Towards Automated Input Generation for Sketching Alloy Models

Ana Jovanovic
The University of Texas at Arlington
Arlington, TX, USA
ana.jovanovic@mavs.uta.edu

Allison Sullivan
The University of Texas at Arlington
Arlington, TX, USA
allison.sullivan@uta.edu

ABSTRACT

Writing declarative models has numerous benefits, ranging from automated reasoning and correction of design-level properties before systems are built, to automated testing and debugging of their implementations after they are built. Alloy is a declarative modeling language that is well suited for verifying system designs. While Alloy comes deployed in the Analyzer, an automated scenario-finding tool set, writing correct models remains a difficult and error-prone task. *ASketch* is a synthesis framework that helps users build their Alloy models. *ASketch* takes as an input a partial Alloy models with holes and an AUnit test suite. As output, *ASketch* returns a completed model that passes all tests. *ASketch*'s initial evaluation reveals *ASketch* to be a promising approach to synthesize Alloy models. In this paper, we present and explore *SketchGen*^{2}, an approach that looks to broaden the adoption of *ASketch* by increasing the automation of the inputs needed for the sketching process. Experimental results show *SketchGen*^{2} is effective at producing both expressions and test suites for synthesis.

CCS CONCEPTS

• **Software and its engineering** → *Formal software verification.*

ACM Reference Format:
Ana Jovanovic and Allison Sullivan. 2022. Towards Automated Input Generation for Sketching Alloy Models. In *International Conference on Formal Methods in Software Engineering (FormaliSE'22), May 18–22, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524482.3527651

1 INTRODUCTION

As software pervades our society and lives, and software failures become increasingly costly, there is a growing need to leverage software models to improve the quality of software systems. Alloy [13] is a well-known modeling language that has been used in both academic and industrial settings [9, 14, 24, 50]. Alloy models are declarative and consist of relational, first-order logic formulas. Two key strengths of Alloy are its expressive notation, with support for operators like transitive closure, that allows for succinctly writing complex structural properties, and the Analyzer, its automated analysis engine that uses off-the-shelf SAT solvers to reason over properties with respect to a user-defined *scope*, i.e., bound on the universe of discourse. The Analyzer finds *instances*, which are assignments to the sets and relations of the model, such that the invoked formulas are true. The instances discovered by the Analyzer have been used to validate software designs [14, 26, 29, 39, 40], to test and debug code [10, 11, 15, 23], to repair program states [30, 49] and to synthesize security attacks for hardware architectures [4, 42, 43].

In the end, Alloy models can only help improve system reliability if they are themselves correct. Unfortunately, writing correct Alloy models is hard, especially for beginning users but even for advanced users. In particular, reasoning about the correctness of constraints in the presence of nested formulas and quantification requires much care. Therefore, to help users write correct models from the start, prior work introduces *ASketch*, an automated sketching framework for Alloy [48]. *ASketch* takes as input: (1) a partial model with user specified holes, (2) a generator which outlines the valid substitutions into each hole and (3) a test suite which outlines the expected behavior of the model. As output, *ASketch* produces a completed model that passes all tests. Specifically, *ASketch* builds an Alloy meta-model that encodes all the possible candidate models and the test suite and then uses Alloy's SAT backend to find a solution.

ASketch's initial evaluation revealed *ASketch* to be a promising start towards building correct from construction models, which in turn, can lead to correct from construction systems. However, there are a few barriers that limit the adoption of *ASketch*. First, *ASketch* requires guidance from the user to handle expression holes by having the user provide a regular expression that is used to generate all possible substitutions into the hole. However, users relying on *ASketch* may not be able to form a regular expression that both adheres to Alloy's grammar rules and is robust enough to contain at least one solution. Second, *ASketch* utilizes user provided test suites to outline the expected behavior of the model. As the complexity of the sketch increases, so does the need for a more expansive test suite. Unfortunately, the two automated test generation techniques for Alloy are white box and rely on information about the formula itself in order to produce tests. Therefore, these techniques can not be used to create tests for *ASketch*.

In this paper, we present *SketchGen*^{2}, a framework that looks to tackle these problems and further automated the sketching process. Specifically, *SketchGen*^{2} automatically creates two inputs for *ASketch*. First, *SketchGen*^{2} leverages *RexGen* [47], a generator for semantically non-equivalent relational expressions, to generate candidates to fill expression holes. Second, while generating expressions, *SketchGen*^{2} builds up a test suite intended for use by *ASketch*. To achieve this, *SketchGen*^{2} interweaves two of *RexGen*'s expression generation strategies, *Modulo-Instance Pruning*, which prunes expressions that are equivalent with respect to a test suite, and *Dynamic Pruning*, which prunes expressions that are equivalent with respect to a scope. *SketchGen*^{2} mitigates the trade-offs between the two strategies by incrementally building a test suite that ensures

FormaliSE'22, May 18–22, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9287-7/22/05.
https://doi.org/10.1145/3524482.3527651

```
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 |}
```

Hole 'e' Generator Values	
List.header.*link	n.*link
List.header.^link	n.^link
List.header.~*link	n.~*link
List.header.~^link	n.~^link

Figure 1: *ASketch* Input for a Singly-Linked List

the collection of expressions produced by *Modulo-Instance Pruning* equals that of *Dynamic Pruning*. Our experimental evaluations show that *SketchGen*^{2} is effective at generating inputs for sketching models and is well-suited to strengthen an initial, small test suite.

This paper makes the following contributions:

Expression Generation for Sketching: We introduce a new framework for efficiently generating all non-equivalent expressions up to a given bound, which leverages *Dynamic Pruning* and *Modulo-Instance Pruning* from *RexGen*.

Test Generation for Sketching: We introduce a new AUnit test input generation technique that can work in a sketching environment.

Experiments: We present an experimental evaluation with small, but intricate Alloy formulas. We demonstrate how *SketchGen*^{2} mitigates scalability concerns compared to directly using *RexGen* as input to *ASketch*.

Open Source: We release a prototype of *SketchGen*^{2} and a our collection of model sketches so researchers can use them in the future. The repo is available at https://SketchGen.github.io.

2 BACKGROUND

In this section, we present an example sketch to introduce key concepts of Alloy, *ASketch* and *RexGen*.

