JSketch: Sketching for Java

Jinseong Jeon* Xiaokang Qiu† Jeffrey S. Foster* Armando Solar-Lezama†
*University of Maryland, College Park, USA †Massachusetts Institute of Technology, USA
jsjeon@cs.umd.edu xkqiu@csail.mit.edu jfoster@cs.umd.edu asolar@csail.mit.edu

ABSTRACT

Sketch-based synthesis, epitomized by the Sketch tool, lets developers synthesize software starting from a *partial program*, also called a *sketch* or *template*. This paper presents JSketch, a tool that brings sketch-based synthesis to Java. JSketch's input is a partial Java program that may include *holes*, which are unknown constants, *expression generators*, which range over sets of expressions, and *class generators*, which are partial classes. JSketch then translates the synthesis problem into a Sketch problem; this translation is complex because Sketch is not object-oriented. Finally, JSketch synthesizes an executable Java program by interpreting the output of Sketch.

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: Program Synthesis

General Terms

Design, Languages.

Keywords

Program Synthesis, Programming by Example, Java, Sketch.

1. INTRODUCTION

Program synthesis aims to automate the development of complex pieces of code. Deriving programs completely from scratch given only a declarative specification is very challenging for all but the simplest algorithms, but recent work has shown that the problem can be made tractable by starting from a partial program—referred to in the literature as a sketch [8], scaffold [10] or template—that constrains the space of possible programs the synthesizer needs to consider. This approach to synthesis has proven useful in a variety of domains including program inversion [9], development of concurrent data-structures [7], and automated tutoring [4].

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy

http://dx.doi.org/10.1145/2786805.2803189

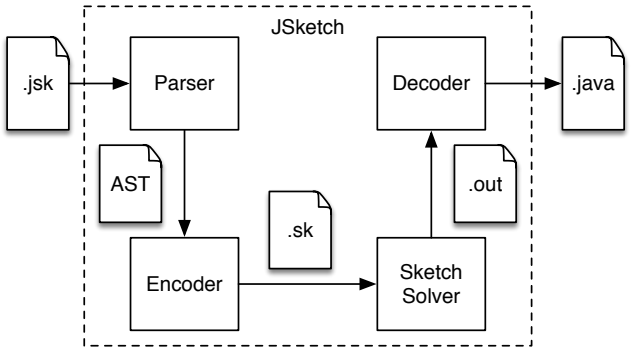

Figure 1: JSketch Overview.

This paper presents JSketch, a tool that makes sketch-based synthesis directly available to Java programmers. JSketch is built as a frontend to the Sketch synthesis system, a mature synthesis tool based on a simple imperative language that can generate C code [8]. JSketch allows Java programmers to use many of the Sketch's synthesis features, such as the ability to write code with unknown constants (*holes* written ??) and unknown expressions described by a *generator* (written $\{|\ e^*\ |\}$). In addition, JSketch provides a new synthesis feature—a class-level generator—specifically tailored for object-oriented programs. Section 2 walks through a running example of JSketch.

As illustrated in Figure 1, JSketch compiles a Java program with unknowns to a partial program in the Sketch language and then maps the result of Sketch synthesis back to Java. The translation to Sketch is challenging because Sketch is not object oriented, so the translator must model the complex object-oriented features in Java—such as inheritance, method overloading and overriding, anonymous/inner classes—in terms of the features available in Sketch. Section 3 briefly explains several technical challenges addressed in JSketch. Section 4 describes our experience with JSketch. JSketch is available at `http://github.com/plum-umd/java-sketch/`.

2. OVERVIEW

We begin our presentation with two examples showing JSketch's key features and usage.

Basics. The input to JSketch is an ordinary Java program that may also contain unknowns to be synthesized. There are two kinds of unknowns: *holes*, written ??, represent unknown integers and booleans, and *generators*, written $\{|\ e^*\ |\}$, range over a list of expressions. For example, con-

sider the following Java sketch, similar to an example from the SKETCH manual [6]:

```
1 class SimpleMath {
2     static int mult2(int x) { return (?? * {| x , 0 |}); }
3 }
```

Here we have provided a template for the implementation of method mult2: The method returns the product of a hole and either x or 0. Notice that even this very simple sketch has 2^{33} possible instantiations. To specify the desired solution, we provide a *harness* with assertions about the mult2 method:

```
4 class Test {
5     harness static void test() { assert(SimpleMath.mult2(3) == 6); }
6 }
```

Now we can run JSKETCH on the sketch and harness.

```
$ ./jsk.sh SimpleMath.java Test.java
```

The result is a valid Java source file in which holes and generators have been replaced with the appropriate code.

```
$ cat result/java/SimpleMath.java
class SimpleMath { ...
static public int mult2 (int x) {
return 2 * x;
} }
```

Finite Automata. Now consider a harder problem: suppose we want to synthesize a finite automaton given sample accepting and rejecting inputs.[1] There are many possible design choices for finite automata in an object-oriented language, and we will opt for one of the more efficient ones: the current automaton state will simply be an integer, and a series of conditionals will encode the transition function.

