Solver-based Sketching of Alloy Models using Test Valuations

Kaiyuan Wang[1], Allison Sullivan[1], Darko Marinov[2], and Sarfraz Khurshid[1]

[1]University of Texas at Austin
[2]University of Illinois at Urbana-Champaign
kaiyuanw@utexas.edu, allisonksullivan@utexas.edu, marinov@illinois.edu, khurshid@utexas.edu

Abstract. We introduce *ASketch*, the first framework for sketching models in the Alloy language. The Alloy Analyzer is a SAT-based constraint solver that allows users to create *valuations* for relations with respect to given constraints and bound on the universe of discourse. Alloy users routinely use the valuations to validate their models: enumerate some valuations and inspect them to detect underconstraints or overconstraints. Our key insight is that valid and invalid valuations enable *sketching* Alloy models where the user writes a *partial* model with *holes* and provides some valuations, and the sketching infrastructure completes the model by synthesizing Alloy fragments for the holes.
ASketch offers the following extensions to Alloy: (1) it expands the Alloy grammar, allowing users to write holes in an Alloy model; (2) it can parse regular expressions and automatically generate pools of matching fragments to replace the holes; (3) it includes a solver-based technique that encodes the model with holes, the fragments for each hole, and the expected valuations to a *meta-model* which completes the holes when solved. Experimental results show that *ASketch* works well for different Alloy models with various number of holes, providing a promising approach to bring the success of traditional program sketching for imperative and functional programs to declarative, relational logic.

1 Introduction

Building software models plays an important role in building reliable systems. Alloy [11] is a well-known, relation-based modeling language that has been used in academic and industrial settings [8,12,22,45]. Alloy has a SAT-based analyzer that performs automatic analysis over a user-defined *scope*, i.e., bound on the universe of discourse. Specifically, the analyzer finds *instances*, i.e., valuations for relations in the model such that the formulas in the model evaluate to true. The analyzer can also find *counterexamples* that refute properties of interest; an instance for the negation of the property formula serves as a counterexample. While Alloy's expressive notation allows succinct formulation of complex properties, reasoning about the correctness of Alloy formulas, e.g., in the presence of quantification and transitive closure, requires much care. Because Alloy models are effectively logical constraints, they can have two basic kinds of faults:

overconstraints that rule out valid valuations and *underconstraints* that permit invalid valuations.

We introduce the first approach for *sketching* Alloy models, where the user does not need to write complete models. Instead, the user writes a *partial* model with *holes* and also provides (1) some regular expressions that encode possible fragments for each hole and (2) some valid and invalid valuations that serve as test cases [35, 38] for the desired model. Our key insight is that these test valuations enable sketching Alloy models, where the sketching framework completes the partial model with respect to the given fragments and valuations.

Our sketching framework, called *ASketch*, focuses on sketching several constructs of Alloy models, including relational expressions, logical operators, and quantifiers. Given a partial model and the corresponding test valuations, *ASketch* first parses the user-provided regular expressions and generates pools of matching fragments that can replace the holes. Then, *ASketch* systematically explores the resulting search space of candidate Alloy models, to find a model that satisfies all test valuations. Specifically, *ASketch* uses constraint *solving* to explore the space of candidate models by creating one Alloy *meta-model* that encodes the model to sketch along with the fragments for holes and test valuations all at once. The meta-model effectively encodes multiple Alloy models, i.e., all models from the entire candidate space. Finally, *ASketch* uses the Alloy Analyzer to find solutions that can fill in the holes.

We perform an experimental evaluation of *ASketch* using 24 sketches derived from 5 core Alloy models. Experimental results show that *ASketch* can complete sketches that can simultaneously have up to 3 expression holes and 3 non-expression holes. To highlight the complexity of the underlying problem, one example sketch, *BinaryTree* with 6 holes, has a search space of over 4 billion candidate Alloy models (3 expression holes with 400 expression fragments each and 3 non-expression holes with 4 fragments each). *ASketch* finds a solution Alloy model (w.r.t. 16 test valuations) in 12 minutes, and the Alloy meta-model generated by *ASketch* creates a SAT problem with 1,378 primary variables and 1,188,735 clauses.

While *ASketch* introduces a new technique for writing Alloy models in general, a particular application that we envision for *ASketch* is for *education* about Alloy and more broadly, software modeling using relational specifications. Our experience with beginner Alloy users shows that they often struggle to make their formulas "just right". They have a general idea for a formula skeleton, and they can tell whether certain instances should or should not satisfy a formula, but they still make mistakes that overconstrain or underconstrain their models. We expect that beginners could greatly benefit from an iterative methodology where the user could start from some skeleton formula with holes, use *ASketch* to complete the formula, obtain some valuations, label them as valid or invalid, and repeatedly iterate until getting all (and only) the valuations that the user expects. In fact, our evaluation subjects are inspired by the example models that beginners often struggle with.

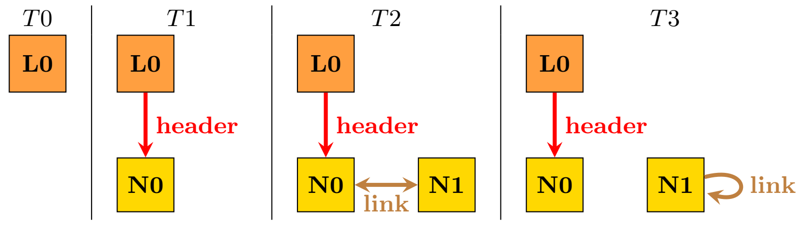

Fig. 1. Four test valuations shown graphically: $T0$, $T1$, and $T3$ are valid for the expected acyclicity; $T2$ is invalid. $L0$ is the list atom; $N0$ and $N1$ are node atoms.