2.1 *ASketch*

To illustrate how *ASketch* works, consider the sketch of a singly-linked acyclic list model in Figure 1. In Alloy, the *signature* (`sig`) declaration introduces a named set of atoms, which creates a user-defined type. Therefore, the signature `List` introduces a named set of list atoms and the addition of the keyword `one` restricts this set to be a singleton set, i.e. there is always exactly one `List` atom. A signature may optionally declare *fields*. `List` introduces the field `header` as a binary relation of the type `List` × `Node`. The addition of the keyword `lone` makes `header` a *partial* function, i.e., each `List` atom maps to at most one `Node` atom. Likewise, the signature `Node` establishes a named set of node atoms and introduces the field `link` as a partial function of type `Node`×`Node`. The predicate (`pred`) `Acyclic` introduces a named formula which can be invoked elsewhere.

The body of the `Acyclic` 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). *ASketch* extends the Alloy grammar [41] with these holes. 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 [34]). To illustrate, in this example, the generator for 'e' is a regular expression that encodes eight different Alloy expressions, outlined in the bottom of Figure 1. The variable `n` is introduced by the quantifier (to be sketched) and is of type `Node`.

For this example, the goal of *ASketch* is to find a substitution into each hole in the sketch, such that the formula ends up representing the concept: "any node in the list is not reachable from itself." To determine if this behavior is met, *ASketch* uses AUnit test suites as a way to relay expectations. An AUnit test consists of two components: (1) a valuation, which is an assignment to all the sets and relations of the model, and (2) a label, which indicates whether the associated valuation should be allowed (valid) or prevented (invalid) by a given Alloy command. Figure 2 graphically illustrates five test valuations for our singly linked list model. Three valuations—$T0$, $T1$, and $T4$—are valid and two valuations—$T2$ and $T3$—are invalid with respect to the to be sketched `Acyclic` constraint. *ASketch* determines that a given candidate model, a complete model being evaluated as a potential solution, is correct if the candidate model passes all tests.

Consider using *ASketch* to complete all five holes. The two expression holes `\E,e\` use the same regular expression to create the fragment space, which expands into 8 unique expressions. For the operator holes, the fragments depicted capture all possible substitutions allowed by the Alloy grammar. In particular, there are 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, 120 ($5 \times 4 \times 8 \times 4 \times 8$) candidate Alloy models. To run this example, we use 12 test cases to outline expected behavior (5 shown in Figure 2 plus 7 more generated by *SketchGen*^{2} and shown in Figure 3). To complete the sketch, *ASketch* takes less than 1 second when solving the entire Alloy meta-model that encodes all 5,120 possible models and 12 test cases at once. Here is a solution *ASketch* finds:

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

The Alloy keyword '`all`' represents universal quantification, '`in`' represents the subset, the operator '`.`' represents relational join, the operator '`*`' represents reflexive transitive closure, and the operator '`^`' represents transitive closure. Thus, this universally quantified formula states that "for all nodes, if a node is in the list, then that node is not reachable from itself following one or more traversals down its link relation."

2.2 Challenge: Automatically Generating Expressions for *ASketch*

Using only a regular expression to fill expression holes is a tradeoff. The regular expression helps keep the search space of possible sketches tractable. However, not only does the regular expression need to generate valid Alloy expressions, it also needs to generate an Alloy expression that can successfully complete the sketch. This can be a high burden for a new Alloy user, who is more likely to adopt *ASketch*. Currently the task of providing an expression fragment list can be automated using *RexGen* [47]. *RexGen* is a generator that produces relational expressions up to a user provided bound on

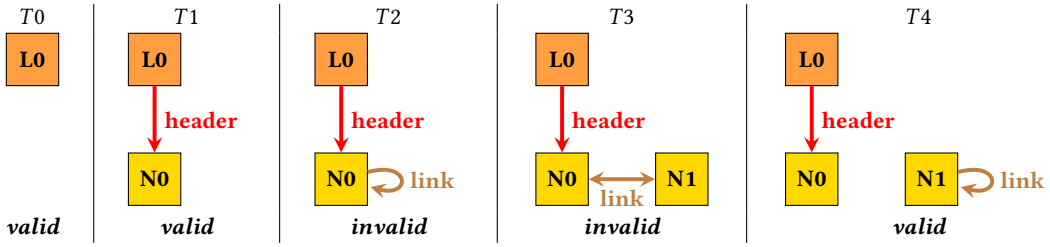


Figure 2: Five test valuations shown graphically: $T0$, $T1$, and $T4$ are valid, while $T2$ and $T3$ are invalid for acyclicity. $L0$ is the list atom; $N0$ and $N1$ are node atoms.

the cost of the expression. Expression generation happens bottom up starting with a cost of 1 and building to larger costs. To avoid generating lists that are too large to be useful, *RexGen* offers three automatic pruning modes: (1) *Static Pruning* directly prunes from generation many equivalent expressions based on a suite of known equivalence rules; (2) *Dynamic Pruning* uses the Analyzer during generation to prune equivalent expressions; and (3) *Modulo-Instance Pruning* allows the user to provide AUnit test cases, and prunes an expression if it is equivalent to some generated expression with respect to all given test cases (even if not equivalent over some other unseen test cases [2]).

As an example, consider using the different pruning methods to determine whether to keep or prune the following expressions for hole e: `header.^link` and `header.^~link`. *Static Pruning* would generate and keep both expressions, as there is no known equivalence rule, such as commutativity, that would eliminate one expression with respect to the other. *Dynamic Pruning* would use the following Alloy command to determine if the two expressions are equivalent:

```
check { header.^link = header.^~link } for 3
```

In this case, the Analyzer would find a counterexample; therefore, *Dynamic Pruning* would view the two expressions as not equivalent and keep both. If we consider the five test cases in Figure 2, *Modulo-Instance Pruning* would evaluate both expressions across all five tests resulting in the following values:

Expression	$T0$	$T1$	$T2$	$T3$	$T4$
`header.^link`	∅	∅	{L0->N0}	{L0->N0, L0->N1}	∅
`header.^~link`	∅	∅	{L0->N0}	{L0->N0, L0->N1}	∅

Thus, *Modulo-Instance Pruning* would view these two expressions as equivalent, prune the higher cost expression (`header.^~link`) and keep the lower cost expression (`header.^link`).

For our singly-linked list model, if the user invokes *RexGen* to generate expressions up to a cost of 6, which is need to produce the expression `List.header.*link` from the oracle solution, *RexGen* generates 214 expressions with *Static Pruning*, 107 with *Dynamic Pruning*, and 107 with *Modulo-Instance Pruning* (using all 12 test cases). For this model and corresponding test suite, the lists produced by *Dynamic Pruning* and *Modulo-Instance Pruning* are equivalent due to the strength of the test suite, although this is not guaranteed. The generation time for *Static Pruning* is less than 1 second, for *Dynamic Pruning* is 34 seconds, and for *Modulo-Instance Pruning* is 5 seconds. To solve the sketch using these expressions, *ASketch* takes 145 seconds using *Static Pruning* and takes 9 seconds using both *Dynamic Pruning* and *Modulo-Instance Pruning*. All together, this produces a total runtime (expression generation + sketching) of 146 seconds, 43 seconds, and 13 seconds for *Static Pruning*, *Dynamic Pruning* and *Modulo-Instance Pruning* respectively.