Figure 2a shows our automaton sketch. The input to the automaton will be a sequence of Tokens, which have a getId method returning an integer (line 8). An Automaton is a class—ignore the generator keyword for the moment—with fields for the current state (line 9) and the number of states (line 10). Notice these fields are initialized to holes, and thus the automaton can start from any arbitrary state and have an arbitrary yet minimal number of states (restricted by SKETCH's minimize function on line 11). The class includes a transition function that asserts that the current state is in-bounds (line 13) and updates state according to the current state and the input Token's value (retrieved on line 14).

Here we face a challenge: we do not know the number of automaton states or tokens, so we cannot bound the number of transitions. To solve this problem, we use a feature inherited from SKETCH: the term minrepeat { e } expands to the minimum length sequence of e's that satisfy the harness. Here, the body of minrepeat (line 16) is a conditional encoding an arbitrary transition—if the guard matches the current state and input token, then the state is updated and the method returns. Thus, the transition method will be synthesized to include however many transitions are necessary.

Finally, the Automaton class has methods transitions and accept; the first performs multiple transitions based on a sequence of input tokens, and the second one determines whether the automaton is in an accepting state. Notice that the inequality (line 21) means that states 0 up to some bound will be accepting; this is fully general because the exact state numbering does not matter, so the synthesizer can choose the accepting states to follow this pattern.

[1]Of course, there are many better ways to construct finite automata—this example is only for expository purposes.

```
7  interface Token{ public int getId (); }
8  generator class Automaton {
9      private int state = ??;
10     static int num_state = ??;
11     harness static void min_num_state() { minimize(num_state); }
12     public void transition (Token t) {
13         assert 0 ≤ state && state < num_state;
14         int id = t.getId ();
15         minrepeat {
16             if (state == ?? && id == ??) { state = ??; return; }
17     } }
18     public void transitions ( Iterator <Token> it) {
19         while ( it .hasNext()) { transition ( it .next ()); }
20     }
21     public boolean accept() { return state ≤ ??; }
22 }
```

(a) Automaton sketch.

```
23 class DBConnection {
24     class Monitor extends Automaton {
25         final static Token OPEN =
26             new Token() { public int getId () { return 1; } };
27         final static Token CLOSE =
28             new Token() { public int getId () { return 2; } };
29         public Monitor() { }
30     }
31     Monitor m;
32     public DBConnection() { m = new Monitor(); }
33     public boolean isErroneous() { return ! m.accept(); }
34     public void open() { m.transition (Monitor.OPEN); }
35     public void close() { m.transition (Monitor.CLOSE); }
36 }
37 class CADsR extends Automaton { ...
38     public boolean accept(String str) {
39         state = init_state_backup;
40         transitions ( convertToIterator ( str ));
41         return accept();
42 } }
```

(b) Code using Automaton sketch.

Figure 2: Finite automata with JSKETCH.

Class Generators. In addition to basic SKETCH generators like we saw in the mult2 example, JSKETCH also supports *class generators*, which allow the same class to be instantiated differently in different superclass contexts. In Figure 2a, the generator annotation on line 8 indicates that Automaton is such a class. (Class generators are analogous to the the function generators introduced by SKETCH [6].)

Figure 2b shows two classes that inherit from Automaton. The first class, DBConnection, has an inner class Monitor that inherits from Automaton. The Monitor class defines two tokens, OPEN and CLOSE, whose ids are 1 and 2, respectively. The outer class has a Monitor instance m that transitions when the database is opened (line 34) and when the database is closed (line 35). The goal is to synthesize m such that it acts as an inline reference monitor to check that the database is never opened or closed twice in a row, and is only closed after

```
43 class TestDBConnection {
44     harness static void scenario_good() {
45         DBConnection conn = new DBConnection();
46         assert ! conn.isErrorneous();
47         conn.open(); assert ! conn.isErroneous();
48         conn.close(); assert ! conn.isErroneous(); }
49     // bad: opening more than once
50     harness static void scenario_bad1() {
51         DBConnection conn = new DBConnection();
52         conn.open(); conn.open(); assert conn.isErroneous(); }
53     // bad: closing more than once
54     harness static void scenario_bad2() {
55         DBConnection conn = new DBConnection();
56         conn.open();
57         conn.close(); conn.close(); assert conn.isErroneous();
58 }   }
59 class TestCADsR {
60     // Lisp−style identifier : c(a|d)+r
61     harness static void examples() {
62         CADsR a = new CADsR();
63         assert ! a.accept("c");   assert ! a.accept("cr");
64         assert a.accept("car");   assert a.accept("cdr");
65         assert a.accept("caar");  assert a.accept("cadr");
66         assert a.accept("cdar");  assert a.accept("cddr");
67 }   }
```

