JSKETCH: Sketching for Java

Jinseong Jeon[†] Xiaokang Qiu[‡] Jeffrey S. Foster[†] Armando Solar-Lezama[‡]

[†]University of Maryland, College Park [‡]MIT CSAIL

{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; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Assertions, Specification techniques

General Terms Design, Languages.

Keywords Program Synthesis, Programming by Example, Java, SKETCH, Input-output Examples.

1. Introduction

Program synthesis [5, 6] is an attractive programming paradigm that 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 [11], scaffold [13] 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 [14], program deobfuscation [4], development of concurrent data-structures [12] and even automated tutoring [8].

This paper presents JSKETCH, a tool that makes sketch-based synthesis directly available to Java programmers. JSKETCH is built as a frontend on top of the SKETCH synthesis system, a mature synthesis tool based on a simple imperative language that can generate C code [11]. 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—that is specifically tailored for object oriented programs. Section 2 walks through JSKETCH's input and output, along with a running example.

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://plum-umd.github.io/java-sketch/.

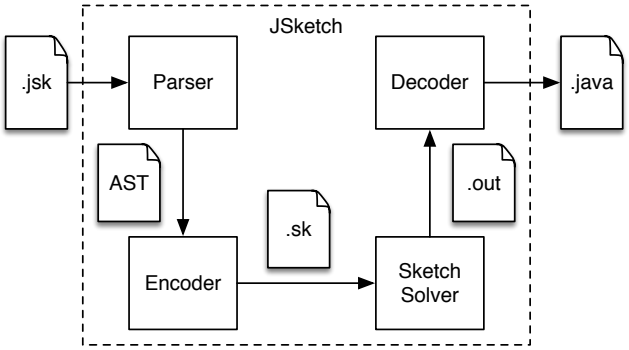

Figure 1: JSKETCH Overview.

2. Overview

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

2.1 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, consider the following Java sketch[1], similar to an example from the SKETCH manual [10]:

```
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 parameter x or 0. Notice that even this very simple sketch has 2^{33} possible instantiations (32 bits of the hole and one bit for the choice of x or 0).

To specify the solution we would like to synthesize, we provide a *harness* containing assertions about the mult2 method:

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

[1] https://github.com/plum-umd/java-sketch/blob/master/test/benchmarks/t109-mult2.java

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;
}
}
```

2.2 Finite Automata

Now consider a harder problem: suppose we want to synthesize a finite automaton given sample accepting and rejecting inputs.[2] 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, however: we do not know the number of automaton states or tokens, so we have no bound on the number of transitions. To solve this problem, we use a feature that JSKETCH inherits from SKETCH: the term minrepeat { e } expands to the minimum length sequence of e's that satisfy the harness. In this case, the body of minrepeat (line 16) is a conditional that encodes 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.

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 [10].)

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

[2] 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.

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).

```
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).

We experimented further with CADsR to see how changing the sketch and harness affects the output. First, we tried running with a smaller harness, i.e., with 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 possible 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 to the examples.

3. Implementation

We implemented JSKETCH as a series of Python scripts that invokes SKETCH as a subroutine. JSKETCH comprises roughly 5.7K lines of code, excluding the parser and regression testing code. JSKETCH parses input files using the Python front-end of ANTLR v3.1.3 [7] and its standard Java grammar. We extended that grammar to support holes, expression-level generators, minrepeat, and the harness and generator modifiers.

There are a number of technical challenges in the implementation of JSKETCH; due to space limitations we discuss only the major ones.

Class hierarchy. The first issue we face is that SKETCH's language is not object-oriented. To solve this problem, JSKETCH follows a similar approach to [8] 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 inheritance 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 out the namespace, and we need to avoid conflating overridden or overloaded method names, or inner classes.

Thus, we name inner classes as *Inner_Outer*, where *Inner* is the name of the nested class and *Outer* is the name of the enclosing class. We also handle anonymous classes by assigning them 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. We simulate the dynamic dispatch mechanism of Java in SKETCH. For each method name M (suitably encoded, as above), we introduce a function dyn_dispatch_M(V_Object self, ...) 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 supportive libraries (if necessary) to .sk files as input to SKETCH. For example, the SimpleMath example in Section 2.1 is translated 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; }
```

We refer the reader elsewhere [9, 10] for details on how SKETCH itself works.