As sketches become more complex, using *RexGen* to fill expression holes for *ASketch* becomes difficult. While *Dynamic Pruning* creates an optimal list of expressions, *Dynamic Pruning* is time intensive. Specifically, *Dynamic Pruning* requires multiple invocations of a SAT solver to resolve all of the Alloy "check" commands needed to determine if an generated expression should be kept. Accordingly, previous experiments [47] have shown that *Dynamic Pruning* takes on average 2288.4× longer than *Static Pruning* and 31.0× longer than *Modulo-Instance Pruning*. However, these same experiments show that *Static Pruning* generates a list that is 1.5× larger than *Dynamic Pruning* and 2.7× larger than *Modulo-Instance Pruning*. The size of the expression list can have an *exponential* impact on the search space when more than one expression hole is present in the sketch. As an example, consider our example singly-linked list model. *Static Pruning* creates a search space of over 3.6 million candidate models while *Dynamic Pruning* creates a search space of just 915,920 candidate models, which is 4× reduction in size. With all these tradeoffs taken into account, *Modulo-Instance Pruning* is often used as a compromise between the other two generation strategies. Yet, *Modulo-Instance Pruning* has its own tradeoff: if the test suite is not robust enough, the set of expressions produced will aggressively over prune non-equivalent expressions.

2.3 Challenge: Quality of the Test Suite

As with any synthesis technique that is based on test suites, *ASketch* is dependent on the quality of the test suite used to outline expected behavior. If the test suite is too weak, *ASketch* can find a plausible solution, a completed model that passes all tests but does not match the end user's expectation. A user would ideally want to run *ASketch* with a robust test suite that produces only correct sketches. In practice, this can be hard. As seen in our evaluation in Section 4.1, the possible search space of some sketches is immense, e.g. `remove` has 5.7×10^{11} candidate models. This can require an extensive test suite to avoid plausible solutions.

While Alloy's known test generations cannot work on a sketch, a key advantage of sketching Alloy models is that we can use the Alloy language itself to alleviate concerns about plausible solutions and increase the quality of the test suite. Namely, we can search

for two solutions, and then use the Analyzer to check if the solutions are equivalent. If they are not, the Analyzer will produce a counterexample, which can then be turned into an additional test case. However, there are some limitations to this approach. First, it requires searching for multiple potential solutions, which can significantly increase the runtime of *ASketch*. In the worse case, if there is only one valid solution in the search space, then *ASketch* would end up exploring the entire search space, even if the solution was found in the beginning. Second, there is no clear stopping condition. For instance, we could check the first 'X' solutions, but there is no way to know in advance what value to set 'X' to for every model and this value can vary significantly depending on the quality of the initial test suite. Third, *ASketch* is not well suited for incremental analysis. When a new test is added, the meta-model is extended to encode the new test and then the SAT solver is re-invoked. Unfortunately, the SAT solver does not remember which candidate solutions it had already eliminated in the previous run. This is crucial, as all the previously eliminated candidate solutions are still invalid: simply adding a new test will not make an eliminated candidate solution now pass the test(s) it previously failed.

SketchGen^2 is designed to address both of these challenges. By design, *SketchGen*^{2} runs a series of smaller dynamic pruning problems over pre-partitioned sets of expressions, enabling the approach to avoid *Dynamic Pruning*'s scalability issues. In addition, since *SketchGen*^{2}'s expression list is equivalent to *Dynamic Pruning*'s, *SketchGen*^{2} does not overprune expressions the way *Modulo-Instance Pruning* can and does not miss equivalences the way *Static Pruning* can. While generating expressions, *SketchGen*^{2} also strengthens the user's initial test suite. Importantly, every new test created by *SketchGen*^{2} distinguishes between at least two expressions, which contributes to the ability of the new test case to eliminate candidate models and reduce the likelihood of discovering a plausible solution.

3 TECHNIQUE

In this section, we introduce *SketchGen*^{2}, a framework for automatic input generation for *ASketch*. We first present how *SketchGen*^{2} intertwines *Modulo-Instance Pruning* and *Dynamic Pruning*. Then, we step over how the impact of creating a new test is resolved.

3.1 Combining *Modulo-Instance Pruning* and *Dynamic Pruning*

A naive approach would require running both *Modulo-Instance Pruning* and *Dynamic Pruning* in their entirety, and then using the difference in the sets produced to expand the test suite. However, the motivation behind *Modulo-Instance Pruning* is to eliminate the high runtime overhead of *Dynamic Pruning*. Our key insight is to first use *Modulo-Instance Pruning* to partition the space of candidate expressions into an initial set of equivalence classes. The expressions in each equivalence class may truly be equivalent, or there may be some test case not currently in the test suite that distinguishes their behavior. However, expressions in one equivalence class are guaranteed to not be equivalent to any expressions in another equivalence class. Therefore, we can reduce the burden of *Dynamic Pruning* by only dynamically checking the equivalence of expressions placed in the same equivalence class by *Modulo-Instance*

