2018 IEEE 11th International Conference on Software Testing, Verification and Validation

EDSYNTH: Synthesizing API Sequences With Conditionals and Loops

Zijiang Yang, Jinru Hua, Kaiyuan Wang, Sarfraz Khurshid
The University of Texas at Austin
Email: {yzjjosh, lisahua, kaiyuanw, khurshid}@utexas.edu

***Abstract*—Good API design enables many clients to effectively use the core functionality implemented by the APIs. For real-world applications however, correctly using the APIs and identifying what methods to use and how to invoke them appropriately can be challenging. Researchers have developed a number of API synthesis approaches that enable a semantically rich form of API completion where the client provides a description of desired functionality, e.g., in the form of test suites, and the automatic tools create method sequences using the desired APIs based on the given correctness criteria (e.g., all given tests pass). However, existing API synthesis approaches are largely limited to creating single basic blocks of code and do not readily handle multiple blocks in the presence of loops (or recursion) and complex test executions. A key issue with handling multiple blocks is the very large space of possible method sequences and their combinations.**

This paper introduces EDSYNTH, an API synthesis approach that explores the sequence spaces *on-demand* during the test execution; that is, the given tests not only provide a validation mechanism – as is common in test-driven API synthesis – but also play a vital role in guiding the space exploration by helping prune much of it. EDSYNTH follows the spirit of recent work on test-execution-driven synthesis and lazily initializes candidates during the execution of given tests where the part of the candidate completion that is actually executed directly determines the generation of future candidates. To further optimize the space exploration, EDSYNTH ranks API candidates based on a set of pre-defined heuristics.

We evaluate EDSYNTH's ability to synthesize complex APIs in the presence of conditional statements, loops and multiple basic blocks. The experimental results show that EDSYNTH is effective at handling synthesis tasks with multiple API sequences in both the conditions and bodies of loops/branches; moreover, when applied to synthesis of straight-line code, EDSYNTH compares well with a state-of-the-art API synthesis tool that only handles straight-line code. The experiments show that EDSYNTH's ranking strategies help reduce synthesis time by 43%.

I. INTRODUCTION

Good API design enables many clients to effectively use the core functionality implemented by the APIs. For real-world applications however, correctly using the APIs in general, and identifying what methods to use and how to invoke them appropriately by providing valid values for all parameters in particular, can be challenging. Prior work shows that even experienced programmers might spend hours trying to understand how to use a simple API [1].

To facilitate writing code against complex APIs, researchers have developed a number of approaches [1]–[7] to support synthesis of code fragments that make appropriate use of the APIs and provide a semantically rich form of API completion. For effective space exploration, some synthesis techniques require users to provide queries in natural language [6], [7] as correctness criteria and synthesize a suitable code fragment that satisfies the given query. Other API synthesis techniques rely on existing code corpus to suggest "statistically similar" code [1], [4], [5] based on the context similarity [2], [8], which assume that a similar API usage already exists in the code base. More recent work on SYPET [3] uses SAT solvers, graph-reachability analysis and Petri nets to synthesize method sequences with respect to given test suites. While existing synthesis approaches for complex APIs are well-founded and handle various practical synthesis problems, previous work is largely limited to creating single basic blocks of code and does not readily handle multiple blocks in the presence of loops (or recursion) and complex tests. A key issue with handling multiple blocks is the very large size of the space of possible method sequences and their combinations.

This paper introduces EDSYNTH, a synthesis approach for creating code fragments that comprise of method sequences for multiple basic blocks against complex APIs with respect to given test suites, which characterize the expected behavior of synthesized fragments. Our key insight is to explore the method sequence spaces *on-demand* – during test execution; therefore, the given tests not only provide a validation mechanism – as is common in test-driven synthesis – but also play a vital role in guiding the space exploration by helping prune much of it. EDSYNTH lazily initializes candidates during the execution of given tests where the part of the candidate completion that is actually executed directly determines the generation of future candidates. To illustrate, consider synthesizing a while-condition and the body of the while-loop, if a test execution returns a false value for while-condition, all combinations of the while-loop body are pruned from the search, which may contains thousands of candidates. Such lazy candidate generation is particularly useful in loops and conditionals, where the part of the candidate that is actually executed directly determines the generation of future candidates. This foundation enables EDSYNTH to naturally support API synthesis in conditionals and loops, as well as synthesizing multiple API blocks without any special handles compared to traditional synthesizing techniques [9]–[15].

Given a partial program (i.e., *sketch* [15]) with unknown method sequence fragments ("holes") and a test suite that characterizes the correctness specification, EDSYNTH compiles a sketch once which may represent thousands of candidates, executes the test suite against the sketch and dynamically selects

0-7695-6388-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICST.2018.00025

candidates to fill in the "holes" of the incomplete program using the EDSKETCH synthesis engine [16]. Whenever the test execution raises a failure (runtime failure or test failure), EDSYNTH backtracks the search immediately and fetches the next candidate until the space of candidates is exhausted or a complete program that satisfies all tests is found. EDSYNTH handles the combinatorial explosion issue with the runtime information of the test execution.

To further expedite the search process, EDSYNTH ranks candidates using a set of heuristics that fit particularly well with on-demand candidate generation. E.g., when EDSYNTH initializes a candidate of the method invocation, if there does not exist any arguments or intermediate values that are in the type of a method parameter, this API is ignored as one of its parameters cannot be initialized based on the Java semantics.

To evaluate EDSYNTH, we first demonstrate its ability to synthesize API sequences in loops and conditionals as well as synthesizing multiple blocks of APIs with 12 API synthesis tasks. EDSYNTH completes all tasks in an average of 4 minutes, while 75% are completed in 1 minute. We then compare EDSYNTH with a state-of-the-art API synthesis tool called SYPET using 38 straight-line tasks in loop-free programs since SYPET only handles straight-line tasks. Within a time limit of 30 minutes, EDSYNTH completes 30 tasks in an average of 67 seconds, while SYPET completes 34 tasks in an average of 53 seconds with advanced graph-reachability analysis. We also discuss our effort to extend an existing API synthesis technique SYPET for conditionals and loops, aiming to illustrate the fundamental hardness of this extension for the traditional test-based API synthesis technique. We finally investigate the efficacy of our prioritization strategies, and the result shows that these strategies effectively reduce 43% of the synthesis time.

This paper makes the following contributions:

- **Synthesizing API sequences with Loops and Conditionals.** We introduce a novel approach to synthesize multiple API sequences at different control points, especially in loops and conditionals where one test may reach multiple basic blocks to synthesize;
- **Lazy Candidate Exploration for API Sequences.** EDSYNTH builds on test-execution-driven synthesis [16] to substantially prune a large number of API sequence candidates and naturally supports API synthesis in loops and conditionals. On-demand candidate generation is particularly helpful where the previously executed candidates in conditions directly determine the generation of next candidates in the loop bodies;
- **Prioritization Strategies.** We introduce ranking strategies to prioritize candidates based on heuristics that work in synergy with on-demand candidate generation. The experimental results show that our strategies can effectively expedite the search process.