Figure 3: Automata use cases.

```
68 class Automaton1 {
69   int state = 0; static int num_state = 3;
70   public void transition (Token t) { ...
71     assert 0 ≤ state && state < 3;
72     if (state == 0 && id == 1) { state = 1; return; } // open@
73     if (state == 1 && id == 1) { state = 2; return; } // open 2x
74     if (state == 1 && id == 2) { state = 0; return; } // ( init )@
75     if (state == 0 && id == 2) { state = 2; return; } // close 2x
76   }
77   public boolean accept() { return state ≤ 1; } ...
78 }
79 class DBConnection{ class Monitor extends Automaton1 { ... } ...}
80 class Automaton2 {
81   int state = 0; static int num_state = 3;
82   public void transition (Token t) { ...
83     assert 0 ≤ state && state < 3;
84     if (state == 0 && id == 99) { state = 1; return; } // c
85     if (state == 1 && id == 97) { state = 2; return; } // ca
86     if (state == 1 && id == 100) { state = 2; return; } // cd
87     if (state == 2 && id == 114) { state = 0; return; } // c(a|d)+r@
88   }
89   public boolean accept() { return state ≤ 0; } ...
90 }
91 class CADsR extends Automaton2 { ... }
```

Figure 4: JSketch Output (partial).

it is opened. The harnesses in TestDBConnection in Figure 3 describe both good and bad behaviors.

The second class in Figure 2b, CADsR, adds a new (overloaded) accept(String) method that converts the input String to a token iterator (details omitted for brevity), transitions according to that iterator, and then returns whether the string is accepted. The goal is to synthesize an automaton that recognizes `c(a|d)+r`. The corresponding harness TestCADsR.examples() in Figure 3 constructs a CADsR instance and makes various assertions about its behavior. Notice that this example relies critically on class generators, since Monitor and CADsR must encode different automata.

Output. Figure 4 shows the output produced by running JSketch on the code in Figures 2 and 3. We see that the generator was instantiated as Automaton1, inherited by DBConnection.Monitor, and Automaton2, inherited by CADsR. Both automata are equivalent to what we would expect for these languages. Two things were critical for achieving this result: minimizing the number of states (line 11) and having sufficient harnesses (Figure 3). (Details can be found in a companion technical report [1].)

We experimented further with CADsR to see how changing the sketch and harness affects the output. First, we tried a smaller harness, i.e., fewer examples. In this case, the synthesized automaton covers all the examples but not the full language. For example, if we omit the four-letter inputs in Figure 3 the resulting automaton only accepts three-letter inputs. Whereas going to four-letter inputs constrains the problem enough for JSketch to find the full solution. Second, if we omit state minimization (line 11), then the synthesizer chooses large, widely separated indexes for states, and it also includes redundant states (that could be merged with a textbook state minimization algorithm). Third, if we manually bound the number of states to be too small (e.g., manually set num_state to 2), the synthesizer runs for more than half an hour and then fails, since there is no solution.

Of these cases, the last two are relatively easy to deal with since the failure is obvious, but the first one—knowing that a synthesis problem is underconstrained—is an open research challenge. However, one good feature of synthesis is that, if we do find cases that are not handled by the current implementation, we can simply add those cases and resynthesize rather than having to manually fix the code (which could be quite difficult and/or introduce its own bugs). Moreover, minimization—trying to ensure the output program is small—seems to be a good heuristic to avoid overfitting.

3. IMPLEMENTATION

We implemented JSketch as a series of Python scripts that invoke Sketch as a subroutine. JSketch comprises roughly 5.7K lines of code, excluding the parser. There are a number of challenges in the implementation of JSketch; due to space limitations we discuss only the major ones.

Class hierarchy. The first issue is that Sketch's language is not object-oriented. To solve this problem, JSketch follows a similar approach to [4] and encodes objects with a new type V_Object, defined as a struct containing all possible fields plus an integer identifier for the class. More precisely, if $C_1, \ldots, C_m$ are all classes in the program, then we define:

```
92 struct V_Object {
93   int class_id; fields-from-C_1 ... fields-from-C_m
94 }
```

where each C_i gets its own unique id.

JSketch also assigns every method a unique id, and it creates various constant arrays that record type information. For a method id m, we set belongsTo[m] to be its class id; argNum[m] to be its number of arguments; and argType[m][i] to be the type of its i-th argument. We model the inher-

itance hierarchy using a two-dimensional array subcls such that subcls[i][j] is true if class i is a subclass of class j.

