ASketch: A Sketching Framework for Alloy

Kaiyuan Wang
University of Texas at Austin, USA
kaiyuanw@utexas.edu

Darko Marinov
University of Illinois at Urbana-Champaign, USA
marinov@illinois.edu

Allison Sullivan
University of Texas at Austin, USA
allisonksullivan@utexas.edu

Sarfraz Khurshid
University of Texas at Austin, USA
khurshid@utexas.edu

ABSTRACT

Alloy is a declarative modeling language that supports first-order logic with transitive closure. Alloy has been used in a variety of domains to model software systems and find design deficiencies. However, it is often challenging to make an Alloy model correct or to debug a faulty Alloy model. ASketch is a sketching/synthesis technique that can help users write correct Alloy models. ASketch allows users to provide a partial Alloy model with holes, a generator that specifies candidate fragments to be considered for each hole, and a set of tests that capture the desired model properties. Then, the tool completes the holes such that all tests for the completed model pass. ASketch uses tests written for the recently introduced AUnit framework, which provides a foundation of testing (unit tests, test execution, and model coverage) for Alloy models in the spirit of traditional unit testing. This paper describes our Java implementation of ASketch, which is a command-line tool, released as an open-source project on GitHub. Our experimental results show that ASketch can handle partial Alloy models with multiple holes and a large search space. The demo video for ASketch can be found at https://youtu.be/T5NIVsV329E.

CCS CONCEPTS

- Software and its engineering → Source code generation;
- Software and its engineering → Software modeling languages

KEYWORDS

Sketching, first-order logic, ASketch

1 INTRODUCTION

Software modeling languages, which can be used to describe key attributes of software systems, play an important role in helping engineers build reliable systems. Many modeling languages have been proposed over the last few decades [1, 2, 8]. Alloy is a well-known modeling language which comes with an automatic SAT-based analyzer [13] that performs analysis using a bounded scope on the universe of discourse. Over the past few years, many Alloy extensions have been developed [3, 4, 14, 17].

The Alloy analyzer helps users validate their models using automated analysis. Typically, Alloy users apply the following validation approaches, which are supported by the standard Alloy analyzer: (1) solving constraints expressed in the model and checking if expected Alloy instances are present while unexpected instances are absent; (2) writing related formulas and validating if the expected properties among them hold, e.g., implication or equivalence; and (3) using a SAT solver’s unsat core feature to highlight parts of the model that make the formulas unsatisfiable.

A few recent projects adapt the notion of traditional testing from imperative programs to declarative models in Alloy. The AUnit framework [9, 12] introduces for Alloy the basic testing foundations, i.e., unit testing, test execution, and model coverage. Follow-up work on MuAlloy [11, 15] introduces mutation testing for Alloy and provides a technique for mutation-based test generation for Alloy models. A more recent complementary project, called ASketch [16, 18], introduces sketching [7] for partial Alloy models. ASketch allows users to write high-level skeletal models, while the tool synthesizes the low-level details.

This paper describes our Java implementation of ASketch, which is a command-line tool that we released as an open-source project (https://github.com/kaiyuanw/ASketch). ASketch takes as input a partial Alloy model with holes, a generator that provides potential candidate fragments to be considered for each hole, and a set of AUnit tests that capture the desired properties of the expected model. ASketch encodes all inputs as a single meta-model in Alloy and invokes the SAT solver to explore the search space. The output is a complete Alloy model with holes replaced with concrete candidate fragments such that all tests pass. We evaluate ASketch using 10 Alloy models that were used in previous work [4, 10]. The experimental results show that ASketch is able to find a solution of a partial Alloy model with multiple holes and a large search space in a reasonable amount of time (up to 15 holes and more than 4.4e11 possible candidate solutions in 7 seconds).

2 AUNIT BACKGROUND

ASketch takes as one of its inputs AUnit tests and then completes partial Alloy models with holes such that all provided AUnit tests pass. We briefly describe AUnit via an example; more details are available elsewhere [9, 12]. Figure 1 shows an acyclic singly-linked list model. The model declares a singleton set of one list atom.
We first illustrate the input language, specifically the supported hole kinds and the generator grammar. Then, we describe through an example how the synthesizer translates the sketching problem into a constraint-solving problem.

### 3.1 Input Language

#### 3.1.1 Hole Kinds

We first illustrate the input language, specifically the supported hole kinds and the generator grammar. For example, users can use a quantifier hole $\langle q, i d \rangle$ where $Q$ indicates that it is the quantifier hole kind, and $i d$ is an identifier that refers to a generator which provides the candidate fragments for the hole via a regular expression (regex). The default regex for a quantifier hole is ('all'/'no'/'some'/ 'lone'/ 'one'). Another example is the expression hole, $\langle i d, e i r \rangle$, which can be completed with any expression syntactically valid in the context of the hole. Note that the regex for unary operator formula hole is either negation (!) or an empty string ($\epsilon$).