Algorithm 1: *SketchGen*^{2} Expression and Test Generation

Input: Parsed Alloy model *module*, Map of representative expression to equivalent expressions *equivClasses*.
Output: Non-equivalent expression list and an AUnit test suite.

```
1  // Initialize helper variables
2  int index = 0
3  ArrayList<Expr> toCheck = equivClasses.keySet()
4  ArrayList<TextCase> newTests = new ArrayList<TestCase>()
5  while index < toCheck.size() do
6      Expr classRep = toCheck.get(index)
7      ArrayList<Expr> exprs = equivClasses.get(classRep)
8      ArrayList<Expr> skip = new ArrayList<Expr>()
9      foreach Expr curr : exprs do
10         // Did a new test move this expression to diff class?
11         if skip.contains(curr) then break
12         // Check for equivalence dynamically
13         Command equivCheck = genCmd(classRep, curr, module)
14         A4Solution sol = module.executeCmd(equivCheck)
15         if sol.satisfiable() // Not equivalent then
16             // Create new test case using counterexample
17             TestCase test = new TestCase(sol, genLabel())
18             newTests.add(test)
19             toCheck.add(curr)
20             equivClasses.put(curr, new ArrayList<Expr>())
21             equivClasses.get(classRep).remove(curr)
22             // Enforce the impact of this test on the current class
23             ArrayList<Expr> temp = new ArrayList<Expr>()
24             temp.addAll(equivClasses.get(classRep)
25             equivClasses.get(classRep).clear()
26             ArrayList<Expr> classOpt = new ArrayList<Expr>()
27             classOpt.add(classRep), classOpt.add(curr)
28             foreach Expr expr : temp do
29                 boolean unique = true
30                 String result1 = getExprValue(expr,test)
31                 for i ← 0 to classOpt.size() do
32                     String result2 =
                        getExprValue(classOpt.get(i),test)
33                     if result1.equals(result2)) then
34                         unqiue = false
35                         equivClasses.get(classOpt.get(i)).add(expr)
36                         // Flag that this expr moved to a new class
37                         if i > 0 then skip.add(expr)
38                         break
39                 if unique // This expression forms a new class then
40                     toCheck.add(expr)
41                     equivClasses.put(expr, new ArrayList<Expr>)
42                     classOpt.add(expr)
43                     // Flag that this expr moved to a new class
44                     skip.add(expr)
45             // Enforce the impact of this test on remaining classes
46             for idx gets index + 1 to toCheck.size() do
47                 updateEquivClass(equivClasses, toCheck.get(idx))
48     index++
49 return newTests, equivClasses.keySet()
```

Pruning. Moreover, the counterexamples produced when running these narrower *Dynamic Pruning* executions can strengthen the original test suite, so that the test suite is now capable of detecting the non-equivalence between the two expressions. To illustrate how this works, Algorithm 1 shows the details of *SketchGen*^{2}.

To start, *SketchGen*^{2} executes after a regular *Modulo-Instance Pruning* execution of *RexGen*. As input, *SketchGen*^{2} takes the parsed Alloy model (`module`) and a map outlining the equivalence classes created by *RexGen* (`equivClasses`), which maps a representative expression to a list of all expressions that are considered equivalent to this representative expression. Next, the first few lines of the algorithm initialize helper variables: `toCheck` is a list of all representative expressions, `index` tracks which equivalence class is being evaluated from `toCheck`, and `newTests` stores any new tests generated by *SketchGen*^{2}.

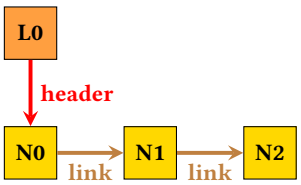
After initializing the variables, the remainder of the algorithm looks to implement the narrowed series of *Dynamic Pruning* executions over the equivalence classes found by *Modulo-Instance Pruning*. First, an outer while loop (line 5) iterates until all equivalence classes captured in `toCheck` have been explored. If the original test suite is inadequate, and *Modulo-Instance Pruning* over-pruned expressions, the list `toCheck` will get updated with any newly discovered equivalence classes. For each equivalence class, *SketchGen*^{2} first creates a series of local variables: `classRep` stores the represenative expression depicting the current equivalence class, `exprs` stores a list of all expressions currently believed to be equivalent to the representative expression, and `skip` stores a list of expressions from the current class that have been moved to a new class due to the creation of a new test.

Then, the for loop starting on line 9 evaluates each expression (`curr`) in the equivalence class, to determine if the expression is truly equivalent to representative expression (`classRep`). First, *SketchGen*^{2} checks if `curr` is in `skip` (line 11). If this check is true, then a new test has already illustrated `curr` is not equivalent to the class representative; therefore, *SketchGen*^{2} skips evaluating this expression and moves onto the next expression in the class. If this check is false, *SketchGen*^{2} uses the SAT solver to dynamically check if the two expressions are equivalent with respect to a given scope (lines 13-14). If the call is unsatisfiable, then the expression remains pruned. However, if this call to the SAT solver is satisfiable; then, the two expressions are actually not equivalent. As a result, the counterexample found by the SAT solver, which highlights the difference in behavior between the two expressions, gets turned into a new test case and `curr` (1) gets moved to its own equivalence class, (2) gets added to the list of equivalence classes to check and (3) gets removed from `classRep`'s equivalence class (lines 21 - 27).

To illustrate, consider the following extended equivalence class from our example in Section 2: ec_i=⟨`header.^link`, {`header.^~link`, `header.link.^link`}⟩ where `header.^link` is the representative expression of ec_i and {`header.^~link`, `header.link.^link`} is the set of expressions that are equivalent to the representative expression across all tests ($T0$ - $T4$ from Figure 2). To determine if the the first expression in the list is equivalent representative expression, we make the following dynamic check:

```
check { header.^link = header.^~link } for 3
```

The Analyzer produces the following counterexample, displayed graphically below:



This counterexample depicts a list with 3 nodes and no cycles. As a result, this counterexample would be labeled *valid*, turning this counterexample into test $T5$ (following the existing tests in Figure 2). Since this new test case reveals `header.^~link` is different from ec_i's representative expression, the expression `header.^~link` gets placed in its own equivalence class, ec_j, and added to the pool of classes to be checked.

3.2 Impact of New Tests

Since *SketchGen*^{2} may generate new tests, the equivalence class an expression belongs to may change as *SketchGen*^{2} executes, which is what the remainder of Algorithm 1 focuses on. Specifically, *SketchGen*^{2} checks if the remaining expressions in the equivalence class still hold the same behavior as the representative expression, `classRep`, in the presence of the new test. To achieve this, *SketchGen*^{2} first gathers all of the expressions in the current equivalence class into a temporary list (`temp`) (lines 23-24) and then resets the equivalence class (line 25). The variable `classOpt` is used to maintain a list of all the equivalence classes that the expressions captured in `temp` can get sorted into and is expanded when a new equivalence class is discovered.

To redistribute the expressions in `temp`, *SketchGen*^{2} loops over each expression (line 28) to determine if the expression is represented by an expression in `classOpt` (lines 31 - 38) or is a new equivalence class (lines 39 - 44). *SketchGen*^{2} uses the helper method `getExprValue` to determine an expression's evaluation over a given test case, which uses *Modulo-Instance Pruning*'s memoization infrastructure to avoid re-evaluating the same expression over the same test case. If *SketchGen*^{2} determines that an expression has moved out of the current equivalence class, the expression gets flagged (lines 37, 44) so that these expressions are not dynamically checked due to the outer for loop on line 9, which both avoids an unnecessary invocation of the SAT solver and prevents the generation of duplicate test cases.