Encoding names. When we translate the class hierarchy into JSketch, we also flatten the namespace. During this process we must not conflate overridden or overloaded method names, or inner classes. Thus, we name inner classes as *Inner_Outer*, and we differentiate anonymous classes using distinct numbers, e.g., *Cls*_1. To support method overriding and overloading, methods are named *Mtd_Cls_Params*, where *Mtd* is the name of the method, *Cls* is the name of the class in which it is declared, and *Params* is the list of parameter types. For example, in the finite automaton example, CADsR inherits method transition from Automaton2 (the second variant of the class generator), hence the method is named transition_Automaton2_Token(V_Object self, V_Object t) in Sketch. The first parameter represents the callee of the method.

Dynamic dispatch. For each method m, we simulate dynamic dispatch in Sketch by introducing a function that dispatches based on the class_id field of the callee:

```
95  void dyn_dispatch_m(V_Object self, ...) {
96    int cid = self.class_id;
97    if (cid == R0_id) return m_R0_P(self, ...);
98    if (cid == R1_id) return m_R1_P(self, ...);
99    ...
100   return;
101 }
```

Note that if m is static, the self argument is omitted.

Java libraries. To perform synthesis, we need Sketch equivalents of any Java standard libraries used in the input sketch. Currently, JSketch supports the following collections and APIs: ArrayDeque, Iterator, LinkedList, List, Map, Queue, Stack, TreeMap, CharSequence, String, StringBuilder, and StringBuffer. Library classes are implemented using a combination of translation of the original source using JSketch and manual coding, to handle native methods or cases when efficiency is an issue. Note that several of these classes include generics (e.g., List), which is naturally handled because the all objects are uniformly represented as V_Object.

Limitations and unsupported features. As Java is a very large language, JSketch currently only supports a core subset of Java. We leave several features of Java to the future versions of JSketch, including packages, access control, exceptions, and concurrency. Additionally, JSketch assumes the input sketch is type correct, meaning the standard parts of the program are type correct, holes are used either as integers or booleans, and expression generators are type correct. This assumption is necessary because, although Sketch itself includes static type checking, distinctions between different object types are lost by collapsing them all into V_Object.

Using Sketch. We translate JSketch file, which is composed of the user-given template and examples, as well as supporting libraries to .sk files as input to Sketch. For example, SimpleMath from Section 2 translates to

```
102 int e_h1 = ??;
103 int mult2_SimpleMath_int(int x) { return e_h1 * {| x | 0 |}; }
104 harness void test_Test() { assert mult2_SimpleMath_int(3) == 6; }
```

Details of how Sketch works can be found elsewhere [5, 6].

After solving the synthesis problem, JSketch then unparses these same Java files, but with unknowns resolved according to the Sketch synthesis results.

4. EXPERIENCE WITH JSketch

We developed JSketch as part of the development of another tool, Pasket [2], which aims to construct *framework models*, e.g. mock classes that implement key functionality of a framework but in a way that is much simpler than the actual framework code and is more amenable to static analysis. Pasket takes as input a log of the interaction between the real framework and a test application, together with descriptions of the API of the framework and design patterns the framework uses. Pasket uses these inputs to automatically generate an input to JSketch which is then responsible for actually synthesizing the models. Through Pasket, we have used JSketch to synthesize models of key functionality from the Swing and Android frameworks. The largest JSketch inputs generated by Pasket contain 117 classes and 4,372 lines of code, and solve in about two minutes despite having over $73^{18} \times 164^{28}$ possible choices; this is possible thanks to a new synthesis algorithm called Adaptive Concretization [3] that is available in Sketch and was also developed as part of this work.

5. ACKNOWLEDGMENTS

This research was supported in part by NSF CCF-1139021, -1139056, -1161775, and the partnership between UMIACS and the Laboratory for Telecommunication Sciences.

6. REFERENCES

[1] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama. JSketch: Sketching for Java. *CoRR*, abs/1507.03577, 2015.

[2] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama. Synthesizing Framework Models for Symbolic Execution. Unpublished manuscript, 2015.

[3] J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster. Adaptive Concretization for Parallel Program Synthesis. In *CAV*, July 2015.

[4] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. In *PLDI*, pages 15–26, 2013.

[5] A. Solar-Lezama. Program sketching. *Int. J. STTT*, 15(5-6):475–495, 2013.

[6] A. Solar-Lezama. *The Sketch Programmers Manual*, 2015. Version 1.6.7.

[7] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.

[8] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[9] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-Based Inductive Synthesis for Program Inversion. In *PLDI*, pages 492–503, June 2011.

[10] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.