This paper makes the following contributions:
Idea: We introduce the idea of sketching Alloy models using test valuations.
***ASketch*:** We introduce a technique for completing Alloy sketches based on constraint solving.
Experiments: We present an experimental evaluation with small but intricate Alloy formulas; the results show that *ASketch* introduces a promising approach for sketching Alloy models.

2 Example

To illustrate our *ASketch* approach, consider the following partial Alloy model for an acyclic singly linked list:

```
one sig List { header: lone Node }   sig Node { link: lone Node }
pred Acyclic() { \Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ \E,e\ }
q := {| all|no|some|lone|one |}
co := {| =|in|!=|!in |}
e := {| (List.header|n).(~?)(*|^)link |}
```

The *signature* (`sig`) declaration introduces a set of atoms and a user-defined type. A signature may declare *fields*, i.e., relations. `List` declares a set of list atoms; `one` makes the set *singleton*, i.e., have exactly 1 atom, which represents the list we are modeling. The field `header` declares a binary relation of type `List` × `Node`; `lone` declares `header` to be a *partial* function, i.e., each `List` atom maps to at most one `Node` atom. `Node` declares a set of nodes and introduces the field `link`, which is a partial function of type `Node` × `Node`. The predicate (`pred`) `Acyclic` introduces a named formula (which may have parameters).

The body of the predicate is a formula *sketch* with three different kinds of holes: `\Q,q\` (quantifier hole), `\CO,co\` (comparison operator hole), and `\E,e\` (expression hole). For the sake of illustrative example, we create several holes of different kinds (potentially more than a user would actually create), and we explicitly list all potential fragments for each hole. Each hole states the syntactic kind of the hole followed by an identifier, e.g., `E` followed by `e`. Each identifier refers to a regular expression (within `{| ... |}`, following [30]), e.g., `e` refers to `(List.header|n).(~?)(*|^)link`, which encodes a set of eight Alloy expressions in this example, including expressions `List.header.*link` and `n.~link`. *ASketch* extends the Alloy grammar [39] with these holes. The variable `n` is introduced by the quantifier (to be sketched) and is of type `Node`; the operator `=>` denotes logical implication.

The goal is to fill in the holes such that the formula constrains the nodes in the list to form an acyclic structure. Figure 1 graphically illustrates four test valuations for the model. Three valuations—$T0$, $T1$, and $T3$—are valid with respect to the expected acyclicity constraint. One valuation, $T2$, is invalid. Note that $T3$ is valid although $N1$ links to itself: $N1$ is not in the list, and the formula we are sketching should constrain only the nodes that are in the list, i.e., reachable from the `header`.

The user can provide the test valuations simply as Alloy predicates. For example, the following represent test valuations $T0$ and $T2$ from Figure 1:

```
pred Test0() {
  some L0: List {
    List = L0 and no header and no Node and no link and Acyclic[] }}
pred Test2() {
  some L0: List | some disj N0, N1: Node {
    List = L0 and header = L0->N0 and Node = N0+N1 and link = N0->N1 + N1->N0 and !Acyclic[] }}
```

The predicate `Test0` uses an existentially quantified (`some`) formula to assign a value to the `List` set. Using the Alloy keyword `no`, `Test0` declares the other signatures and relations to be empty. The predicate invocation `Acyclic[]` labels the valuation as *valid* for the expected acyclicity constraint. The predicate `Test2` uses existentially quantified formulas to assign values to the `List` and `Node` sets. The keyword `disj` requires the variables in the declaration to represent disjoint sets (i.e., unique nodes), the operator `->` denotes Cartesian product, the operator `+` denotes set union, and the predicate invocation `!Acyclic[]` labels the valuation as *invalid* for the expected acyclicity constraint.

Consider using *ASketch* to complete all five holes. Two are expression holes `\E,e\` with the same given regular expression assigned for the fragment space, and each expression hole has eight syntactically different expression fragments. Alloy also allows five quantifiers for `\Q,q\` (`all`, `no`, `some`, `lone`, and `one`) and four comparison operators for `\CO,co\` (`=`, `in`, `!=`, and `!in`). In total, there are $5 \times 4 \times 8 \times 4 \times 8 = 5,120$ candidate Alloy models. For our example, we use 8 test valuations to obtain the expected solutions (4 shown in Figure 1 plus 4 more). To complete the sketch, *ASketch* takes less than 1 second when solving the entire Alloy meta-model that encodes all 5,120 models and 8 valuations at once. Here is a solution *ASketch* finds:

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

The Alloy keyword `in` represents the subset, and `!` denotes logical negation. The operator `*` denotes reflexive transitive closure, and `^` denotes transitive closure. The expression `List.header.*link` represents the set of all nodes reachable from the list's header (following *zero* or more traversals of the field `link`). The expression `n.^link` represents the set of all nodes reachable from `n` (following *one* or more traversals of the field `link`). Thus, this universally quantified formula states that for any node that is in the list, the node is not reachable from itself, which correctly characterizes our expected acyclicity constraint.