II. BACKGROUND AND ILLUSTRATIVE EXAMPLES

In this section, we first define the problem of API synthesis using a straight-line example in loop-free program. To illustrate the limitation of existing API synthesis techniques on loops and conditionals, we present a synthesis task that involves unknown APIs on both the while-condition and the body of the loop. Lastly, we present our preliminary study on the usage of APIs in loops and conditionals using a set of large-scale open source projects. The examples used in this section come from the `Apache-Math` library [17], which consists of 95k lines of code, 541 classes, and 5k methods.

(A) A method that contains an unknown API sequence

```
//Compute the ith  eigenvalue of a matrix
1.Vector2D eigenvalue(RealMatrix arg0, int arg1) {
2. //Synthesize a sequence of APIs
   //for desired functionality
3.}
```

(B) A given test case that specifies the desired behavior

```
1.public static boolean test0() throws Throwable {
2. double[][] mat = new double[][]{{0,-20},{10,10}};
3. RealMatrix matrix = new Array2DRowRealMatrix(mat);
4. Vector2D result = eigenvalue(matrix, 0);
5. Vector2D target = new Vector2D(5,5*Math.sqrt(7));
6. return Math.abs(result.getX()-target.getX())<1e-6
 && Math.abs(result.getY()-target.getY())<1e-6;
7.}
```

(C) Input configuration JSON file for SYPET written by users

```
1. {
2.  "methodName": "eigenvalue",
3.  "srcTypes": [  "RealMatrix", "int"],
4.  "paramNames": ["arg0", "arg1"],
5.  "tgtType": "Vector2D",
6.  "packages": ["org.apache.commons.math3.linear"],
7.  "testPath": "TestSource.java"
8.}
```

(D) Input sketch for EDSYNTH with similar information

```
1.Vector2D eigenvalue(RealMatrix arg0, int arg1) {
2. EdSynth.INVOKE(0) /* Hole ID*/
3. .addParameter(RealMatrix.class, arg0)
4. .addParameter(int.class, arg1)
5. .setReturnType(Vector2D.class)
6. .addPackage("org.apache.commons.math3.linear")
7. .invoke();
8.}
```

(E) A solution generated by EDSYNTH based on the test suite

```
1.EdSynth.INVOKE(0)
2. EigenDecomposition res0 = new
    EigenDecomposition(arg0);
3. double res1 = res0.getImagEigenvalue(arg1);
4. double res2 = res0.getRealEigenvalue(arg1);
5. Vector2D res3 = new Vector2D(res2, res1);
6.return res3;
```

Fig. 1. A synthesis example of straight-line API sequence

A. Problem Definition

API synthesis is an approach that generates a sequence of method invocations to perform a desired functionality based on the given input types, arguments and the output type of the API sequence. Similar to other API synthesis tools [2],

[3], EDSYNTH defines the desired behavior as the satisfaction of all test cases. We present an example of the API synthesis in Figure 1 (A), which tries to create an API sequence to compute the i^{th} eigenvalue of a matrix. Several test cases are provided as correctness criteria and we highlight one of them in Figure 1 (B). This example is brought from the evaluation of SYPET [3], a state-of-the-art API synthesizer that compares favorably with other synthesis tools [2], [18]. The authors find this task by mining the StackOverflow forum where developers post their questions and seek for the answers. This API synthesis task is not trivial and even experienced developers spend hours trying to solve the problem [1], [3]. In this scenario, developers know visible objects, have ideas about the scope of the relevant APIs and can specify the output type they plan to receive as the output of the API sequence. Considering the large number of method invocations provided by the open source projects, the search space of candidates[1] for this example can be as large as 10^{15}.

We show an input configuration file for SYPET in Figure 1 (C). SYPET asks for the visible variables, libraries that specify the scope of the API exploration as well as the output type of the API sequence. Figure 1 (D) presents the input of EDSYNTH for the same synthesis task. In order to complete a desired task, developers use `EdSynth.INVOKE()` methods provided by EDSYNTH to register visible variables (`addParameter()`), define the scope of API exploration (`addPackage()`) and set the target type of the API sequence (`setReturnType()`). This example indicates that the input for EDSYNTH is also required by other techniques.

API synthesis tools return a desired method sequence that satisfies the given test suite. Figure 1 (E) presents a synthesized method sequence from EDSYNTH that pass all test cases. This solution is semantically identical to the result from SYPET using the same test suite. Given that our experimental environment is different from SYPET, we could not fully replicate their experiments. Based on their report, SYPET completes this task in 164 seconds. Using our machine, EDSYNTH synthesizes the desired API sequence in 62 seconds whereas SYPET throws an out of memory exception.

B. Synthesizing API Sequences in Loops and Conditionals

Figure 2 presents a task that synthesizes both the condition and the body of the while-loop using API sequences. Existing techniques [1]–[3], [18] do not handle multiple API blocks in loops or if-branches where one test may reach multiple basic blocks to synthesize. In this example, developers try to use a vector iterator to conduct the element-by-element division for a `RealVector` and returns an `OpenMapRealVector` object. Shown as Figure 2 (A), with the insight that the element-by-element division requires a while-loop, developers write a code skeleton with a while-loop and leave the condition and the body as "holes". To specify the expected behavior of the synthesis task, users provide a suite of test cases and Figure 2 (B) highlights one of them written in JUnit framework. Taking

[1]We describe how we calculate this space in Section IV

(A) A code skeleton that has APIs in both loop condition and body

```
1.public OpenMapRealVector ebeDivide(RealVector v){
2. checkVectorDimensions(v.getDimension());
3. OpenMapRealVector res = new
   OpenMapRealVector(this);
4. Iterator iter = res.entries.iterator();
5. while(/*Hole 0*/(Boolean)EdSynth.INVOKE(0)...){
6.  /*Hole 1*/ EdSynth.INVOKE(1)...;
7. }
8. return res;
9.}
```

(B) A test case written in the JUnit framework

```
  @Test
1.public void testBasicFunctions() {
2. RealVector  v_ebeDivide = v1.ebeDivide(v2);
3. double[] result_ebeDivide = {0.25d, 0.4d, 0.5d};
4. assertClose("compare vect",v_ebeDivide.getData(),
   result_ebeDivide, normTolerance);
5.}
```

(C) A solution generated by EDSYNTH that passes all tests

```
1. EdSynth.INVOKE(0):
2.  iter.hasNext()
3. EdSynth.INVOKE(1):
4.  iter.advance();
5.  Object o1 = iter.key();
6.  Object o2 = divideVal(v, itr);
7.  res.setEntry(o1, o2);
```

Fig. 2. An API synthesis example with multiple APIs in a loop

a program sketch that specifies the high-level user insight and a suite of test cases that define the expected behavior, EDSYNTH completes the low-level implementation details for the synthesis task.