To illustrate, for ec_i, this means running *Modulo-Instance Pruning* to determine what equivalence class – either ec_i, ec_j or neither – `header.link.^link` is in now that $T5$ needs to be accounted for. Over $T5$, the three expressions from ec_i resolve to the following:

```
header.^link = {L0->N0, L0->N1}
header.^~link = {}
header.link.^link = {L0->N0}
```

This reveals that "`header.link.^link`" is not equivalent to either expression; therefore, `header.link.^link` gets put into its own equivalence class, ec_k. If there were any remaining expressions in ec_i, then they would also be checked to see if they belong to one of the three known equivalence classes (ec_i, ec_j, or ec_k) or if the expression is also different over $T5$, resulting in another equivalence class.

Table 1: Basic information of models.

Model	#Sig	#Rel	#PVar	Scope
bempl	6	3	38	5
btree	2	2	24	3
contains	3	3	27	3
dll	2	4	72	3
dijkstra	3	1	57	3
fsm	2	3	40	5
grade	5	4	48	4
graph	1	1	30	3
remove	3	6	48	3
sll	2	2	15	3

The new tests do not just impact the current equivalence class, but also impact all remaining unchecked equivalence classes. Therefore, lines 46-47 reshape the remaining equivalence classes by invoking the helper method `updateEquivClasses`. The method `updateEquivClasses`'s implementation is nearly identical to the steps to re-partition the active equivalence class (lines 23 - 44). The only difference is that `updateEquivClasses` does not need to flag any expressions to skip. While adding tests may change unexplored equivalence classes, *SketchGen*^{2} does not need to backtrack and re-check any previously explored equivalence classes, as those will have already been shown to be full of equivalent expressions using *Dynamic Pruning*. Therefore, `updateEquivClasses` is only called for the equivalence classes after the current index location. Once all equivalence classes are checked, including any newly identified ones, *SketchGen*^{2} has produced a set of expressions equivalent to *Dynamic Pruning*'s expression list as well as a more robust test suite. *SketchGen*^{2}'s output is designed to be integrated with *ASketch*: the expressions produced can be directly used as is, while the user does need to first provide an oracle for the test cases.

4 EVALUATION

We evaluate *SketchGen*^{2} on 10 Alloy models, previously used to evaluate *ASketch* [45]. All experiments were performed on Ubuntu 20.04.2 LTS with 1.8 GHz Intel Core i7-10510U and 16GB of RAM.

4.1 Set Up

The models used in the evaluation include: an access control model for entering rooms (**bempl**), a binary tree (**btree**), list operation contains (**contains**), dijkstra's deadline prevention (**deadlock**), a doubly-linked list (**dll**), a finite state machine (**fsm**), a process control model for grading assignments (**grade**), a connected graph (**graph**), list operation remove (**remove**) and a singly-linked list (**sll**). For each model, the authors select a predicate in the model and abstract the entire formula into a sketch, excluding any attributes of the formula *ASketch* does not currently support.Table 1 shows the basic information of these models. **Model** is the name. **#Sig** is the number of signatures declared in each model. **#Rel** is the number of relations declared in each model. **#PVar** is the number of primary variables when we run an empty command (`run {}`) without test-specific constraints; it represents the basic complexity of signature declarations and constraints that always hold in each model. **Scope** shows the upper bound on the universe of discourse.

In this section, we address the following research questions:
RQ1: How does the size of the starting test suite impact the performance of *SketchGen*^{2}?
RQ2: What is the expression generation efficacy of *SketchGen*^{2}?
RQ3: What is the sketch efficacy of *SketchGen*^{2}?
RQ4: What is the quality of test suites produced by *SketchGen*^{2}?

4.2 RQ1: Test Suite Impact

Table 2 presents *SketchGen*^{2}'s performance when different sized test suites are used to start the technique, focusing on time and the size of the test suites produced, as all result in the same set of expressions. Details about the list of expressions generated by *SketchGen*^{2} can be seen in Table 3. For Table 2, the column **Model** shows the model under consideration. One model, **deadlock**, has two expression holes, each with a different set of domains, resulting in two rows in the table. The headings **Start X** represent the three different configurations: starting *SketchGen*^{2} with 1, 5 and 10 test cases respectively. The test suites all start with the same test cases, i.e. the first five tests for the **Start 10** configuration are the five tests used for the **Start 5** configuration. For each configuration, we present three pieces of information: the number of tests generated by *SketchGen*^{2} (**#Gen**), the total number of tests, which is the size of the starting test suite plus the number of generated tests (**#Total**) and the execution time in milliseconds (**Time**).

Since *SketchGen*^{2} works by first running *Modulo-Instance Pruning*, our expectation is that starting with too small of an initial test suite would result in *SketchGen*^{2} relying more on the expensive dynamic equivalence checks and relying less on the cheaper modulo test checks, inflating the runtime. The results in Table 2 supports this assumption, although the results reveal that the difference in runtime between configurations is overall minor. On average, the **Start 1** configuration takes 1.1× longer than both **Start 5** and **Start 10** configurations. Corresponding, the **Start 5** configuration takes on average 1.03× longer than the **Start 10** configuration. Of note, for 9 of the 11 executions, all three configurations finish within four seconds of each other. The two exceptions are the two models which generate the most expressions: **btree** and **remove**. For **btree**, the three configurations finish within 40 seconds of each other, with the **Start 5** configuration being the fastest. For **remove**, the three configurations finish within 10 seconds of each other, with the **Start 10** configuration being the fastest.

In terms of test generation, the **Start 1** configuration creates the most tests. In the worst case, for **remove**, the **Start 1** configuration generates 5 and 9 more tests than the **Start 5** and **Start 10** configurations, respectively. However, the **Start 1** configuration is not always an increased burden. For **deadlock**, all three configurations generate the same number of tests and for **btree**, both the **Start 1** and **Start 5** configurations produce two less tests than the the **Start 10** configuration. On average, the number of new tests generated for each configuration is 24 (**Start 1**), 21 (**Start 5**) and 19 (**Start 10**). The decrease in the number of tests generated corresponding with a larger starting test suite is expected, as having less tests means that there is likely more undetected equivalences that need to be accounted for. Of note, the average number of new tests created for all configurations is similar to Alloy's coverage-based and mutation-based test generation techniques [40], which also rely

Table 2: Test suite details for *SketchGen*^{2} using different starting test suites.