Table 1. Supported fragments for non-recursively defined holes

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

3 *ASketch* Framework

We next present the *ASketch* grammar for Alloy models with holes and describe how *ASketch* determines which fragments complete the sketch to produce an Alloy model that satisfies all the given test valuations.

3.1 Input Language

The input to *ASketch* is an Alloy model with holes. For lack of space, we do not show the full grammar for *ASketch*'s input language, but it effectively extends the Alloy grammar with new syntactic constructs that represent holes. The current Alloy grammar is available at `http://alloy.csail.mit.edu/alloy/documentation/alloy4-grammar.txt`; we follow an older exposition [11] that included the semantics of the kernel Alloy language. Consider this part of the *ASketch* grammar:

```
quant ::= "all" | "no" | "some" | "lone" | "one" | "\Q," identifier "\"
expr ::= "*"expr | expr "+" expr | ... | "\E," identifier "\"
compareOp ::= "=" | "in" | "!=" | "!in" | "\CO," identifier "\"
formula ::= quant v ":" type "|" formula | ...
regExDecl ::= identifier ":=" "{|" regex "|}"
regex ::= nonSpecial | regex "?" | "(" regex ")" | regex regex | regex "|" regex
```

We extend `quant` so the quantifier can be a hole \\`Q,i`\\ where `Q` indicates the quantifier hole kind and `i` is an identifier that maps to a regular expression via `regExDecl`. The `expr` options include the expressions from Alloy, formed with unary (e.g., `*`) or binary operators (e.g., `+`), and we add a hole (\\`E,i`\\) that can replace an entire expression. Comparison operators include all operators from Alloy and also a hole \\`CO,i`\\. The `formula` options include the Alloy first-order logic formulas. `regExDecl` has the form `i:={|e|}` where `i` is referred from a hole and `e` is a regular expression. We follow the design of popular sketching system [13, 30, 32] that include a few regular expression operators: options (`e?`), concatenations (`e1 e2`), and choices (`e1 | e2`). `nonSpecial` is any character that Alloy supports except for `?`, `(`, `)`, and `|`; to use those, requires escaping them as `\(`, `\)`, and `\|`. Finally, *ASketch* generates all possible fragments that match `e` using a standard backtracking algorithm [20]. *ASketch* supports all fragments for non-expression holes, as shown in Table 1. Our current implementation requires an explicit regular expression for every hole, although a default could be set up such that non-expression holes implicitly get all possible fragments without listing them explicitly.

3.2 Solver-based sketching

ASketch reduces the sketching problem to a constraint-solving problem in the Alloy language itself, which is then solved by the Alloy Analyzer. Effectively, *ASketch* generates one *meta-model* in Alloy that encodes multiple potential solutions (i.e., concrete models) to the sketch. To represent the fragments for each hole, two constructs are added to the meta-model: (1) an Alloy *atom* that names a specific fragment for the hole, and (2) constraints that characterize the semantics of the different fragments for the sketch.

Because *ASketch* uses the Alloy tool-set itself to encode Alloy expressions and formulas, their semantics need not be explicitly modeled in Alloy; rather, they just need to be stated—indeed, the Alloy tool-set understands the semantics of Alloy. Therefore, we can use a shallow embedding of Alloy fragments in the model. Specifically, to represent the expression fragments, *ASketch* creates new Alloy *functions*, i.e., parameterized expressions. To represent the operator fragments, *ASketch* creates new Alloy *predicates*, i.e., parameterized formulas. Moreover, to encode multiple given test valuations in the same meta-model, *ASketch parameterizes* formulas with respect to user-defined relations, which are extracted out of their declaring *signatures* and added as new parameters. Our encoding allows constraining the model with respect to *all* valuation constraints at once—without causing an unnecessary increase in the number of propositional variables in the resulting SAT formula and without requiring higher-order solving [22].

We use the linked-list example from Section 2 to describe how *ASketch* sketches the body of a predicate and completes five holes of three kinds—quantifiers (`\Q,q\`), comparison operators (`\CO,co\`), and expressions (`\E,e\`). *ASketch* uses the following steps to create an Alloy meta-model whose solutions complete the sketch: (1) parameterize Alloy construct (Section 3.2.1); (2) create Alloy meta constructs to encode holes (Section 3.2.2); (3) translate test valuations to facts (Section 3.2.3); and (4) invoke the Alloy Analyzer to complete the holes (Section 3.2.4).

3.2.1 Parameterize Alloy constructs In the first step, *ASketch* parameterizes all predicates, functions, and facts. To parameterize an Alloy fact, *ASketch* first converts it to a semantically equivalent predicate. Without loss of generality, we only present how *ASketch* parameterizes predicates. The goal is to allow *multiple* test valuations to be encoded *in the same meta-model*. *ASketch* constructs a meta-model which includes (1) all signature declarations from the partial model, but *without* any of the declared relations, and (2) all predicates. Moreover, all predicates in the meta-model get additional parameters: one new parameter per signature and one new parameter per field; parameters that represent signatures have fresh variable names generated, whereas those that represent fields use the same names as in the partial model. In the body of the predicates, any reference to a declared signature is replaced by the corresponding fresh variable name.

