair: models and first results. In: ASE, pp. 356–366 (2013)
45. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: ESEC/FSE, pp. 166–178 (2015)
46. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: ICSE 2016 (2016)
47. Hua, J., Khurshid, S.: A sketching-based approach for debugging using test cases. In: ATVA 2016, pp. 463–478 (2016)