Model	Start 1			Start 5			Start 10		
	#Gen	#Total	Time	#Gen	#Total	Time	#Gen	#Total	Time
bempl	10	11	1007	6	11	974	2	12	923
btree	54	55	240452	54	59	227267	56	66	266605
contains	31	32	16070	28	33	17283	26	36	18023
deadlock	3	4	198	3	8	340	3	13	317
deadlock2	5	6	2996	3	8	1831	1	11	2063
dll	26	27	5954	25	30	5599	26	36	6620
fsm	21	22	6026	17	22	5620	15	25	5839
grade	12	13	1440	10	15	3156	8	18	1168
graph	10	11	1992	5	10	1678	2	12	1718
remove	79	80	230223	74	79	235759	70	80	226075
sll	12	13	4910	7	12	4156	3	13	2645

on human oracles, and is small enough that it is feasible for a user to label the test cases. For the overall test suite, all configurations result in an average final test suite sizes that are nominally different from one another: 25 (**Start 1**), 26 (**Start 5**) and 29 (**Start 10**).

While the overhead of all three configurations is similar to each other, the results in Table 2 point to a few trends. The **Start 1** configuration generates the highest number of new tests but often has the smallest total test suite. At the same time, the **Start 1** configuration also frequently takes the longest to run. In contrast, the **Start 10** configuration often generates the least number of new tests, but also frequently ends up with the largest total test suite. Meanwhile, the **Start 5** configuration generates expressions the fastest more often than the other two configurations. Given the higher runtime of the **Start 1** configuration and the diminishing returns of starting with a higher amount of tests, we use the **Start 5** configuration in the remainder of our experiments. In terms of easing the adoption of *ASketch*, manually creating 5 diverse test cases requires a small amount of effort from the user but still results in an efficient *SketchGen*^{2} execution. At the same time, it is worth noting that *SketchGen*^{2} performs well even when starting with only a single test case. Therefore, a user can feasible utilize *SketchGen*^{2} to effectively sketch a model by creating only one test case.

4.3 RQ2: Expression Generation Efficacy

The author's motivation is to use *SketchGen*^{2} to generate *RexGen*'s *Dynamic Pruning* list, which is an optimal list to use for sketching since every expression is non-equivalent up to a given scope. To explore the efficacy of *SketchGen*^{2}'s expression generation capabilities, Table 3 shows the performance of the different expression generation techniques and their application to sketching Alloy models. The column **Model** shows the model under evaluation. The **Expression Generation** columns show information related to expression generation: **Strat** conveys the generation strategy, **#Expr** shows the number of expressions generated and **Time** shows the runtime of the expression generation technique for the appropriate problem in milliseconds. In our experiments, we generate expressions up to the minimum cost needed to sketch our oracle solution. For **deadlock**, which has two different expression holes, the numbers are reported in pairs in the table. The remainder of the columns outline the sketch environment. Column **#Holes** shows the number of holes in the sketch, column **Space** shows the size of the search space (number of fragments combinations for all holes), and the columns **#Prim**, **#Cls** and, **Time** show the number of primary variables, clauses, and solving time in milliseconds for the meta-model that solves the sketch, respectively. Lastly, column **Total** depicts the total runtime, including both the expression generation time and the sketching time. Models with (⊥) timed out trying to solve the sketch, meaning the models require more than 30 minutes to sketch. All models that timed out did so when generating the CNF representation for the SAT problem, resulting in no information about the size of the meta-model. The test suite used to sketch the models is generated by *SketchGen*^{2} using the **Start 5** configuration.

To evaluate the tradeoffs between *SketchGen*^{2}'s expression generation strategy and *RexGen*'s *Static Pruning* and *Dynamic Pruning* strategies, the authors focus on the columns under the **Expression Generation** header. The design of *SketchGen*^{2} targets two main expression generation goals: producing the same list as *Dynamic Pruning* while achieving this list more efficiently. The results in Table 3 demonstrate how *SketchGen*^{2} meets both of these goals. In terms of the size of the expressions generated, *SketchGen*^{2} generates on average 2.9× fewer expressions than *Static Pruning* and does generates the same number of expressions as *Dynamic Pruning*. For **remove**, *SketchGen*^{2} see its largest reduction in number of expressions, generating 6.5× fewer expressions than *Static Pruning*.

While *Static Pruning* generates more expressions, because the pruning is only based on known equivalence rules applied during formation of expressions, *Static Pruning* is efficient: all models generate expressions in less than a second. In contrast, *Dynamic Pruning* makes numerous SAT calls to prune expressions and takes longer: *Dynamic Pruning* times out trying to generate expressions for both **btree** and **remove** and takes 2.73 minutes to generate expressions for **contains**. In comparison, while *SketchGen*^{2} does not finish nearly as fast as *Static Pruning*, *SketchGen*^{2} does achieve a speedup over *Dynamic Pruning*, as desired. On average, excluding **btree** and **remove** which timed out, *SketchGen*^{2} is 10.7× faster than *Dynamic Pruning* and unlike *Dynamic Pruning*, *SketchGen*^{2} does complete both **btree** and **remove**. For **contains**, which took *Dynamic Pruning* the longest, *SketchGen*^{2} runs 2.4 minutes faster, which is a 9.5× decrease in runtime. In addition, for **dll**, *SketchGen*^{2}

Table 3: *ASketch* performance with different expression generation techniques. Times are in ms. ⊥ indicates timeout (>30 min).

Model	Expression Generation			#Holes	Space	Sketching			Total
	Strat	#Expr	Time			#Prim	#Cls	Time	
bempl	*Static*	131	49	3	68644	681	3.1e5	5034	5083
	Dynamic	51	5226		10404	521	1.3e5	2016	7242
	SketchGen^2	51	974						2990
btree	*Static*	3473	333	6	2.7e12	-	-	⊥	⊥
	Dynamic	-	⊥		-	-	-	-	-
	SketchGen^2	1637	227267		2.8e11	-	-	⊥	⊥
contains	*Static*	1305	138	3	6.8e6	-	-	⊥	⊥
	Dynamic	238	163891		2.3e5	1426	1.0e6	94113	258004
	SketchGen^2	238	17283						111396
deadlock	*Static*	(17;115)	(25;52)	6	2.8e5	321	2.7e5	10684	10761
	Dynamic	(11;29)	(587;5375)		45936	229	62773	2851	8813
	SketchGen^2	(11;29)	(340;1831)						4682
dll	*Static*	182	66	5	7.2e7	930	1.1e6	85064	85130
	Dynamic	139	86650		3.4e7	810	8.6e5	37473	124123
	SketchGen^2	139	5599						43072
fsm	*Static*	297	79	4	1.1e6	930	1.5e6	254270	254349
	Dynamic	150	51504		2.7e5	636	7.3e5	48752	100256
	SketchGen^2	150	5620						54372
grade	*Static*	182	66	6	1.7e10	1288	8.8e5	84413	84479
	Dynamic	64	5340		2.7e8	816	2.3e5	7002	12342
	SketchGen^2	64	1356						8358
graph	*Static*	169	59	5	1.1e6	433	3.7e5	22889	22948
	Dynamic	102	13759		3.7e5	297	2.0e5	7319	21150
	SketchGen^2	102	965						8897
remove	*Static*	5746	965	4	5.7e11	-	-	⊥	⊥
	Dynamic	-	⊥		2.0e9	-	-	⊥	⊥
	SketchGen^2	876	235759						⊥
sll	*Static*	214	70	5	2.2e6	598	5.6e5	145388	145458
	Dynamic	107	34203		5.5e5	348	2.7e5	9001	43204
	SketchGen^2	107	4156						13157