For our acyclic linked-list example from Section 2, we get the following:

```
one sig List {}   sig Node {}
pred Acyclic(ls: one List, header: List -> Node, ns: set Node, link: Node -> Node) {
  \Q,q\ n: ns | n \CO,co\ \E,e\ => n \CO,co\ \E,e\ }
```

3.2.2 Create Alloy meta constructs to encode holes *ASketch* creates Alloy meta constructs that encode concrete values for every hole in Alloy predicates. We present how to encode only quantifier holes, comparison operator holes, and expression holes in Alloy predicates. The algorithm takes as inputs a mapping from expression holes to the corresponding expression fragments and a mapping from holes to *all Alloy variables* (sigs, fields, predicate parameters, `let`-bound variables, and quantified variables) in scope of the holes. The algorithm iterates over each Alloy predicate in the meta-model and updates the predicate body by recursively replacing *ASketch* holes with predicate/function calls, and creating and adding the predicate/function declarations to the meta-model. Note that any reference to a declared signature in the generated predicate/function is replaced by the corresponding fresh variable name as described in Section 3.2.1, e.g., `List` with `ls`.

After this step, *ASketch* constructs the following meta-model (note that the two comparison operator holes share the same operator fragments, and the two expression holes share the same expression fragments):

```
pred Acyclic(ls: one List, header: List -> Node, ns: set Node, link: Node -> Node) {
  q1[RQ1, ls, header, ns, link] }
abstract sig Q {}   one sig RQ1 in Q {}
one sig Q_All, Q_No, Q_Some, Q_Lone, Q_One extends Q {}
pred q1(h: Q, ls: one List, header: List -> Node, ns: set Node, link: Node -> Node) {
  h = Q_All => all n: ns | co2[RCO2, n, expr3[RE3, ls, header, ns, link, n]] =>
    co2[RCO4, n, expr3[RE5, ls, header, ns, link, n]]
  h = Q_No => no n: ns | co2[RCO2, n, expr3[RE3, ls, header, ns, link, n]] =>
    co2[RCO4, n, expr3[RE5, ls, header, ns, link, n]]
  ... }
abstract sig CO {}   one sig RCO2 in CO {}   one sig RCO4 in CO {}
one sig CO_Eq, CO_In, CO_NEq, CO_NIn extends CO {}
pred co2(h: CO, e1, e2: set univ) {
  h = CO_Eq => e1 = e2
  h = CO_In => e1 in e2
  ... }
abstract sig E3 {}   one sig RE3 in E3 {}   one sig RE5 in E3 {}
one sig E3_1, E3_2, E3_3, E3_4, E3_5, E3_6, E3_7, E3_8 extends E3 {}
fun expr3(h: E3, ls: one List, header: List -> Node,
          ns: set Node, link: Node -> Node, n: one Node): univ {
  (h = E3_1 => ls.header.*link else
  (h = E3_2 => n.^link else
  ... else none)) }
```

For quantifier holes, *ASketch* creates a unique *abstract* sig `Q` and declares 5 disjoint singleton sigs that represent all possible values for the hole (`all`, `no`, `some`, `lone`, and `one`). For each quantifier hole, *ASketch* translates the quantified formula to a predicate call. The predicate has the following parameters: (1) one parameter of the new abstract sig type that allows evaluating the predicate to one of the 5 quantifiers; and (2) one parameter for each variable in scope: signatures and fields from the original model, and optionally, predicate parameters, `let`-bound variables, and/or quantified variables in case of nested quantified formulas. The corresponding predicate declaration, `q1` in our example, is added to the meta-model. The predicate body is a conjunction of implications that model different quantified formulas corresponding to the hole. *ASketch* also introduces a result sig, `RQ1` in our example, that will obtain one of the 5 values (`Q_All`, `Q_No`, `Q_Some`, `Q_Lone` and `Q_One`) to represent the quantifier to fill in the hole.

For comparison operator holes, *ASketch* creates a unique *abstract* sig `CO` and declares 4 disjoint singleton sigs that represent all possible values for the hole (`=`, `in`, `!=`, and `!in`). Unlike for quantifier holes where each hole requires a new predicate, all comparison operator holes (of the same arity) can be encoded using a single predicate if they share the same set of fragments. *ASketch* creates a predicate, `co2` in our example, which encodes a formula that contains a comparison operator. The predicate contains 3 parameters: (1) one parameter of the new abstract sig type that allows evaluating the predicate to one of the 4 comparison operators (`CO_Eq`, `CO_In`, `CO_NEq`, and `CO_NIn`); (2) left operand; and (3) right operand. For each comparison operator hole, *ASketch* introduces a result sig, `RCO2` and `RCO4` in our example, similar as for quantifier holes. (*ASketch* treats the other non-expression holes similar to comparison operator holes, but we do not present details due to space limits.)

To model values of expression holes, *ASketch* creates one new *abstract* sig, `E3` in our example, for all holes that share the same set of expression fragments and declares k singleton sigs that partition the new sig, where k is the number of expression fragments for the corresponding expression hole, 8 in our example. *ASketch* also introduces result sigs, `RE3` and `RE5` in our example, that will obtain one of the k values to represent which fragment fills the hole. Next, *ASketch* creates an Alloy function that can select from these choices. The function has these parameters: (1) one parameter of the new abstract sig type that allows evaluating the function to one of the expression fragments based on the invocation context; and (2) one parameter for each Alloy variable in scope. The function body is a nested if-then-else expression where exactly one choice is true for any invocation, and the function evaluates to the value of the expression fragment corresponding to that choice.