Figure 2 (C) presents a solution generated by EDSYNTH that satisfies all test cases. This solution is semantically identical to the original implementation in the library based on the manual inspection. The search space of candidates for this API synthesis task can be as large as 9.4 billion and EDSYNTH finds this solution in 16 seconds (including compilation and test execution time) after exploring 7.6k candidates.

C. Preliminary Study of APIs in Loops and Conditionals

We discuss our preliminary study on the use of APIs in conditions and bodies of loops and conditionals where one test may reach multiple basic blocks to synthesize. We first select top 10 popular Java projects from Github, following the spirit of prior works [19], [20]. Github is a widely-used repository for open source projects that allows users to mark the projects they are interested in with stars. Leveraging Github API, we construct a dataset of Java projects and sort them by their number of stars. The entire dataset contains 3m lines of code, 31k classes and 251k methods.

Using this small but representative dataset, we calculate the number of conditions with APIs for loops (`while` loops and `for` loops) and if-else branches. Our result shows that 29% of loops and 41% of if-branches use APIs in their conditions. And for these loops and if-branches that have APIs

in their conditions, almost all of them have at least one method invocations in the body of loops or branches, and 59% have more than one consecutive APIs in their body. Although our dataset is relatively small, our preliminary study brings the attention to extend the scope of API synthesis: APIs have been widely adopted in the conditions and bodies of loops and if-branches. To the best of our knowledge, none of the existing API synthesis techniques supports API synthesis for conditions and bodies of loops or if-branches where one test may reach multiple basic blocks to synthesize.

III. Approach

In this section, we explain how we synthesize API sequences during the test execution that naturally supports loops and conditionals. We then discuss our prioritization strategies that aim to expedite the synthesis process.

A. Synthesizing During Test Execution

EdSynth takes a *sketch* with *holes* and a given test suite as input, trying to fill all holes and generate a complete program such that all test cases pass. User specifies holes for unknown APIs (`EdSynth.INVOKE()`) as a regular method invocation, which can be used at any locations including conditions and bodies of loops/if-branches. Therefore, EdSynth does not require additional handling for conditionals and loops where one test may reach multiple basic blocks to synthesize, because the method invocation provided by EdSynth is treated exactly the same as a regular method invocation during the test execution. EdSynth only compiles the user-provided sketch once and generates candidates on demand, i.e., if the condition is evaluated to be false, the candidates for the body of the loops or if-branches will not be generated.

Algorithm 1 illustrates the execution-driven synthesizing procedure [16]. When the test execution reaches a hole at the first time, EdSynth initializes this hole and continues executing the program based on the selected candidate. If this candidate throws a runtime exception or fails in a test assertion, EdSynth breaks from the current execution (line 13) and executes the test again from the beginning. When the program backtracks, EdSynth increments the counter (line 7), thus when the test execution reaches a hole again, EdSynth selects the next candidate for the hole based on the counter, and executes the program with the new candidate. This process stops when a generated candidate passes all tests or the entire search space of candidates is explored.

Constructing API Sequence. Shown as Algorithm 2, when the test execution first reaches the hole of unknown API sequence (`hole.isNotInitialized()`), EdSynth lazily generates the API sequence and invokes them in place using the function `sketchAPISequence()`. The function `sketchAPISequence()` dynamically generates candidates for API sequences when the test execution first reaches the "hole". If this method invocation chain has been initialized, this chain will be used consistently across all test cases (line 5). Yet if this hole is not reached by the test execution, its candidate initialization will not be triggered. In the function

Algorithm 1: Test-Execution-Driven Synthesis [16]

```
   Input  : Partial program P with holes, 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              nextChoice() ;
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(); /* if only one result needed */
```

`sketchAPISequence()`, EdSynth first yields a set of constraints described in the next section of ranking strategies. Based on these constraints (`rule`), EdSynth incrementally inserts a given number of method calls into the sequence (`rule.stmt`). The inserted method is invoked via reflection. If it throws any runtime exception, EdSynth backtracks immediately and re-executes the whole program from the very beginning. Whenever the test execution reaches the hole again, it selects the next candidate (line 15) based on the incremented counter (Algorithm 1 line 7).

Single Method Generation. For each API in the unknown API blocks, EdSynth generates a vector of method candidates based on the generated constraints (`rule`, line 10) and the constraints from the Java syntax. Considering a large amount of classes in the libraries, EdSynth only collects all methods from the given libraries once and reuse them for the entire synthesis process. Shown as the function `sketchAPI()`, for each method invocation, EdSynth selects all type in the search scope, fetches all methods of these types (line 20) and filters out the infeasible methods based on the generated constraints and Java syntax constraints. If there does not exist a given argument or intermediate value with the type of a parameter in the method `m`, this method will not be considered because one of its parameters cannot be initialized. The selected methods are put into a vector of method candidates for this unknown API. EdSynth dynamically selects a candidate method with a non-deterministic `choose()` operator. To complete the selected method candidate (line 23), EdSynth further non-deterministically selects receiver object and parameters of this method from given arguments and intermediate values. Finally, EdSynth returns this generated method call, invokes it using reflection and incrementally inserts more invocations into the API sequence.

Following the same spirit of other API synthesis tools [2], [3], we only consider one method invocation for each

Algorithm 2: API Candidate Generation

```
Input : Program hole hole, Maximum length of the API
        sequence len, Input variables vars
Output: Invocation result result of generated API sequence
1  Function execute () is
2      if hole.isNotInitialized() then
3          return hole.sketchAPISequence();
4      else
5          return hole.getAPISequence().invoke(vars);
6  Function sketchAPISequence () is
7      rule ← getConstraints(len).choose();
8      sequence ← ∅, args ← vars;
9      for int i=0; i<rule.stmt; i++ do
10         call ← sketchAPI(rule, i);
11         try
12             result ← call.invoke(args);
13             args ← args ∪ result;
14         catch Exception e
15             reExeute();
16         sequence ← sequence ∪ call;
17     hole.setAPISequence(sequence);
18     return getReturnVal(rule, args);
19 Function sketchAPI(rule, i) is
       // Generate single API sketch
20     methods ← fetchMethods(hole.getClasses(), rule, i);
21     method ← methods.choose();
       // Complete expressions holes in methods
22     calls ← method.fillParameters(hole.getArgs(i), rule);
23     return calls.choose();
```

statement. E.g., EDSYNTH only considers a method call `iterator.next()` for one statement, but will not consider method chains like `iterator.next().toString()` when it synthesizes a single method invocation. The method chain will be represented as multiple API calls: `Object obj = iterator.next()` and `String str = obj.toString()`. Similar to SYPET, EDSYNTH currently only supports variables as receiver objects and parameters, without considering field dereferences derived from these variables. E.g., field dereferences like `node.val` are not considered while `node` is an element in a linked list and `val` is the integer value of this element. EDSKETCH supports field dereferences sketching [16] and we leave field dereference for EDSYNTH as future work.