<table>
<thead>
<tr>
<th>Hole Kind</th>
<th>Notation</th>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Operator</td>
<td>$\langle B O \rangle$</td>
<td>&amp;+, -</td>
</tr>
<tr>
<td>Compare Operator</td>
<td>$\langle C O \rangle$</td>
<td>=, in, !in</td>
</tr>
<tr>
<td>Logical Operator</td>
<td>$\langle L O \rangle$</td>
<td></td>
</tr>
<tr>
<td>Quantifier</td>
<td>$\langle Q \rangle$</td>
<td>all, no, some, lone, one</td>
</tr>
<tr>
<td>Unary Operator</td>
<td>$\langle U O \rangle$</td>
<td>no, some, lone, one</td>
</tr>
<tr>
<td>Unary Operator Expression</td>
<td>$\langle U O E \rangle$</td>
<td>$-$, $*$, $+$</td>
</tr>
<tr>
<td>Unary Operator Formula</td>
<td>$\langle U O F \rangle$</td>
<td>$\epsilon$, $e$</td>
</tr>
<tr>
<td>Expression</td>
<td>$\langle E \rangle$</td>
<td>any expression</td>
</tr>
</tbody>
</table>

#### 3.1.2 Generator Grammar

Users can provide a generator for a hole using regexes with the following grammar:

```
regExDecl ::= id ::= "(" regex ")"
regex ::= nonSpecial \ regex | "?" \ regex | "(" regex ")" | regex regex | regex ")" regex
```

For `regExDecl`, `id` introduces an identifier to be referred from a hole (e.g., $\langle q, i d \rangle$), and $e$ is a regex. We follow the design choice of the Sketch framework [7] that includes three regex operators—option (`?`), concatenation (`e1 e2`), and choice (`e1 | e2`)—as well as parentheses for precedence. `nonSpecial` can be any string that contains characters supported by the Alloy grammar except for `\` (carriage return), `/` (slashes), and `\` (backslash), respectively. We use ANTLR4 [5] to generate the parser with the generator grammar and implement a backtracking algorithm to decode the regex into all possible candidate fragments.

### 3.2 Synthesizer

ASketch converts the sketching problem into a constraint-solving problem in the Alloy language itself, which is then solved by the Alloy analyzer. Specifically, ASketch generates a single Alloy meta-model that encodes all possible solutions, i.e., candidate models obtained from all possible combinations of all candidate fragments for all holes.

For example, consider an Alloy user who wants to model the Acyclic property (shown in Figure 1) and comes up with the following skeleton for the predicate:

```alloy
?? n: Node | n in List.header.\!link \Rightarrow n \!in n
```

The user is not sure what quantifier (first `??`) and what expression (second `??`) to use in the skeleton but knows roughly what to search for each hole and also which instances are desired and
which are undesired. The user can provide the following partial Alloy model, a generator and a set of tests to ASketch:

```alloy
// Model with holes
one sig List { header: lone Node } sig Node { link: lone Node }

pred Acyclic() {
    \forall a \in n in List.header.*link \iff \n in [E,n] }
run Acyclic // Generator
q := { (all|some|no |) e := e in (List.header|Node|n)(.(*|^)?link)? |}

// AUnit tests that capture desired model properties
pred test (...) {

    The generator states that the quantifier hole should take a value from ['all', 'some', 'no'], and the expression hole should take a value from ['List.header', 'List.header.link', 'List.header.*link', 'List.header.elem', 'Node', 'Node.elem', 'Node.*link', 'n', 'n.elem', 'n.*link', 'n.elem']. The user also provides AUnit tests similar to the example in Figure 3 to represent both desired and undesired valuations for acyclic lists.

ASketch automatically creates an Alloy meta-model shown in Figure 5. The field declarations are removed from the generated meta-model (line 1). Instead, ASketch parameterizes the Acyclic predicate with one new parameter per signature and one new parameter per field (lines 2-3).

The quantifier hole is replaced with a predicate call (line 4) q, which is declared to encode all possible quantifiers that can appear in the quantifier hole (lines 7-14). The first parameter of the predicate h in line 7 represents the value chosen for the hole. Lines 5-6 encode the potential candidate quantifiers as signatures and RQ is one of Q.All, Q.Some, or Q.No. The rest of the parameters in q are exactly the same as the parameters in the Acyclic predicate. The body of the predicate q states how the parameter h determines the value of the qualiﬁer hole: if h is Q.All, then the quantifier hole is 'all' (lines 9-10), and similarly for 'some' (lines 11-12) and 'no' (lines 13-14). Any reference to a declared signature or field is replaced by the corresponding parameter in predicate q, e.g., 'List.header.*link' is replaced with 'List.header.*link'.

The expression hole is replaced with a function call (lines 10, 12, and 14) expr, which is declared to encode all possible expressions that can appear in the expression hole (lines 18-28). Similar to the predicate q, the first parameter of function expr (h in line 18) represents the value chosen for the hole. Lines 15-17 encode the candidate expressions as signatures (E0 to E11) and RE is equal to one of them. Note that the expression hole is in the scope of variable n, so the function expr also has a parameter for it (line 19). Because all user speciﬁed candidate expressions are of arity 1, the return type of expr also has an arity of 1 (univ in line 19). The body of the function expr states how the parameter h determines the value of the expression hole: if h is E0, then the expression is 'List.header' (line 20), which maps back to 'List.header' in the original model, and the expression encoding is similar for the other candidate expressions.

The AUnit test from Figure 3 is translated to a fact (lines 30-34), where the signatures and fields are represented by fresh variables introduced by the let expressions. The Acyclic predicate is also invoked with the corresponding fresh variables. The entire meta-model uses Alloy facts to require that all AUnit tests must be satisﬁed. If we invoke the empty run command (line 29), then the Alloy analyzer returns a solution that assigns Q.All
```
We ran ASketch on a MacBook Pro with a 2.5GHz Intel Core i7. We describe the experiments and results for evaluating ASketch.