3.2.3 Express test valuations as facts To complete the sketch with respect to the given test valuations (labeled as valid or invalid), *ASketch* automatically translates the test valuations (expressed as predicates in Section 2) to *facts*, which forces any solution that is created (in the final meta-model) to conform to all given valuations. Because valuations from different tests may contradict one another, *ASketch* uses Alloy's `let` construct to introduce the necessary names for sets and relations that are assigned values. Then, *ASketch* passes these sets and relations to the parameterized predicates (described in Section 3.2.1) so that the final sketched model satisfies all the tests at once. For example, `Test0` from Section 2 becomes the following fact:

```
fact Test0 {
  some L0: List {
    let ls = L0 | let header = none->none | let ns = none | let links = none->none |
      Acyclic[ls, header, ns, links] }}
```

3.2.4 Invoke Alloy Analyzer to complete holes The final meta-model consists of all pieces generated in sections 3.2.1, 3.2.2, and 3.2.3. *ASketch* invokes the Alloy Analyzer to execute an empty `run` command (`run {}`) on the final meta-model. The analyzer searches for possible valuations of the result `R` sigs so that they conform to all tests. In our example, `RQ1` evaluates to `Q_All`, `RCO2` to `CO_In`, `RE3` to `E3_1`, `RCO4` to `CO_NIn`, and `RE5` to `E3_2`. Finally, *ASketch* maps result values to the corresponding Alloy fragments and reports concrete values of all holes to the user, e.g., ⟨`all, in, List.header.*link, !in, n.^link`⟩ in our example. The completed, sketched model becomes this:

```
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 }
```

Our example used only 8 expressions, but realistic *ASketch* models may have hundreds of expressions, which results in much larger meta-models. Our experiments show that the above encoding technique still works relatively well even for a large number of expressions. It also works much better than all other meta-model encoding techniques we tried.

4 Experimental evaluation

We next present our experimental evaluation of *ASketch*. We use five small but intricate Alloy problems to derive 24 sketching models for evaluation (Section 4.1). We evaluate how much time *ASketch* takes to find complete Alloy models that satisfy all test valuations (Section 4.2).

4.1 Sketching problems

We use 24 sketches derived from five core Alloy models: *LinkedList* from Section 2, *BinaryTree* models the acyclicity constraint of a binary tree, *Contains* checks whether a list contains an element, *Remove* models removing an element from a list, and *Dijkstra* models Dijkstra's mutual exclusion algorithm.

For each core model, we picked one predicate to create several sketches by increasing the total number of holes in the body of the predicate, from left to right. This process enables us to systematically create model variants to explore how the number of holes affects our techniques. For example, for *LinkedList*, we identified 3 non-expression holes and 2 expression holes in the `Acyclic` predicate and produced these 5 variants:

```
\Q,q\ n: Node | n in List.header.*link => n !in n.^link         // LinkedList 1H
\Q,q\ n: Node | n \CO,co\ List.header.*link => n !in n.^link    // LinkedList 2H
\Q,q\ n: Node | n \CO,co\ \E,e\ => n !in n.^link                // LinkedList 3H
\Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ n.^link            // LinkedList 4H
\Q,q\ n: Node | n \CO,co\ \E,e\ => n \CO,co\ \E,e\              // LinkedList 5H
```

Our experiments are performed on a MacBook Pro running OS X El Capitan with 2.5 GHz Intel Core i7-4870HQ and 16GB of RAM.

4.2 *ASketch* results

Table 2 shows the results of *ASketch* for various sketching problems. The column *Model* shows the model variants for each core model; columns *#N* and *#E* show the number of non-expression holes and expression holes, respectively; the column *Search Space* shows the number of fragments combinations for all holes; and the columns *#Primary Vars*, *#Clauses*, and *Solving Time* show the number of primary variables, clauses, and solving time in seconds for the meta model, respectively. The *Search Space* is computed as the product of the number of fragments for each hole in the model. For example, if the *LinkedList* model with 5 holes has 1 quantifier hole with 5 fragments, 2 comparison operator holes with 4 fragments each, and 2 expression holes with 400 fragments each, then the sketching problem has a search space of $5 \times 4^2 \times 400^2 = 12,800,000 \cong 1.3e7$.

The columns 50, 100, 200, 300, and 400 show the number of expression fragments in the experiment, e.g., 50 means that we use 50 syntactically different expressions for each expression hole in the model variant. We generate regular expressions for expression holes using RexGen[1] [43] such that two properties hold. First, the set of expressions contains the expected solutions. Second, the larger set of expressions contains all expressions from the smaller set, e.g., the set of 100 expressions includes the set of 50 expressions and adds 50 more. We ensure the first property as follows. Suppose we have H expression holes and E expected expressions to fill the holes. We run RexGen to get X expressions and exclude E expected expressions from X expressions. Next, we run *ASketch* to find all solutions w.r.t. the test valuations and exclude any expression in the solutions that is non-equivalent to any of the E expected expressions. The idea is to remove all expressions that could lead to a solution that passes all tests but is incorrect. Then, to form a set of expressions with size Y (where Y is 50, 100, 200, 300, or 400), we sample the remaining expressions to obtain $Y - E$ expressions, and add the E expected expressions back .

Dijkstra has two expression holes with different variables in scope, so each expression hole uses a different set of expression fragments (but with the same number of expressions). Expression holes for each of *LinkedList*, *BinaryTree*, *Contains*, and *Remove* share the same set of expression fragments. In the experiments, we use 16 test valuations for each core model, and all model variants of the same core model share the same test suite. All experiment settings, with various fragments and test valuations, yield solutions that are semantically equivalent to the correct solutions.