Advanced Java Feature Support Many Java libraries use parametric polymorphism. Existing techniques require special models based on static analysis to handle generic types [2], [3]. Without additional modeling, EDSYNTH supports API sequences with generic types by leveraging the runtime information. We present an example of generic type in Figure 3. EDSYNTH also supports reflections and native calls, which can hardly be translated to SAT (refer to EDSKETCH [16]). If users are not clear about the return type of the sequence, they can set the return type as `null`, and EDSYNTH will consider all types as well as the `void` type.

B. Prioritization Strategies.

Once the test execution triggers on-demand candidate exploration, EDSYNTH first generates a set of pre-defined constraints to rank the candidates (line 7). These constraints divide the search space of the API candidates to multiple sub-spaces with more constraints. EDSYNTH generates 4 elements to prioritize the candidates: the number of API invocations in the method sequence ($rule.stmt$), the statement that generates the return object ($rule.rtn$), the maximum distance that intermediate values must be consumed ($rule.vc$), and the maximum number of repetitive APIs in the method sequence ($rule.rep$). We describe the effect of each constraint element as below.

The number of API calls. Intuitively, the search space of the API sequence increases if EDSYNTH wants to synthesize more methods for the "hole". Therefore, we prioritize the candidates with the fewer API invocations and increase the bound of API calls until we reach the given bound from users or the pre-set bound for the number of API calls. For each unknown API hole, this bound is set as 4 by default and is configurable by the end users. Yet EDSYNTH can synthesize multiple API blocks while each of them contains a given bound of APIs. For instance, the fifth task shown in Table I synthesizes an if-condition with one API, a while-loop-condition with one API invocation and 4 APIs (in total of 6 API calls) in the body of loops/if-branches. It can be a while-loop nested with an if-condition or vice versa, and the 4 APIs can be inside the loop but not within the if-branch, inside the if-branch body nested in the loop, or even outside the loop.

The statement that generates the return object. Method invocations appearing later in the execution are more likely to have more information because it can use parameters generated before. Therefore, these methods tend to generate the return value for the API sequence. We prioritize the statements that are called later during the test execution and add a constraint that the selected statement must generate a value with the return type. Figure 1 illustrates this idea that the return value `res3` of the API sequence is the output of the last API call.

Maximum distance that values must be consumed. Similar to other API synthesis tools [1], [3], EDSYNTH assumes that all arguments and intermediate values generated from previous method invocations must be consumed in later API calls. With the insight that arguments and intermediate values should be consumed as fast as possible, we set up a constraint that all values must be used within a given bound of statements. For instance, shown as Figure 1 (E), given a constraint that all intermediate values must be consumed in the distance of two statements, the code at line 5 that initializes a new `Vector2D` object `res3` must consume the intermediate value `res1` generated at line 3. To eliminate duplication, we ensure that there exists at least one value used at the maximum value-consumption distance. If the maximum value-consumption distance is set as 2, at least one value is used at the second next statement, otherwise all objects will be consumed at the next statement, which has been explored with $rule.vc = 1$.

Maximum number of repetitive APIs. With the notion that

developers hardly use the same API for multiple times in a method sequence, we restrict the number of repetitive APIs in a method sequence, and relax this bound by allowing multiple repetitive APIs.

These prioritization strategies are based on the heuristics for API synthesis, which may not always expedite the API synthesis process or lead to the correct answer. Therefore, we conduct a comprehensive evaluation for these strategies in Section IV-C.

IV. EVALUATION

To evaluate EDSYNTH, we first demonstrate our ability to synthesize multiple API blocks where one test may reach multiple basic blocks to synthesize. In particular, we select 12 tasks with loops and conditionals derived from open source projects. To further compare EDSYNTH's performance with existing techniques, we use a benchmark of 38 straight-line tasks in loop-free programs that have been used to evaluate SYPET, a state-of-the-art API synthesis approach, followed by a discussion of extending existing API synthesis techniques for loops and conditionals. Lastly, we investigate the effectiveness of our prioritization strategies based on the same benchmark. We address the following research questions in the evaluation:

- Can EDSYNTH complete multiple API blocks that may be reached by a single test execution, especially for loops and conditionals?
- How well does EDSYNTH perform on synthesizing straight-line code compared to other techniques?
- Can our prioritization strategies effectively expedite the synthesizing process?

All performance experiments are conducted on a MacBook Pro with 2.7 GHz Inter core i5 processor and 8GB memory running OS X version 10.12.4. The maximum heap memory is set as 2 Gigabytes.

A. Synthesizing Multiple Sequences in Loops and Conditionals

To evaluate EDSYNTH's ability to synthesize API blocks that may be reached by the same test execution, especially for the conditions and bodies of loops/if-branches, we select 12 tasks that use 1) at least one API in the condition, 2) at least one method invocation in the body of loops/branches, 3) at least one test case in the test suite should covered both the conditions and the bodies of the loops/branches and 4) every hole of the partial program is covered by at least one test case. We use the original test suites from the open source projects as the correctness criteria, which are usually written in JUnit test framework. These tasks are selected from three open source Java projects that have been widely used in the evaluation of software testing [21], [22]: `JFreeChart` [23], `Apache-Math` [17], and `Closure` compiler [24] for Javascript. We manually create program sketches and introduce holes for API sequences based on the original implementation.