achieves its largest magnitude speed up over *Dynamic Pruning* of 47.3×, which translates to a speed up of 81.0 seconds.

Overall, *Static Pruning* is runtime efficient; however, it produces notably larger lists compared to the other strategies. While *Dynamic Pruning* has a large overhead, *SketchGen*^{2} successfully generates the same expressions with a significantly shorter runtime.

4.4 RQ3: Sketch Efficacy

While the results in Section 4.3 show that *SketchGen*^{2} is preferable to *Dynamic Pruning*, this section evaluates whether the tradeoff regarding size and time between *SketchGen*^{2} and *Static Pruning* is worthwhile when sketching models. Therefore, to evaluate the efficacy of utilizing the different expression generation strategies to sketch models, the authors focus on the columns under the **Sketching** header in Table 3. Since *Dynamic Pruning* and *SketchGen*^{2} use the same set of expressions, their sketching details are reported together, with the total time (column **Total**) being the difference.

SketchGen^2's reduction in the number of expressions generated has a clear impact in reducing the size of the search space of candidate models. *ASketch*'s search space when using *SketchGen*^{2} is on average 53.5× smaller than when using *Static Pruning*, with **remove** and **grade** seeing the largest reduction at 282.2× and 261.0× respectively. This large magnintude reductions in search space highlight how quickly expression holes can inflate the size of the search space. Consider **remove** which has the largest reduction in search space. The sketch for **remove** includes 3 expression holes. *Static Pruning* creates 5,746 different expressions to fill each of the 3 holes while *SketchGen*^{2} only creates 876 expressions. Even for the model with the smallest reduction in search space, **dll**, reducing the number of expressions from 182 to 139 across the 3 expression holes in **dll**'s sketch still reduces the search space by 2.1×, which, in turn, results in a 2.3× reduction in runtime for *ASketch*.

As seen with **dll**, given the reduction in the size of the search space produced by *SketchGen*^{2}, we expect that using *SketchGen*^{2}'s expression list would also results in improved runtime performance for *ASketch*, which is supported by our results. Excluding models

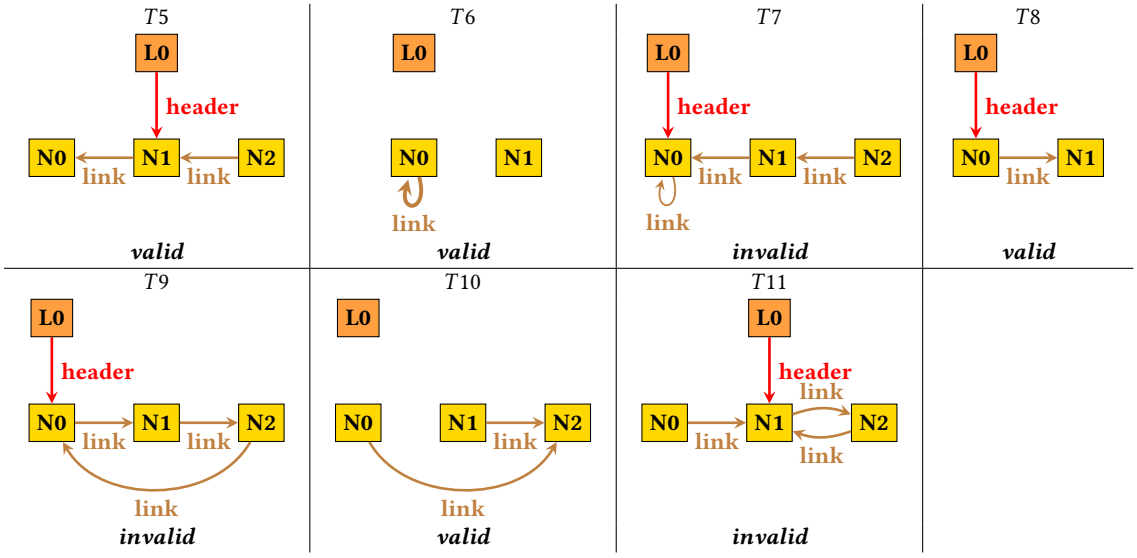


Figure 3: The 7 test cases generated by *SketchGen*^{2} after starting with the five tests in Figure 2.

that time out, *ASketch* finds a solution to the sketch on average 6.5× faster when using *SketchGen*^{2}'s rather than using *Static Pruning*. Models **sll** and **grade** see the largest speed up at 16.2× and 12.1× respectively. Despite reducing the search space, the sketch problems for **btree** and **remove** remain too large to sketch within their timeout bounds of 30 minutes. However, *SketchGen*^{2}'s smaller expression list for **contains** does result in *ASketch* finding a solution in 111 seconds, compared to timing out with *Static Pruning*.

While running *ASketch* with *SketchGen*^{2} instead of *Static Pruning* results in better performance, *SketchGen*^{2} does have a longer expression generation time. As a result, it is important to consider the total overall runtime, depicted in column **Total**. For the models that do not time out, the overall time is 4.7× faster utilizing *SketchGen*^{2} over *Static Pruning*. Again, models **sll** and **grade** see the largest speed up at 11.1× and 10.5× respectively. The minimum speed up by *SketchGen*^{2} is 1.7×, which occurs for the **bempl** model. Therefore, when sketching Alloy models, *SketchGen*^{2}'s smaller list of expressions, which makes the search space more tractable, outweighs its longer generation time in comparison to *Static Pruning*.