If a sketch has no expression hole, then increasing the number of the expression fragments does not increase the search space, primary variables, or clauses in the generated meta-model. For example, *BinaryTree* model with 1 hole has only a comparison operator hole, and the search space (4), the number of primary variables (170), and clauses (7,957) remain unchanged as the number of expression fragments increases. If the sketch has expression holes, then the search space, primary variables, and clauses increase when we use more expression fragments. In our experiment, the search space goes up to 4.1e9 (*BinaryTree*), the number of primary variables goes up to 1420 (*Remove*), and the number of clauses goes up to 2.3e6 (*Dijkstra*). Overall these numbers show that the sketching problems are non-trivial.

[1] Note that RexGen can work in the mode where it prunes out equivalent expression fragments. We do not use that mode because we want to generate a large number of expression fragments for our experiments. All expressions that we generate are syntactically different but some may be semantically equivalent.

Table 2. *ASketch* results for finding a solution. Times are in seconds.

Model		#N	#E	Search Space					#Primary Vars					#Clauses					Solving Time				
				50	**100**	**200**	**300**	**400**	**50**	**100**	**200**	**300**	**400**	**50**	**100**	**200**	**300**	**400**	**50**	**100**	**200**	**300**	**400**
LinkedList	1H	1	0	5	5	5	5	5	138	138	138	138	138	6170	6170	6170	6170	6170	0.1	0.2	0.1	0.1	0.1
	2H	2	0	20	20	20	20	20	142	142	142	142	142	7397	7397	7397	7397	7397	0.2	0.2	0.2	0.2	0.2
	3H	2	1	1e3	2e3	4e3	6e3	8e3	192	242	342	442	542	6.3e4	1.1e5	2.3e5	3.6e5	5.1e5	2.8	4.5	88.6	267.9	141.0
	4H	3	1	4e3	8e3	1.6e4	2.4e4	3.2e4	196	246	346	446	546	6.5e4	1.2e5	2.4e5	3.8e5	5.3e5	3.2	5.2	91.1	286.6	150.1
	5H	3	2	2e5	8e5	3.2e6	7.2e6	1.3e7	246	346	546	746	946	1.1e5	2.1e5	4.4e5	7e5	1e6	6.7	32.5	252.9	574.6	759.1
BinaryTree	1H	1	0	4	4	4	4	4	170	170	170	170	170	8e3	8e3	8e3	8e3	8e3	0.1	0.1	0.1	0.1	0.1
	2H	1	1	2e2	4e2	8e2	1200	1600	220	270	370	470	570	6e4	1.1e5	2.2e5	3.5e5	4.8e5	0.6	1.1	3.1	5.7	5.6
	3H	2	1	8e2	1600	3200	4800	6400	224	274	374	474	574	6.1e4	1.1e5	2.2e5	3.5e5	4.9e5	0.5	1.4	2.9	3.6	12.3
	4H	2	2	4e4	1.6e5	6.4e5	1.4e6	2.6e6	274	374	574	774	974	8.6e4	1.7e5	3.6e5	5.8e5	8.4e5	1.4	5.6	27.6	36.9	265.7
	5H	3	2	1.6e5	6.4e5	2.6e6	5.8e6	1e7	278	378	578	778	978	8.6e4	1.7e5	3.6e5	5.8e5	8.4e5	1.8	7.0	30.3	47.5	127.4
	6H	3	3	8e6	6.4e7	5.1e8	1.7e9	4.1e9	328	478	778	1078	1378	1.1e5	2.3e5	4.9e5	8.1e5	1.2e6	1.4	37.7	138.7	515.1	709.3
Contains	1H	0	1	50	1e2	2e2	3e2	4e2	312	362	462	562	662	3.3e4	6.1e4	1.3e5	2.2e5	3.2e5	0.4	0.7	3.2	7.4	7.6
	2H	1	1	2e2	4e2	8e2	1200	1600	316	366	466	566	666	5.4e4	1e5	2.1e5	3.2e5	4.6e5	0.7	1.7	9.4	25.1	24.9
	3H	1	2	1e4	4e4	1.6e5	3.6e5	6.4e5	366	466	666	866	1066	7.7e4	1.5e5	3.3e5	5.4e5	7.9e5	2.0	15.6	103.3	397.0	1925.2
Remove	1H	0	1	50	1e2	2e2	3e2	4e2	267	317	417	517	617	4e4	7.3e4	1.5e5	2.5e5	3.6e5	0.2	0.5	1.1	2.2	4.4
	2H	1	1	150	3e2	6e2	9e2	1200	270	320	420	520	620	4.1e4	7.4e4	1.5e5	2.5e5	3.6e5	0.3	0.7	1.3	2.8	5.3
	3H	1	2	7500	3e4	1.2e5	2.7e5	4.8e5	320	420	620	820	1020	7.2e4	1.4e5	3e5	5e5	7.3e5	0.7	2.0	3.6	16.7	75.9
	4H	1	3	3.8e5	3e6	2.4e7	8.1e7	1.9e8	370	520	820	1120	1420	1e5	2.1e5	4.5e5	7.5e5	1.1e6	7.0	62.2	61.6	6903.8	19579.8
Dijkstra	1H	1	0	4	4	4	4	4	370	370	370	370	370	9348	9348	9348	9348	9348	0.1	0.1	0.0	0.0	0.0
	2H	1	1	2e2	4e2	8e2	1200	1600	420	470	570	670	770	9.2e4	1.4e5	2.6e5	3.9e5	5.4e5	1.1	2.1	4.8	24.2	21.9
	3H	2	1	1e3	2e3	4e3	6e3	8e3	404	454	554	654	754	9.2e4	1.4e5	2.6e5	3.9e5	5.4e5	1.1	2.2	4.9	21.6	22.1
	4H	3	1	5e3	1e4	2e4	3e4	4e4	409	459	559	659	759	9.4e4	1.5e5	2.6e5	4e5	5.4e5	1.9	4.2	6.0	40.9	41.3
	5H	4	1	2e4	4e4	8e4	1.2e5	1.6e5	413	463	563	663	763	9.8e4	1.5e5	2.7e5	4e5	5.5e5	1.7	2.9	14.3	26.9	46.9
	6H	4	2	1e6	4e6	1.6e7	3.6e7	6.4e7	463	563	763	963	1163	2.7e5	5.4e5	1.1e6	1.7e6	2.3e6	26.8	80.6	1053.1	4542.8	6535.0