Case Study. Figure 3 presents an example from `Closure` project [24] with generic types and an if-else branch. Using the

(A) A program sketch with generic types and unknown APIs

```
1. public void visit (...) {...
2.  T type = ...;
3.  if ((Boolean) EdSynth.INVOKE(0)
     .addArgument(Object.class, type)...) {
4.   EdSynth.INVOKE(1)
      .addArgument(Object.class, type)
      .setReturnType(null)...;
5.  } else {
6.  EdSynth.INVOKE(2)
     .addArgument(Object.class, type)
     .setReturnType(null)...;
7. }
```

(B) A sample test case of Closure project with special format

```
1. public void testTypedExterns() {
2.  testSets(false, externs, js, output,
     "{alert=[[Foo.prototype]]}");
3. }
```

(C) A solution found by EDSYNTH that satisfies all test cases

```
1. EdSynth.INVOKE(0):
2.  typeSystem.isInvalidatingType(type)
3. EdSynth.INVOKE(1):
4.  prop.invalidate();
5. EdSynth.INVOKE(2):
6.  prop.addTypeToSkip(type);
```

Fig. 3. A synthesis task from open source projects that uses generic types

`visit()` method, developers try to set properties as ineligible, but they are not sure about the implementation details of the branch and they also don't know the return type of the branch, but they are able to provide some high-level insights of the if-else skeleton and ask EDSYNTH to complete the condition of the if-branch, the if-body as well as the else-body. Note that the visible variable `type` is in generic type. Without additional modeling, EDSYNTH simply requires users to define the argument type as `Object.class`. The type of the argument is necessary because if a variable is `null`, EDSYNTH is not able to identify its type using reflection at runtime. Users further set the return type as `null` since they are not sure about the return type, which might be `void`. In particular, the test cases of the `Closure` project are known to be organized in a non-conventional way of using scripts rather than the standard JUnit framework. Figure 3 (B) presents a test case from `Closure` project that covers one branch of this if-else statement, written in this special test format. Our practical test-execution-driven synthesis approach does not have special requirements on the test format and can be applied to large-scale projects in the presence of libraries.

Evaluation Results. Table I lists these 12 subjects, including basic information of the open source projects (lines of code *L*oc and the number of test cases *#*Tests), a brief summary of each synthesis task (Column *Description*) and the number of test cases that reaches the "holes" (Column *#Test*). Column *#Cond* represents the number of APIs in conditions of loops/if-branches. The *2if* indicates that EDSYNTH synthesizes a total of 2 method invocations in the if-condition expressions. It

TABLE I
SUBJECTS WITH MULTIPLE API SEQUENCES IN LOOPS AND CONDITIONALS

Project	ID	Description	#Cond	#Stmt	Space	#Tests	#Run	Time(s)
Chart	1	`createAndAddEntity()`: Created an entity for the axis	3if	3	5.7×10^{13}	1	497	18
Loc: 90k;	2	`addBaseTimelineException()`: Add a segment	1wh	1	2.4k	2	2	12
#Tests:2.2k	3	`addBaseTimelineException()`: Add a segment	1if	3	180.3m	1	23	13
Math	4	`ebeDivide()`: Element-by-element division in a vector	1wh	4	9.4b	1	7.6k	16
Loc: 95k	5	`add()`: Optimized method to add two RealVectors	1if, 1wh	4	565.2b	1	10	1
#Tests: 3.6k	6	`setEntry()`: Set entry in specified row and column	3if	2	9.3×10^{14}	1	529	33
	7	`append()`: append a vector to an existing one	1wh	3	208.8m	1	53	1
	8	`visit()`: sets properties as ineligible	1if	2	9k	57	1	3
Closure	9	`applyCollapses()`: Collapse variable declarations	1if, 1wh	2	12.5b	6	125.0k	220
Loc: 90k	10	`remove()`: Remove this node	1if	3	1.4×10^{11}	10	172.4k	333
#Tests: 7.8k	11	`flattenReferences()`: Flattens to collapsible properties	2if	2	7.0m	23	9	6
	12	`removeVar()`: remove var if it has been coalesced	3if	3	1.8×10^{14}	28	195.1k	2.3k

*if represents the total number of synthesized APIs in if-conditions, and *wh represents the total number of generated methods in while loops.

can be an API chain of two methods in an if-condition, an if-condition and an else-if-condition, or nested if-conditions. Similarly, we use *1wh* to represent a while-condition with one API call. The column *#Stmt* represents the total number of synthesized APIs in the body of the conditions/loops, and these APIs may scatter in different branches. Column *Space* shows the search space of API sequence candidates with respect to the identified solution. Column *#Tests* represents the numbers of provided test cases for the synthesis. Column *#Run* shows the number of executed program candidates when EDSYNTH finds the first solution that passes all test cases. Column *Time* represents the total performance time including the program compilation time and the test execution time when EDSYNTH finds the first solution.

Following other API synthesis techniques, we define the search space of program candidates as $\sum_1^N (m \times v^{ap})^i$ where N is the number of synthesized APIs in the correct solution, m is the total number of methods collected from the input objects, ap represents the average number of parameters for these m methods, and v is the number of given input variables. We define the search space as is because each argument in a method candidate can have a maximum of v options and we incrementally insert more APIs into the sequence searching for a desired solution. Note that it is just an estimated search space because API candidates are dynamically generated based on the previous APIs and constraints, and not all candidates in the search space will be generated based on the on-demand candidate generation.

On average, EDSYNTH explores 41.7k program candidates when it finds the API sequences that satisfy all test assertions. Our on-demand candidate generation approach can substantially prune a significant portion of candidates. In our experiment, EDSYNTH completes partial programs by executing only a very small amount of candidates (less than 0.001%). Regarding the performance time, 75% of synthesis tasks can be done in 1 minute. We also observe that EDSYNTH does not require many test cases to synthesize an API sequence. In particular, some tasks with multiple holes in both the while-condition and the while-body can be synthesized using a single JUnit test case, such as the subject No.4 shown in Figure 2. We manually validate each synthesized program to ensure that it is semantically identical to the original implementation. To the best of our knowledge, none of these 12 tasks can be address by the state-of-the-art API synthesizers.

B. Synthesizing Straight-Line API sequences

To further compare EDSYNTH's synthesizing efficacy with other API synthesizers, we curate a benchmark of 38 synthesis tasks for straight-line APIs in loop-free program. 30 of them are used in the original evaluation of SYPET [3]. The 30 synthesis tasks used in SYPET evaluation are collected from 4 open source projects based on *StackOverflow* online forum and Github repositories, and the corresponding test cases are manually created in an incremental manner until SYPET can find a correct solution with respect to the original implementation. To eliminate the overfitting issue of using SYPET's dataset, we add another 8 tasks derived from the same open source projects. We select SYPET as it is known as the state-of-the-art API synthesizer using advanced graph reachability analysis, which compares favorably with other synthesis tools [2], [18]. As we cannot replicate the experiments with the same machine used in SYPET's evaluation, we execute it under our experiment setting (with smaller memory compared to their machine) and report the comparison result. We set up a time limit as 30 minutes for our experiments, the same as the default setting of SYPET.