The number of primary variables for the generated meta-models of candidate fragments, then the sketching problem has a large number of holes, and the generator for each hole provides a large number of primary variables and clauses among all generated meta-models, and it takes more than 6 seconds to find the first solution. In contrast, arr has the smallest number of primary variables and clauses, and it takes the least amount of time to find the first solution. However, the increasing size of the search space does not necessarily imply increasing number of primary variables and clauses in the generated meta-model. For example, grade has a search space of size 1.2E7 and deadlock has a search space of size 4800, but the generated meta-model for deadlock is more complicated than that for grade. Overall, these results show that ASketch is able to handle partial Alloy models with multiple holes (up to 15) and a large search space (more than 400 billion).

### 5 EVALUATION

We describe the experiments and results for evaluating ASketch. We ran ASketch on a MacBook Pro with a 2.5GHz Intel Core i7-4870HQ. Figure 6 lists the 10 partial Alloy models involved in the experiment and the corresponding stats from ASketch. model is the model names: arr models arrays; bt models binary trees; cd models Java class hierarchy; contains checks whether a list contains an element; etree models two colored undirected trees; deadlock models process deadlocks; dll models doubly-linked lists; grade models teaching assistants grading assignments; remove models removing an element from a list; and sll models singly-linked lists. All models are from previous work [6, 15, 18].

We make these models partial by replacing fragments with various hole kinds. For expression holes, we use candidate expressions of different arities (e.g., arity 1 and 2) and types (e.g., signature types and the Alloy integer type). #hole shows the number of holes for each partial model. space shows the size of the search space, i.e., the number of combinations of candidate fragments across all holes. #test shows the number of AUnit tests provided to complete each partial model. scp shows the scope used to search for solutions. #prim and #cls show the number of primary variables and clauses for running the generated meta-model, which measures the complexity of the sketching problem. T_solv shows the time (in milliseconds) taken to find the first solution that satisfies all tests.

The number of holes of our evaluation models ranges from 3 to 15. The search space ranges from 80 to 4.4e11. We use the same test suite that comes with each model in the previous work and delete irrelevant tests that do not invoke the predicates we want to sketch. The number of primary variables for the generated meta-models ranges from 49 to 531. The number of clauses for the meta-models ranges from 1873 to 9.8e4. The solving time to find the first solution ranges from 96ms to 6473ms.

The size of the search space depends on the number of holes and the generators for the holes. If a partial model has a large number of holes, and the generator for each hole provides a large number of candidate fragments, then the sketching problem has a large search space. Typically, a more complex meta-model makes constraint solving slower. For example, etree has the largest number of primary variables and clauses among all generated meta-models, and it takes more than 6 seconds to find the first solution. In contrast, arr has the smallest number of primary variables and clauses, and it takes the least amount of time to find the first solution. However, the increasing size of the search space does not necessarily imply increasing number of primary variables and clauses in the generated meta-model. For example, grade has a search space of size 1.2E7 and deadlock has a search space of size 4800, but the generated meta-model for deadlock is more complicated than that for grade. Overall, these results show that ASketch is able to handle partial Alloy models with multiple holes (up to 15) and a large search space.

### ACKNOWLEDGMENTS

We thank Manos Koukoutos, Viktor Kuncak, and the anonymous reviewers for the valuable comments and helpful suggestions. The work is partially supported by the National Science Foundation under Grant Nos. CCF-1421503, CCF-1718903, and CNS-1740916.

### REFERENCES