The solving time depends on various factors, including the number of primary variables and clauses, the size of each clause, the complexity of the expression fragments, the search strategy of the SAT solver, etc. In general, the solving time increases with the size of the search space and the number of holes. However, there are exceptions. For example, in *LinkedList* with 4 holes, the solving time decreases as the size of expression fragments grows from 300 to 400. The reason is that multiple expression fragments are correct and equivalent. We cannot control how the Alloy Analyzer generates CNF clauses from the meta-model, so some solutions are found sooner than the others even if we increase the search space. Another exception is when *BinaryTree* goes from 4 holes to 5 holes using 400 expression fragments. Again, the solving time decreases as the number of holes increases. The reasons are that (1) adding an operator hole does not increase the number of primary variables or clauses by much; (2) it can make the sketching problem easier to solve as more equivalent correct solutions can be found; and (3) the Alloy Analyzer encodes the problem such that the solver is able to find the solution fast. Overall, *ASketch*'s encoding is relatively efficient and works well for large search spaces.

5 Related work

We introduce the first approach to sketching Alloy models. Program sketching [1, 13, 28–33] is a form of program synthesis, which is a mature yet active research topic [2, 5–7, 9, 17, 19, 21, 25, 28]. Researchers have proposed program synthesis techniques for a number of languages, including synthesis of logic programs, e.g., using inductive synthesis based on positive and negative examples [3]. However, prior work has not addressed the complexity of synthesis in the presence of quantifiers, transitive closure, relational operators, and more generally, formulas that express structurally complex properties, which are the focus of our work.

The Sketch system [30] takes as input a partial program in the Java-like Sketch language, and uses SAT and inductive synthesis in a counterexample-guided loop. Sketch requires users to provide generators for expression fragments for expression holes. The JSketch tool translates Java to Sketch to allow sketching Java programs [13]. Some tools focus on specific kinds of programs to sketch, such as PSketch for concurrent data structures [32].

Previous work on program synthesis has also used user-provided tests, albeit for imperative code, to guide synthesis. SyPet [5] introduced a novel use of Petri nets in synthesizing sequences of method invocations for complex APIs using tests. EdSketch [10] and EdSynth [44] introduced an optimized backtracking search for completing Java sketches using test executions for pruning. Test-Driven Synthesis iteratively builds a C# program such that it satisfies all tests [26]. Component-based synthesis builds programs by combining components from given libraries, e.g., work in this line used I/O oracles to synthesize loop-free programs [14].

Our approach also shares the spirit of storyboard programming, which uses user-provided graphical representations of data structures to synthesize imperative code that performs desired data structure manipulations based on the insight that it can be easier and more intuitive for a user to provide concrete data structure manipulations than to write the code [29]. Our test valuations make use of a similar insight.

An approach for creating Alloy models using instances was introduced by aDeryaft [15] in the spirit of Daikon [4] that uses a collection of known properties to check which hold with respect to given inputs. Alchemy [18] defined a translation to database update operations and integrity constraints. AUnit [37, 38] recently defined the concepts of test case, test execution, and model coverage for unit testing of Alloy models in the spirit of popular xUnit frameworks for imperative languages. AUnit has also enabled the adoption of other traditional imperative testing infrastructures to Alloy such as mutation testing [37,42]. The test valuations that ASketch uses in the context of synthesis follow AUnit's definition of a test case. *ASketch*'s solver-based approach for sketching also inspired a way to model state and state transitions in Alloy [36].

While this paper focuses on sketching for Alloy, one of the earliest approaches for helping Alloy users build their models correctly was based on identifying unsatisfiable cores in overconstrained models [27, 40, 41], which aids in automated debugging. More recent work introduced different strategies for scenario exploration for better understanding of the properties modeled [23, 24].