After solving the synthesis problem, JSKETCH then unparses these same Java files, but with unknowns resolved according to the SKETCH synthesis results. We use partial parsing [1] to make this output process simpler.

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 a description of the API of the framework and of the design patterns that 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.

Acknowledgments

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

References

[1] A. Demaille, R. Levillain, and B. Sigoure. TWEAST: A Simple and Effective Technique to Implement Concrete-syntax AST Rewriting Using Partial Parsing. In *SAC '09*, pages 1924–1929, 2009.

[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 *Computer Aided Verification (CAV '15)*, July 2015.

[4] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE '10*, pages 215–224, 2010.

[5] Z. Manna and R. Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, Jan. 1980.

[6] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, Mar. 1971.

[7] T. Parr and K. Fisher. LL(*): The Foundation of the ANTLR Parser Generator. In *PLDI '11*, pages 425–436, 2011.

[8] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. In *PLDI '13*, pages 15–26, 2013.

[9] A. Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

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

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

[12] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI '08*, pages 136–148, 2008.

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

[14] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-Based Inductive Synthesis for Program Inversion. In *PLDI '11*, pages 492–503, June 2011.

A. Tool Demonstration Walkthrough

As mentioned in the introduction, JSKETCH is available at http://plum-umd.github.io/java-sketch/. The tool is at a fairly early stage of development, but is robust enough to be used by the wider research community.

Our demonstration will generally follow the overview in Section 2. Below are more details of what we plan to present.

A.1 Basics

JSKETCH performs *program synthesis*—generating an output program given an input specification.

Let's begin with a small example:

```
$ cat >> SimpleMath.java
class SimpleMath {
    static int mult2(int x) {
        return ?? * {| x , 0 |};
    }
}
```

This is a *sketch* (also *scaffold* or *template*), which is a *partial Java program*. The ?? is a *hole*—unknown integer—and the other part of the product is a *generator*—ranging over the listed expressions. Notice that this sketch has 2^{33} possible instantiations.

In addition to the template, the other important input to JSKETCH is *examples* that specify the expected behavior of the template. These are analogous to unit tests. We provide a *harness* containing assertions about the mult2 method:

```
$ cat >> Test.java
class Test {
    harness static void test() {
        assert SimpleMath.mult2(3) == 6;
    }
}
```

Now we can run JSKETCH on the sketch and harness:

```
$ ./jsk.sh SimpleMath.java Test.java
06:07:15 rewriting syntax sugar
06:07:15 specializing class-level generator
06:07:15 rewriting exp hole
06:07:15 semantics checking
06:07:15 building class hierarchy
06:07:15 encoding result/sk_Test/SimpleMath.sk
06:07:15 encoding result/sk_Test/Test.sk
...
06:07:15 sketch running...
06:07:15 sketch done: result/output/Test.txt
06:07:15 replacing holes
06:07:15 replacing generators
06:07:15 semantics checking
06:07:15 decoding result/java/SimpleMath.java
...
06:07:15 synthesis done
```

The final result synthesized by JSKETCH is a valid Java source file where unknowns have been replaced with the appropriate code:

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

A.2 Database Connection Monitor

Now consider a harder problem: suppose we want to synthesize an automaton-based inline reference monitor to check basic properties of a database connection, namely that the connection is never opened or closed twice in a row and is only closed after being opened. Let's use a simple, efficient implementation: representing states via distinct integers, along with a series of conditionals that encode state transitions.