Evaluation Results. Table II reports the 38 subjects with brief descriptions. With ranking strategies, EDSYNTH successfully synthesizes 30 subjects with an average of 67 seconds. We manually investigate the tasks that EDSYNTH could not generate a solution within the time limit and find that these outliers usually have a relatively large distance for the intermediate value consumption, thus EDSYNTH de-prioritizes these candidates. For instance, shown as Figure 4 (A), the input argument `arg1` is not consumed until the 6^{th} statements, indicating that the maximum distance of the value consumption is 6. Therefore, EDSYNTH fails to prioritize the correct solution.

TABLE II
STRAIGHT-LINE SUBJECTS

ID	Description	#Tests	SYPET		EDSYNTH					
			#API	Time(s)	#API	Space	With Ranking Time(s)	With Ranking #Run	Without Ranking Time(s)	Without Ranking #Run
1	Compute the pseudo-inverse of a matrix	1	3	41	3	1.7b	6	1.8k	12	5.3k
2	Compute the inner product between two vectors	1	3	24	2	258.6m	2	124	3	134
3	Determine the roots of a polynomial equation	1	3	35	3	23.1t	109	9.0k	380	69.3k
4	Compute the singular value decomposition of a matrix	1	3	24	3	45.5b	3	286	8	993
5	Invert a square matrix	1	3	28	2	12.7m	2	135	3	217
6	Solve a system of linear equations	1	6	115	-	10^{22}	-	146.7k	-	959.7k
7	Compute the outer product between two vectors	1	4	37	2	258.6m	2	117	3	127
8	Predict a value from a sample by linear regression	2	3	244	3	18.1b	517	3.7k	4	3.8k
9	Compute the ith eigenvalue of a matrix	2	-	$\perp$	4	10^{15}	62	211.2k	339	850.3k
10	Scale a rectangle by a given ratio	1	4	36	3	502.0b	314	32.0m	710	18.3m
11	Shear a rectangle and get its bounds	1	4	36	3	502.0b	308	31.9m	133	13.1m
12	Rotate a rectangle about the origin by quadrants	1	4	17	3	32.2b	2	907	3	2.6k
13	Rotate 2-D shape by the specified angle about a point	2	4	33	-	10^{15}	-	19.7m	-	18.2m
14	Perform a translation on a rectangle	1	4	26	3	502.0b	308	32.2m	-	55.0m
15	Intersect a rectangle and an ellipse	1	3	11	2	10.1m	2	40	1	3
16	Compute number of days since a date	2	3	22	3	45.8b	37	80.8k	70	111.6k
17	Subtract two dates considering timezone	3	4	408	3	2.3t	56	150.4k	122	214.2k
18	Determine if a year is a leap year	3	4	68	3	45.8b	24	102.3k	3	1.2k
19	Return the day of a date string	2	3	11	3	2.5t	42	252.9k	43	57.7k
20	Find the number of days of a month in a date string	2	4	83	-	10^{16}	-	38.1m	-	5.6m
21	Find the day of the week of a date string	2	4	44	-	10^{16}	-	19.4m	-	5.4m
22	Compute age given date of birth	2	3	30	3	45.8b	63	158.6k	86	174.1k
23	Compute the offset for a specified line in a document	1	3	19	3	1.7t	5	1.2k	11	4.0k
24	Get a paragraph element given its offset in a document	1	3	24	3	1.7t	4	1k	11	3.8k
25	Obtain the title of a webpage specified by a URL	1	3	110	3	1.1b	35	10.7k	35	19.2k
26	Return doctype of XML document generated by string	1	6	21	-	10^{19}	-	5.4m	-	5.6m
27	Generate an XML element from a string	1	6	24	-	10^{19}	-	3.8m	-	4.8m
28	Read XML document from a file	1	3	14	3	5.5b	4	3.9k	5	2.8k
29	Generate an XML from file and query it using XPath	1	6	78	-	10^{26}	-	3.2m	-	3.3m
30	Get the value of root attribute from a XML file	1	5	17	-	10^{18}	-	13.0m	-	13.8m
31	Check if a point is inside a rectangle	8	5	66	1	3.1k	0.5	1	0.4	1
32	Check if a line segment intersects a rectangle.	8	-	$\perp$	2	63.1m	1	224	2	265
33	Compute number of minutes between two time	8	-	$\perp$	2	175.5m	1	183	1	350
34	Get number of seconds since the midnight of sometime	8	2	15	1	3.5k	1	2	0.7	2
35	Compute the transpose of a matrix	8	3	12	3	1.7b	1	88	1	401
36	Compute the sum of two matrices	8	4	14	4	911.1t	99	3.8k	1.4k	56.5k
37	Compute exclusive or between an area and a rectangle	2	-	-	2	10.1m	3	135	3	136
38	Create an element with given name	3	4	18	4	3.3t	5	6.5k	1	779

$\perp$ represents out of memory. - represents time out after 30 minutes. The first 30 subjects are from the evaluation benchmark of SYPET, 31-38 are 8 subjects derived from the open source projects used in SYPET evaluation, and the rest are small synthesis tasks such as absolute value calculation.

SYPET finds desired API sequences for 34 subjects with an average of 53 seconds. Figure 4 (B) presents a subject that SYPET fails to identify a correct solution within the time limit whereas EDSYNTH detects the solution in 3 seconds. We are not aware of the root cause of this failure as the source code of SYPET is not publicly available, yet we conjecture that the translation to SAT and the advanced graph reachability analysis may cause some impractical issues for open-source projects. In addition, SYPET throws out of memory exception in 3 synthesis tasks, indicating that construction of the large reachability graph using SAT solver can consume a large amount of memory. The memory EDSYNTH uses for API synthesis is linear with respect to the number of method invocations under synthesizing. Regarding the performance of the API synthesis, out of 26 subjects that both tools can generate desired code, EDSYNTH outperforms SYPET in 17 subjects.

Discussion It is possible in principle to enhance existing API synthesis approaches with test partition to handle a broader class of synthesis problems which are handled by EDSYNTH. To illustrate, consider using SYPET to synthesize parts of an "if-else" statement. Intuitively, the synthesis problem can be divided into three subproblems of synthesizing the if-condition, the if-body, and the else-body. Given that a test case can only exclusively execute either the if-block or the else-block, we partition the given test suite and synthesize these two blocks and the if-condition based on two subsets of test cases. Yet this partition requires an enumeration of all combinations for the given test cases, in order to search for an adequate test partition such that SYPET could generate API sequences for both the if-body and the else-body based on these two subsets of tests. If the test partition can successfully generate two method sequences for the if-body and the else-body, we further collect test oracles for the if-condition with

(A) A subject that EDSYNTH times out (No.6)

```
1.public static double[] solveLinear
   (double[][] arg0, double[] arg1) {
2. RealMatrix v1 = MatrixUtils.createMatrix(arg0);
3. RealMatrix v2 = v1.transpose();
4. LUDecomposition v3 = new LUDecomposition(v2);
5. DecompositionSolver v4 = v3.getSolver();
6. RealMatrix v5 = v4.getInverse();
7. double[] v6 = v5.preMultiply(arg1);
8. return v6; }
```