6 Conclusions and Future Work

We introduced *ASketch*, the first approach for sketching Alloy models. Given a model with holes and some (valid and invalid) valuations for the desired model, *ASketch* completes the given model with respect to the valuations. *ASketch* performs two key steps: it generates a pool of fragments (e.g., expressions) for each hole from user-provided regular expressions, and it creates a meta-model to explore the resulting space of candidate (completed) models to find a model that conforms to the valuations. An experimental evaluation using a suite of sketches shows that *ASketch* introduces a promising approach for sketching Alloy models. *ASketch* brings the spirit of traditional program sketching [1, 10, 13, 16, 28–33]—often regarded as the breakthrough approach in program synthesis for imperative and functional programs during the last decade—to a declarative, relational logic. We hope *ASketch* serves as a sound basis for a highly effective methodology for synthesizing Alloy models, which ultimately increases the use of analyzable models and leads to better software.

Future work can build on *ASketch* for solving other problems, such as automated debugging of faulty Alloy models. To illustrate, consider a model that is erroneously overconstrained. To repair it, first identify its unsat core using SAT to localize likely faulty expressions or formulas, and then create a sketch and complete it using *ASketch*. Future work can also evaluate the usability of *ASketch* via a user study; as common in sketching [30], we start first from the algorithmic foundations for sketching and leave actual user evaluations for later. An alternative to *ASketch*, which is a *solver-based* technique, is to employ an *enumeration-based* technique [34]; future work can rigorously compare the two techniques and combine them for a likely more effective synergistic approach.

Acknowledgements

We thank Manos Koukoutos and Viktor Kuncak for their comments on this work. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-1409423, CCF-1421503, CNS-1646305, CCF-1718903, and CNS-1740916.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD (2013)
2. Bodík, R., Jobstmann, B.: Algorithmic program synthesis: Introduction. STTT (2013)
3. Deville, Y., Lau, K.K.: Logic program synthesis. The Journal of Logic Programming (1994)
4. Ernst, M.D.: Dynamically Discovering Likely Program Invariants. Ph.D. thesis, University of Washington Department of Computer Science and Engineering (2000)
5. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex APIs. In: POPL (2017)
6. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: PLDI (2015)
7. Galenson, J., Reames, P., Bodik, R., Hartmann, B., Sen, K.: CodeHint: Dynamic and interactive synthesis of code snippets. In: ICSE (2014)
8. Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: Efficient SAT-based bounded verification using symmetry breaking and tight bounds. TSE (2013)
9. Gvero, T., Kuncak, V., Piskac, R.: Interactive synthesis of code snippets. In: CAV (2011)
10. Hua, J., Khurshid, S.: EdSketch: Execution-driven sketching for Java. In: SPIN (2017)
11. Jackson, D.: Alloy: a lightweight object modelling notation. TSE (2002)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
13. Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: JSketch: Sketching for Java. In: FSE (2015)
14. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE (2010)
15. Khurshid, S., Malik, M.Z., Uzuncaova, E.: An automated approach for writing Alloy specifications using instances. In: ISoLA (2006)
16. Kneuss, E., Koukoutos, M., Kuncak, V.: Deductive program repair. In: Kroening, D., Păsăreanu, C. (eds.) CAV (2015)
17. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA (2013)
18. Krishnamurthi, S., Fisler, K., Dougherty, D.J., Yoo, D.: Alchemy: Transmuting base Alloy specifications into implementations. In: FSE (2008)
19. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI (2010)
20. Larson, E., Kirk, A.: Generating evil test strings for regular expressions. In: ICST (2016)

21. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: Helping to navigate the API jungle (2005)
22. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: A general-purpose higher-order relational constraint solver. In: ICSE (2015)
23. Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": Enriching scenario exploration with provenance. In: FSE (2017)
24. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: ICSE (2013)
25. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. In: PLDI (2015)
26. Perelman, D., Gulwani, S., Grossman, D., Provost, P.: Test-driven synthesis. PLDI (2014)
27. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: ASE (2003)
28. Singh, R., Gulwani, S.: Predicting a correct program in programming by example. In: CAV (2015)
29. Singh, R., Solar-Lezama, A.: Synthesizing data structure manipulations from storyboards. In: FSE (2011)
30. Solar-Lezama, A.: Program Synthesis by Sketching. Ph.D. thesis, University of California, Berkeley (2008)
31. Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Sketching stencils. PLDI (2007)
32. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: PLDI (2008)
33. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
34. Sullivan, A.: Automated Testing and Sketching of Alloy Models. Ph.D. thesis, University of Texas at Austin (2017)
35. Sullivan, A., Wang, K., Khurshid, S.: AUnit: A test automation tool for Alloy. In: ICST (2018)
36. Sullivan, A., Wang, K., Khurshid, S., Marinov, D.: Evaluating state modeling techniques in Alloy. In: SQAMIA (2017)
37. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST (2017)
38. Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a test automation framework for Alloy. In: SPIN (2014)
39. Team, A.: `http://alloy.mit.edu/alloy/documentation/alloy4-grammar.txt`
40. Torlak, E., Chang, F.S.H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: FM (2008)
41. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS (2007)
42. Wang, K., Sullivan, A., Khurshid, S.: MuAlloy: A mutation testing framework for Alloy. In: ICSE (2018)
43. Wang, K., Sullivan, A., Koukoutos, M., Marinov, D., Khurshid, S.: Systematic generation of non-equivalent expressions for relational algebra. In: ABZ (2018)
44. Yang, Z., Hua, J., Wang, K., Khurshid, S.: Test execution driven synthesis of API sequences with conditionals and loops. In: ICST (2018)
45. Zave, P.: Using lightweight modeling to understand Chord. SIGCOMM Comput. Commun. Rev. (2012)