ASketch: A Sketching Framework for Alloy

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

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

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

KEYWORDS

Sketching, first-order logic, ASketch

ACM Reference Format:
Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. ASketch: A Sketching Framework for Alloy. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages.
https://doi.org/10.1145/3236024.3264594

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 languages 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 AUnitframework [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

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264594

```
one sig List { header: lone Node } sig Node { link: lone Node }
pred Acyclic() {
  all n: Node | n in List.header.*link => n !in n.^link }
run Acyclic
```

Figure 1: Acyclic Singly-linked List

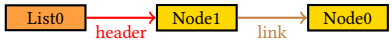

Figure 2: Example List Instance

```
pred test {
  some disj List0: List { some disj Node0, Node1: Node {
    List = List0 and Node = Node0 + Node1
    header = List0->Node1 and link = Node1->Node0
    Acyclic[] } } }
run test
```

Figure 3: Example Test for List

and a set of Node atoms. Each List atom has zero or one header nodes. Each Node atom has zero or one subsequent nodes along the link. Both header and link are partial functions. The predicate Acyclic states that the List is acyclic if the following holds: for each Node n, if n is reachable from the List's header following zero or more traversals along the link, then n is not reachable from itself following one or more traversals along the link.

If we run the Acyclic predicate using Alloy analyzer, we can get an example instance shown in Figure 2. The instance contains a single List atom (List0) and two Node atoms (Node0 and Node1). List0's header is Node1, and Node1's next node is Node0. List0 satisfies the Acyclic predicate because there is no cycle in the list.

An AUnit test is a pair of a model valuation and a run command. For example, the instance in Figure 2 can be written as an AUnit test shown in Figure 3. The test declares a single List atom (List0) and 2 disjoint Node atoms (Node0 and Node1). It restricts the entire List set to be {List0} and Node set to be {Node0, Node1}. The test predicate also states that the header maps List0 to Node1, and the link maps Node1 to Node0. Since the valuation represents an acyclic list, we expect the Acyclic predicate to be satisfied. Thus, we invoke Acyclic[] in the test predicate and expect the existence of a solution to the run test command. In this case, running the test predicate results in an isomorphic Alloy instance similar to the one shown in Figure 2. If a valuation is not expected, then we can invoke the negation of the corresponding predicate in the test predicate itself. For example, if a list is cyclic, then we can invoke !Acyclic[] in the test. ASketch assumes that all tests should pass, i.e., all test predicates should be satisfied.

3 TECHNIQUE

ASketch has two main components:

- *Input Interpreter*: Parse the model with holes and interpret the generator to create candidate fragments.
- *Output Synthesizer*: Encode the model with holes, the candidate fragments and the AUnit tests into a meta-model and invoke SAT solver to search for solutions.

The quality of the solutions depends on both the generator and the test suite. If the generator provides fragments that are semantically equivalent to the desired candidate fragments, and the test suite captures the properties of the desired model, then ASketch is more likely to generate the desired solution in a few iterations. In practice, there could be multiple solutions that make all the tests pass. ASketch can iteratively provide these solutions if the user wants to inspect them.

Hole Kind	Notation	Candidates
Binary Operator	\BO\	&, +, -
Compare Operator	\CO\	=, in, !=, !in
Logical Operator	\LO\	||, &&, <=>, =>
Quantifier	\Q\	all, no, some, lone, one
Unary Operator	\UO\	no, some, lone, one
Unary Operator Expression	\UOE\	~, *, ^
Unary Operator Formula	\UOF\	!, ϵ
Expression	\E\	any expression

Figure 4: Supported Holes

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. Figure 4 shows the kind, notation, and candidate fragments for all holes supported by ASketch. For example, users can use a quantifier hole \Q,id\, where Q indicates that it is the quantifier hole kind, and id 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, \E,id\, 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 (ϵ).

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
```

For regExDecl, id introduces an identifier to be referred from a hole (e.g., \Q,id\), and e is a regex. We follow the design choice of the Sketch framework [7] that includes three regex operators—option (e?), 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 "(", ")", and "|" which need to be escaped as "\(", "\)", and "\|", 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:

```
?? n: Node | n in List.header.*link => n !in ??
```

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:

```
// Model with holes
one sig List { header: lone Node } sig Node { link: lone Node }
pred Acyclic() {
  \Q,q\ n: Node | n in List.header.*link => n !in \E,e\ }
run Acyclic
// Generator
q := {| all|some|no |}
e := {| (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.*link", "Node", "Node.link", "Node.^link", "Node.*link", "n", "n.link", "n.^link", "n.*link"]. 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 q (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 "Lists.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 "Lists.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

```
1. one sig List {} sig Node {}
2. pred Acyclic(Lists: List, header: List->Node,
3.   Nodes: Node, link: Node->Node) {
4.   q[RQ, Lists, header, Nodes, link] }
5. one sig RQ in Q {} abstract sig Q {}
6. one sig Q_All, Q_Some, Q_No extends Q {}
7. pred q(h: Q, Lists: List, header: List->Node,
8.   Nodes: Node, link: Node->Node) {
9.   h = Q_All => all n: Nodes | n in Lists.header.*link
10.    => n !in expr[RE, Lists, header, Nodes, link, n]
11.  h = Q_Some => some n: Nodes | n in Lists.header.*link
12.    => n !in expr[RE, Lists, header, Nodes, link, n]
13.  h = Q_No => no n: Nodes | n in Lists.header.*link
14.    => n !in expr[RE, Lists, header, Nodes, link, n] }
15. one sig RE in E {} abstract sig E {}
16. one sig E0, E1, E2, E3, E4, E5, E6,
17.   E7, E8, E9, E10, E11 extends E {}
18. fun expr(h: E, Lists: List, header: List->Node,
19.   Nodes: Node, link: Node->Node, n: Node): univ {
20.   (h = E0 => Lists.header else
21.   (h = E1 => Lists.header.link else
22.   (h = E2 => Lists.header.^link else
23.   (h = E3 => Lists.header.*link else (h = E4 => Nodes else
24.   (h = E5 => Nodes.link else (h = E6 => Nodes.^link else
25.   (h = E7 => Nodes.*link else (h = E8 => n else
26.   (h = E9 => n.link else (h = E10 => n.^link else
27.   (h = E11 => n.*link else none))))))))))))
28. }
29. Sketch: run {}
30. fact test {
31.   some disj List0: List { some disj Node0, Node1: Node {
32.     let Lists = List0 { let header = List0->Node1 {
33.     let Nodes = Node0 + Node1 { let link = Node1->Node0 {
34.     Acyclic[Lists, header, Nodes, link] }}}}}}
35. // More tests...
```