(B) A subject that SYPET times out (No.37)

```
1.public void exclusiveOr(Area area,
   Rectangle2D rect){
2. Area res0 = new Area(rect);
3. area.exclusiveOr(res0); }
```

(C) A subject that ranking strategies fail to expedite the search (No.11)

```
1.public Rectangle2D shear(Rectangle2D arg0,
   double arg1, double arg2){
2. AffineTransform res0 = AffineTransform
   .getShearInstance(arg1, arg2);
3. Shape res1 = res0.createTransformedShape(arg0);
4. Rectangle2D res2 = res1.getBounds2D();
5. return res2; }
```

Fig. 4. Ourlier examples in the evaluation

(A) A code skeleton with branches as the input of extended SYPET

```
1.String classify(int a, int b, int c) {
2. if(condition1(a, b, c)) {
3.  return body1();
4. } else if(condition2(a, b, c)) {
5.  return body2();
6. } else {
7.  return body3();
8. } }
```

(B) Sample test cases that define correct behaviors

```
public static boolean test1() throws Throwable {
 return classify(78, 79, 80).equals("acute");
}
public static boolean test2() throws Throwable {
 return classify(12, 13, 5).equals("right");
}
public static boolean test3() throws Throwable {
 return classify(10, 15, 10).equals("obtuse");
}
```

(C) A solution from extended SYPET

```
1. body3(a,b,c): Library.obtuse(a,b,c);
2. body2(a,b,c): Library.right(a,b,c);
3. body1(a,b,c): Library.acute(a,b,c);
4. condition2(a,b,c): Library.isRight(a,b,c);
5. condition1(a,b,c): Library.isAcute(a,b,c);
```

Fig. 5. An example of extended SYPET that synthesizes multiple APIs in conditionals

respect to the tests, and let SYPET synthesize an API sequence for the if-condition that generates this test partition. Based on this idea of dividing multiple API sequences synthesis tasks to multiple straight-line synthesis problems, we build an extension of SYPET to synthesize chained if-conditions.

To illustrate this extension based on the test partitioning, consider the method `classify()` in Figure 5 that tries to classify the triangle based on its three edges. The method takes the lengths of three edges as input and returns the triangle's classification as either acute, right angled, or obtuse. Figure 5 (A) presents a method skeleton with chained if-conditions. We manually provide more than 10 test cases to ensure 100% branch coverage for this synthesis task: three bodies of the branches and two if-conditions. Figure 5 (C) shows a solution from the extension of SYPET based on the test partition.

Using this extension of the existing straight-line API synthesis technique [25], we observe that it can be very expensive to find an adequate test partition that satisfies constraints for both conditions and bodies of the branch. Given 10 test cases, the possible partitions can be as large as 3^{10}. The test-partition-based extension for multiple API blocks assumes that the synthesized APIs for the conditions cannot change the current program state, otherwise it may fail due to the side effect of the if-condition. EDSYNTH does not suffer from these limitations as it effectively leverages runtime information of the test execution to synthesize the desired method invocations. Moreover, the test-partition-based synthesis for multiple API blocks can fail in synthesizing while-loops and more complex control flows such as nested if-branches whereas EDSYNTH treats the unknown API blocks as a regular Java method invocation that can be used in nested loops and if-branches.

C. Efficacy of Prioritization Strategies

To evaluate if our prioritization strategies expedites the search of the desired APIs, we report the performance time (*Time*) and the number of executed programs (*#Run*) when EDSYNTH finds the first solution that satisfies all test cases. The last 4 columns of Table II lists these results with and without prioritization strategies.

Given a 30-minute time limit, EDSYNTH successfully completes 30 synthesis tasks with prioritization strategies, whereas only 29 can be completed without the ranking strategies and 23 of these tasks take longer to complete the task without the ranking strategies. We manually inspect the subjects where the ranking strategies perform poorly and highlight one example in Figure 4 (C). In this example, the first argument (`arg0`) is used in the second statement while the other two arguments are used in the first statement, which leads to some additional exploration with our ranking strategies. We further use the Spearman test to measure if the performance time and the number of executed candidates are significantly different with and without prioritization strategies. And the result of $p < 0.01$ indicates that our prioritization strategies significantly reduce the performance time and the number of explored candidates.

Threats to Validity. We use test cases as the correctness criteria, which can generate plausible solutions that pass all test cases but are not equivalent to some hypothetical correct ones. We manually inspect the first generated solution for each subject to validate its correctness.

We focus on a new problem of synthesizing multiple API sequences, and we curate a benchmark of tasks with multiple unknown API sequences in both the conditions and the bodies of loops/if-branches to evaluate our approach, yet these tasks may not be real challenges from developers' perspective. We conduct a preliminary study to investigate the usage of APIs in loops and conditionals, and our relatively small dataset used in the study may not extend to larger datasets. We leave the user study as future work and refer to existing studies [1], [2].

We only compare EDSYNTH with a state-of-the-art API synthesizer; thus our comparison result might not extend to other such tools, e.g., CODEHINT [2] and INSYNTH [18]. We note that SYPET has shown to compare favorably with both CODEHINT and INSYNTH using its evaluation subjects, which are also used in our evaluation.

V. RELATED WORK

EDSYNTH is related to a number of works on API synthesis, code completion and program synthesis.

API synthesis. PROSPECTOR [1] is one of the first works that introduces the notion of synthesizing "jungloid code snippets". A *jungloid* is a composition of single-argument methods with a single return type. PROSPECTOR synthesizes method sequence based on the *jungloids* mined from existing code corpus. Similar to PROSPECTOR, CODEHINT [2] leverages empirical statistic models to prioritize runtime recommendation for the API completion. It allows users to set up breakpoints in Eclipse IDE and rollbacks to the previous state whenever the selected candidate encounters a runtime exception or test failure. In contrast to the tools with expensive state restoration, EDSYNTH simply backtracks till the beginning of the program and re-executes the next candidate. SYPET [3] further extends PROSPECTOR by introducing Petri-net for reachability analysis based on SAT solvers. We believe their reachability analysis is complementary to ours, yet in this paper, we show that using a relatively simple test-execution-driven approach for API synthesis, EDSYNTH compares generally well with SYPET and further supports multiple API sequences where one test may reach multiple basic blocks to synthesis, especially for the loops and conditionals where both conditions and bodies of loops/if-branches use APIs.