Here's the initial sketch:

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

Here are some key things to notice about the source code:

- transitions are taken based on the Token interface
- the initial state is arbitrary (line 3)
- the number of states is arbitrary (line 4)
- states are dense (line 5)
- we use an assertion to check the validity of the current state (line 7)
- a transition is arbitrary as it depends on an arbitrary current state and an arbitrary id (line 10)
- minrepeat replicates its body the minimum necessary number of times.
- the number of transitions is arbitrary (line 9)
- the number of accepting states is arbitrary (line 15); by packing the states densely, we could use an inequality here

Now we can define an inline reference monitor as follows:

```
17 class DBConnection {
18     class Monitor extends Automaton {
19         final static Token OPEN =
20             new Token() { public int getId() { return 1; } };
21         final static Token CLOSE =
22             new Token() { public int getId() { return 2; } };
23         public Monitor() { }
24     }
25     Monitor m;
26     public DBConnection() { m = new Monitor(); }
27     public boolean isErroneous() { return ! m.accept(); }
28     public void open() { m.transition (Monitor.OPEN); }
29     public void close() { m.transition (Monitor.CLOSE); }
30 }
```

The key idea is that each database connection operation is associated with an unique id, and the monitor maintains an automaton that keeps receiving operation ids. At any point, a client can check the status of the connection by asking the monitor whether it is in an accepting state.

As expected, we need to provide a harness:

```
31 class TestDBConnection {
32     harness static void scenario_good() {
33         DBConnection conn = new DBConnection();
34         assert ! conn.isErroneous();
```

```
35          conn.open(); assert ! conn.isErroneous();
36          conn.close(); assert ! conn.isErroneous();
37      }
38      // bad: opening more than once
39      harness static void scenario_bad1() {
40          DBConnection conn = new DBConnection();
41          conn.open(); conn.open(); assert conn.isErroneous();
42  }   }
```

These examples illustrate one normal usage—opening a connection and closing it—and one abnormal usage—opening a connection twice. Given these harnesses, JSKETCH finds a solution:

```
43  class Automaton {
44    int state = 0;
45    static int num_state = 3;
46    public void transition (Token t) { ...
47      assert 0 ≤ state && state < 3;
48      if (state == 0 && id == 1) { state = 1; return; } // open@
49      if (state == 1 && id == 1) { state = 2; return; } // open 2x
50    }
51    public boolean accept() { return state ≤ 1; } ... }
```

This sort of looks okay, but it's odd that there are no transitions for the close operation. When the monitor is in state 1 and the given operation is close, it is fine for the monitor to stay at the same accepting state. But, it is problematic if we close the connection more than once, which we should have specified:

```
52  class TestDBConnection { ...
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  }   }
```

After adding that abnormal case—closing twice, JSKETCH finds this solution:

```
59  class Automaton {
60    int state = 0;
61    static int num_state = 3;
62    public void transition (Token t) { ...
63      assert 0 ≤ state && state < 3;
64      if (state == 0 && id == 1) { state = 1; return; } // open@
65      if (state == 1 && id == 1) { state = 2; return; } // open 2x
66      if (state == 1 && id == 2) { state = 0; return; } // ( init )@
67      if (state == 0 && id == 2) { state = 2; return; } // close 2x
68    }
69    public boolean accept() { return state ≤ 1; } ... }
```

The resulting automaton is exactly same as what one can write by hand.

A.3 A Regular Language: Lisp-Style Identifier

Now let's consider synthesizing another automaton, trying to create a finite automaton given sample accepting and rejecting inputs. One benefit of Java as an object oriented language is code reuse via subclassing, so we could just make another class that extends Automaton, assuming we want this to be part of the same program. But subclassing won't quite work here because we need different automata for each use case.

To solve this problem, we can make Automaton a *class generator*, so that it can be instantiated differently in different superclass contexts.

```
70  generator class Automaton { ... }
71  class DBConnection {
72      class Monitor extends Automaton { ... }
73      ...
74  }
75  class CADsR extends Automaton { ... }
```

Now let's use this to synthesize an example automaton:

```
76  class CADsR extends Automaton { ...
77      public boolean accept(String str) {
78          state = init_state_backup ;
79          transitions ( convertToIterator ( str ));
80          return accept();
81  }   }
```