Figure 5: Meta-Model for List

to RQ and E10 to RE, which indicates that the quantifier hole and the expression hole in the original partial model should be replaced by "all" and "n.^link", respectively.

4 USAGE

In this section, we describe how users can invoke ASketch. More details can be found on the ASketch GitHub homepage.

To sketch a partial Alloy model, run "./asketch.sh --run -m <arg> -f <arg> -t <arg> [-s <arg>] [-n <arg>]" or "./asketch.sh --run --model-path <arg> --fragment-path <arg> --test-path <arg> [--scope <arg>] [--sol-num <arg>]". The options are:

- "-m,--model-path": This argument is *required*. Pass the file name of the partial Alloy model to be sketched.
- "-f,--fragment-path": This argument is *required*. Pass the generator which provides the candidate fragments to be considered for each hole.
- "-t,--test-path": This argument is *required*. Pass the test suite which contains AUnit tests that capture the desired properties of the expected model.
- "-s,--scope": This argument is *optional*. Pass the Alloy scope for solving the generated Alloy meta-model. The scope is typically larger than or equal to the minimum scope necessary to make all AUnit tests satisfied. If the argument is not specified, the default value of 3 is used.
- "-n,--sol-num": This argument is *optional*. Pass the number of unique solutions that ASketch should report. If the argument is not specified, the default value of 1 is used.

For each run, the ASketch tool reports: (1) the candidate fragments for each hole after expanding the corresponding regular expression; (2) the order, the associated identifier, the kind, and the

model	#hole	space	#test	scp	#prim	#cls	T_{sol}
arr	3	220	5	3	49	1873	96
bt	6	3.7e4	20	3	251	3.9e4	554
cd	10	1.5e7	17	3	186	2.1e4	550
contains	3	80	16	3	275	1.2e4	148
ctree	15	4.4e11	31	3	531	9.8e4	6473
deadlock	6	4800	16	3	371	4.2e4	1083
dll	13	1.2e7	27	3	297	2.7e4	324
grade	10	1.2e7	17	4	116	4076	114
remove	4	2.1e4	20	3	377	4.9e4	1197
sll	5	3072	16	3	162	2e4	609

Figure 6: ASketch Model Stats (time in milliseconds)

candidate space size for each hole; (3) the size of the entire search space computed by multiplying the sizes of individual fragments across all holes; and (4) a set of solutions, if any, as well as the solving time for each solution. Furthermore, ASketch stores the generated Alloy meta-model in a hidden directory ("`${project_dir}/.hidden/solve.als`").

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; **ctree** 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. $\mathbf{T}_{sol}$ 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, **ctree** 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 models with multiple holes (up to 15) and a large search space (more than 400 billion).

6 CONCLUSION

This paper introduced the open-source ASketch tool for sketching partial Alloy models with holes. ASketch provides command-line options to automatically translate the sketching problem into a constraint solving problem and then search for solutions. Given a partial Alloy model with holes, a generator, and a set of AUnit tests that capture the desire properties of the model, ASketch is able to report a set of solutions such that all AUnit tests pass. Our evaluation shows that ASketch is able to handle partial Alloy models with multiple holes 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

[1] Jean-Raymond Abrial. 2005. *The B-Book: Assigning Programs to Meanings.*
[2] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM TOSEM* (2002).
[3] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-purpose Higher-order Relational Constraint Solver. In *ICSE.*
[4] Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. 2017. The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance. In *ESEC/FSE.*
[5] Terence Parr. 2013. *The Definitive ANTLR 4 Reference.*
[6] Salman Saghafi, Ryan Danas, and Daniel J. Dougherty. 2015. Exploring Theories with a Model-Finding Assistant. In *CADE.*
[7] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph.D. Dissertation. University of California, Berkeley.
[8] J Michael Spivey. 1988. *Understanding Z: A Specification Language and Its Formal Semantics.*
[9] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. 2018. AUnit: A Test Automation Tool for Alloy. In *ICST.*
[10] Allison Sullivan, Kaiyuan Wang, Sarfraz Khurshid, and Darko Marinov. 2017. Evaluating State Modeling Techniques in Alloy. In *SQAMIA.*
[11] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *ICST.*
[12] Allison Sullivan, Razieh Nokhbeh Zaeem, Sarfraz Khurshid, and Darko Marinov. 2014. Towards a Test Automation Framework for Alloy. In *SPIN.*
[13] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS.*
[14] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. Automated Model Repair for Alloy. In *ASE.*
[15] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. MuAlloy: A Mutation Testing Framework for Alloy. In *ICSE.*
[16] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. 2018. Systematic Generation of Non-Equivalent Expressions for Relational Algebra. In *ABZ.*
[17] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. Fault Localization for Declarative Models in Alloy. In *eprint arXiv:1807.08707.*
[18] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. Solver-based Sketching Alloy Models using Test Valuations. In *ABZ.*