Code Completion. A series of other tools [4], [5], [26], [27] mine API usage pattern from the code repositories and use empirical probabilistic models to guide the search towards the methods that are more often used in practice. In particular, SLANG [28] predicts probabilities of API calls using statistical models based on machine learning, while STRATHCONA [8] assists developers in finding relevant API invocations with similar program contexts. Different from these techniques that train offline data corpus for method invocation completion, EDSYNTH leverages runtime information to substantially prune a large portion of candidates. We envision that empirical statistical models can be helpful to prioritize the sequences.

Program Synthesis. Program synthesis has achieved many success on synthesizing code in small well-defined domains such as bit-vector logic [11] and data structures [29], [30] based on test cases [9] or specifications [10], [12]. These tools transform partial programs [15], input-output examples [31], [32] or oracles [11], [33] to decision procedures and SMT solvers, which have shown as very efficient in certain domains that have been fully modeled [18], [34]. Other synthesis tools try to generate small code snippets. Perelman et al. [35] infer partial expressions using type-directed completion and INSYNTH [18] handles high-order functions and polymorphism using theorem proving. Yet both of them are confined to generating a single statement rather than the method sequence. Recent works leverages natural language queries [6], [7], [36] to infer expressions and simple method invocations, whereas EDSYNTH could synthesize code in a much large scope: multiple method invocations in loops and conditionals.

Our idea of exploiting the programmers' expertise to provide high-level insight of program skeletons share the same spirit with the sketching-based synthesis [37]. The SKETCH system [15] asks programmers to write a program sketch with "holes", and uses counter-example-guided inductive synthesis to complete the holes. JSKETCH brings the sketch-based synthesis to Java [38]. Given a partial Java program written in the sketch syntax, JSKETCH translates the Java program to SAT-based sketch synthesizer and transfers the synthesizer result back to executable Java code. Yet JSKETCH supports a limited number of libraries due to the difficulty of transforming libraries to SAT. Sullivan's doctoral dissertation [39] introduces a test-driven approach for sketching declarative models in Alloy [40]. Our synthesis approach based on the test execution is also similar to other test-driven synthesis techniques [9], [34], [41], yet we focus on a more complex scenario of synthesizing Java method invocations with loops and conditionals in open source projects.

VI. CONCLUSION

We introduced EDSYNTH, an approach that synthesizes method invocation sequences in programs with loops and conditionals, where the fragments may be scattered across multiple basic blocks. EDSYNTH utilizes runtime information to substantially prune a large number of candidate fragments using on-demand candidate generation. To further expedite the search, EDSYNTH leverages a set of ranking strategies that apply in synergy with on-demand candidate generation.

We evaluated EDSYNTH using a suite of API synthesis tasks with loops and conditionals, and compared it with a state-of-the-art tool for straight-line API sequences. The experimental results showed the effectiveness of EDSYNTH at handling synthesis tasks with loops and conditionals where one test may reach multiple basic blocks to synthesize. We believe execution-driven synthesis with on-demand candidate generation holds a key to a practical and scalable approach for synthesis in a variety of domains.

ACKNOWLEDGMENT

We thank Isil Dillig for valuable comments. This research was partially supported by the US National Science Foundation under Grant Nos. CCF-1704790 and CCF-1718903.

REFERENCES

[1] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *PLDI 2005*, 2005, pp. 48–61.

[2] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen, "CodeHint: dynamic and interactive synthesis of code snippets," in *ICSE 2014*, 2014, pp. 653–663.

[3] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *POPL 2017*, 2017, pp. 599–612.

[4] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 37:1–37:30, 2013.

[5] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), USA*, 2007, pp. 204–213.

[6] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A search engine for java using free-form queries," in *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, 2009, pp. 385–400.

[7] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, 2015, pp. 416–432.

[8] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *ICSE 2005*, 2005, pp. 117–125.

[9] D. Perelman, S. Gulwani, D. Grossman, and P. Provost, "Test-driven synthesis," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, 2014*, 2014, pp. 408–418.

[10] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD 2013*, 2013, pp. 1–8.

[11] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE 2010*, 2010, pp. 215–224.

[12] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Complete functional synthesis," in *PLDI 2010*, 2010, pp. 316–329.

[13] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 90–121, Jan. 1980.

[14] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," 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*, 2017, pp. 319–336.

[15] A. Solar-Lezama, "Program sketching," *STTT*, vol. 15, no. 5-6, pp. 475–495, 2013.

[16] J. Hua and S. Khurshid, "Edsketch: Execution-driven sketching for java," in *International Symposium on Model Checking of Software, SPIN 2017*, 2017, pp. 162–171.

[17] "Apache math project," http://commons.apache.org/proper/commons-math/, 2017, accessed: 2017-04-30.

[18] T. Gvero, V. Kuncak, and R. Piskac, "Interactive synthesis of code snippets," in *CAV 2011*, 2011, pp. 418–423.

[19] M. Allamanis and C. A. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 472–483.

[20] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 38–49.

[21] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset," vol. 22, no. 4, 2017, pp. 1936–1964.

[22] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA 2014*, 2014, pp. 437–440.

[23] http://www.jfree.org/jfreechart/, 2017, accessed: 2017-04-30.

[24] "Closure compiler," https://developers.google.com/closure/compiler/, 2017, accessed: 2017-04-30.

[25] Z. Yang, "An experimental evaluation and possible extensions of sypet," in *Master Thesis, The University of Texas at Austin*, 2017.

[26] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR 2013*, 2013, pp. 319–328.

[27] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of AST nodes to study actual and potential usage of java language features," in *36th International Conference on Software Engineering, ICSE 2014*, 2014, pp. 779–790.

[28] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI 2014*, 2014, pp. 419–428.

[29] R. Singh and A. Solar-Lezama, "Synthesizing data structure manipulations from storyboards," in *FSE 2011*, 2011, pp. 289–299.

[30] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard, "Inference and enforcement of data structure consistency specifications," in *ISSTA 2006*, 2006, pp. 233–244.

[31] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *PLDI 2015*, 2015, pp. 229–239.

[32] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *PLDI '13*, 2013, pp. 15–26.

[33] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, 2011, pp. 62–73.

[34] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *POPL 2010*, 2010, pp. 313–326.

[35] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *PLDI 2012*, 2012, pp. 275–286.

[36] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, 2016, pp. 345–356.

[37] A. Solar-Lezama, "The sketching approach to program synthesis," in *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009*, 2009, pp. 4–13.

[38] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama, "JSketch: sketching for Java," in *ESEC/FSE 2015*, 2015, pp. 934–937.

[39] A. Sullivan, "Automated testing and sketching of alloy models," Ph.D. dissertation, University of Texas at Austin, 2017.

[40] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[41] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Practical program repair with on-demand candidate generation," in *40th International Conference on Software Engineering, ICSE 2018*, 2018.