Note the overloaded accept(String) method. Now, we need to specify sample strings that are accepted or rejected by the synthesized automaton. Suppose we want to synthesize an automaton that recognizes Lisp-style identifier `c(a|d)+r`. The following harness constructs a CADsR instance and makes several assertions about its behavior:

```
82  class TestCADsR {
83      harness static void examples() {
84          CADsR a = new CADsR();
85          assert ! a.accept("c"); assert ! a.accept("cr");
86          assert a.accept("car"); assert a.accept("cdr");
87          assert a.accept("caar"); assert a.accept("cadr");
88          assert a.accept("cdar"); assert a.accept("cddr");
89  }   }
```

If we provide less examples, e.g., if we remove examples about rejected strings, the synthesizer simply returns an automaton that does not make any transitions, while the initial state is an accepting state. This awkward automaton actually conforms to any accepted strings, and one can easily figure out the necessity of rejected strings.

To see the advantage of using minimize, let's run synthesis without line 5. We get:

```
90  class Automaton2 {
91    int state = 106; static int num_state = 120;
92    public void transition (Token t) { ...
93      if (state == 106 && id == 99) { state = 64; return; }  // c
94      if (state == 64 && id == 97) { state = 100; return; }  // ca
95      if (state == 100 && id == 114) { state = 50; return; } // car@
96      if (state == 64 && id == 100) { state = 119; return; } // cd
97      if (state == 119 && id == 114) { state = 32; return; } // cdr@
98      if (state == 64 && id == 114) { state = 72; return; }  // cr
99    }
100   public boolean accept() { return state ≤ 50; } ... }
```

Notice that the synthesizer picked fairly strange numbers for the states and left a lot of states unused. Moreover, the automaton is inefficient in that it uses two different paths and final states to accept "car" and "cdr".

If we run synthesis again using minimize, we get:

```
101 class Automaton2 {
102   int state = 0; static int num_state = 3;
103   public void transition (Token t) { ...
104     assert 0 ≤ state && state < 3;
105     if (state == 0 && id == 99)  { state = 1; return; } // c
106     if (state == 1 && id == 97)  { state = 2; return; } // ca
107     if (state == 1 && id == 100) { state = 2; return; } // cd
108     if (state == 2 && id == 114) { state = 0; return; } // c(a|d)+r@
109   }
110   public boolean accept() { return state ≤ 0; } ... }
```

This result is better in the sense that it uses dense states and that it encompasses only one path and final state to accept the valid strings.

To double-check whether it is indeed the minimum number of states, we can test with the bounded number of states:

```
111 class Automaton {
112     static int num_state = 2; ...
113 }
```

In this case, the synthesizer runs for more than half an hour and then fails, as there is no possible solution using only two states.

A.4 Internals of JSKETCH

If time permits, we will show a bit of JSKETCH's translation to SKETCH. For example, the translation of the mult2 example looks like:

```
$ cat result/sk_Test/SimpleMath.sk
...
int e_h1 = ??;
int mult2_SimpleMath_int(int x) {
return e_h1 * {| x | 0 |};
}
$ cat result/sk_Test/Test.sk
...
harness void test_Test() {
assert mult2_SimpleMath_int(3) == 6;
}
```

JSKETCH extracts the synthesis results by looking at how each hole was solved by SKETCH:

```
$ cat result/log/log.txt
...
06:07:15 [DEBUG] java_sk/decode/finder.py:41
hole: SimpleMath.e_h1
06:07:15 [INFO] java_sk/decode/__init__.py:69
replacing holes
06:07:15 [DEBUG] java_sk/decode/replacer.py:72
replaced: SimpleMath.e_h1 = 2
06:07:15 [DEBUG] java_sk/decode/replacer.py:89
replaced: e_h1 @ int SimpleMath.mult2(int) with 2
06:07:15 [DEBUG] java_sk/decode/finder.py:93
generator@mult2: {| x | 0 |}
06:07:15 [INFO] java_sk/decode/__init__.py:79
replacing generators
06:07:15 [DEBUG] java_sk/decode/replacer.py:151
{| x | 0 |} => x
...
```

Then it traverses the original Java sketch and outputs it, plugging in the solved values for the holes.