4.5 RQ4: Test Suite Quality

To consider the quality of the generated test suites, the authors check whether the solutions found using these test suites were correct or plausible. We use the Analyzer to determine equivalence between the solutions found by *ASketch* and the oracle formulas. For example, below is the check command for the **sll** model:

```
check {
  all n: Node | n in List.header.^link => n !in n.link.^link
  <=> all n: Node | n in List.header.*link => n !in n.^link
} for 3
```

For all the models used in our evaluation, we find that all solutions found when running *ASketch* for the results in Table 3 are equivalent to the respective oracle solutions.

Furthermore, the authors manually inspect the generated test suites to determine if these test cases are valuable for sketching. Prior work [38] has shown that there are two important characteristics for a sketching-oriented test suite: (1) the tests should exercise high formula-level coverage of the final formula, which helps mitigate concerns about plausible solutions, and (2) each test case should be able to eliminate unique candidate models in comparison to the rest of the test suite, which helps improve the runtime. Figure 3 shows all of the test cases generated by the **Start 5** configuration for the singly-linked list model. To achieve high coverage, it is important for the test suite to not only be comprised on both invalid and valid tests, but to also explore all the different ways that valid and invalid behavior can occur. For the tests in Figure 3, we can see that of the tests depicting invalid list, we are presented with lists that are invalid due to: (1) a self loop at the start of the list ($T7$), (2) two nodes pointing directly back to each other ($T11$), and (3) a larger, more indirect cycle ($T9$). For valid behavior, we can see lists that explore (1) correctly connected nodes ($T8$), (2) nodes with cyclic behavior but are disconnected from the list ($T6$), (3) empty lists and disconnected nodes ($T10$) and (4) lists with no cycles and disconnected nodes ($T5$).

The creation of tests that do not contribute any unique information for eliminating candidate models can impact *ASketch*'s performance by both increasing the size of the meta-model as well as adding redundant constraints that bloat the satisfiability problem. Unfortunately, our inspection revealed the test suites created by *SketchGen*^{2} are not guaranteed to be minimal for sketching purposes. To illustrate, consider the following two tests $T1$ (Figure 2) and $T8$ (Figure 3). Conceptually, test $T8$'s valuation is an extension of $T1$'s valuation. In terms of representing valid behavior of the `Acyclic` predicate, both tests represent a scenario in which all the nodes are in the list and there is no cycle. Rather than having both tests, it would be more desirable to just have $T8$. This behavior

occurs again with $T2$ and $T7$, which both depict invalid behavior where a node in the list has a self loop, and with $T3$ and $T11$, which both depict invalid behavior where two nodes link directly back to each other. If we rerun *SketchGen*^{2} with $T8$, $T7$, and $T11$ in place of $T1$, $T2$, and $T3$, then we do not generate $T1$, $T2$, and $T3$.

While the presence of all six tests will not cause an issue in terms of the correctness of the sketch, the presence of these tests highlight that *SketchGen*^{2}'s resulting test suite can be improved. First, many of the extended scenarios are built out of our manually created tests used to initiate *SketchGen*^{2}. Therefore, we note the importance of generating larger individual tests before running *SketchGen*^{2}. In addition, *SketchGen*^{2} can be extended to detect if a test case is an extension of a previous test, and look to see if the smaller test case can be safely removed. Second, *SketchGen*^{2} currently uses the first counterexample produced that explicitly distinguishes between *two* non-equivalent expressions. However, *SketchGen*^{2} does not currently place any other constraints on this counterexample. One avenue of future work is to attempt to generate more valuable counterexamples by asking Alloy to find either a maximal counterexample or to generate a counterexample that explicitly distinguishes between *multiple* non-equivalent expressions, rather than distinguishing between two non-equivalent expressions.

5 THREATS TO VALIDITY

There exists a few threats to the validity for the results. For the *SketchGen*^{2} configurations, the test cases chosen are from previous work sketching Alloy models. Therefore, these test cases may be particularly well suited for *SketchGen*^{2}, as these tests were designed to evaluate *ASketch* in the past. Therefore, different starting test suite may result in a different performance for *SketchGen*^{2}. However, our evaluation does highlight that an end user should target creating a small number of diverse and intricate tests for their starting test suite to help improve *SketchGen*^{2}'s performance and its application to *ASketch*. For expression generation, *RexGen* has several parameters to specify the upper bound on the size of expressions to generate. For the evaluation models, the authors use the minimum size of expressions needed to create the known oracle solution, which ensures that each sketch is solvable. In practice, the minimum size needed to generate the valid expression is not known in advance. Lastly, the models are benchmark models used to evaluate prior sketching work and largely fall into two categories: data structures and protocols. The authors' results may not generalize to other types of system models. However, these models have frequently been used to evaluate new Alloy techniques [25, 26, 40, 46].

6 RELATED WORK

Input Generation for Constraint Languages. *SketchGen*^{2} is at its core an input generation technique for Alloy that automatically creates AUnit test cases that need to be labeled valid or invalid by an oracle. Prior work has addressed automated input generation for Alloy, including coverage-based generation and mutation-based generation [40]. Testing constraint languages outside of Alloy has been addressed in previous work. For example, a test framework was built for the constraint language OPL which focuses on using an oracle model to derive tests that look for differences in behavior based on conformity properties and provides guidance for fault localization [20, 21]. Moreover, previous work introduced a reduction of testing UML models to satisfiability checking by encoding the model and a property of interest, and using SAT [33]. These efforts have largely focuses on creating testing environments and are not applied or created for a synthesis environment.

Expression Generation. When generating expressions for programming languages, there quickly arises a need to prune the search space, as the number of expressions often becomes intractable, preventing the use of these expressions to other applications, such as synthesis. Pruning techniques include determining the indistinguishability of expressions modulo a set of inputs [2, 44] and partial evaluation of incomplete expressions [7]. Additionally, knowledge about operator properties has also been used to explore equivalent expressions, either after expression generation [7] or by applying an automated transformation to the grammar [18]. *SketchGen*^{2} does not add to the types of pruning for Alloy expressions, but instead aims to improve the efficiency of how Alloy expressions can be pruned through a novel combination of *Modulo-Instance Pruning* and *Dynamic Pruning*.

Program Sketching. The aim of *SketchGen*^{2} is to provide automated input generation for sketching. Program sketching [1, 16, 31, 32, 34–37] is a form of program synthesis, which is a mature yet active research topic [3, 6–8, 12, 17, 19, 22, 27, 31]. 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 [5]. *SketchGen*^{2} is designed to work with *ASketch*, which uses unit tests to outline the expected behavior. Previous work on program synthesis has also used user provided tests to synthesize imperative code. SyPet [6] uses tests and Petri nets to synthesize0 sequences of method invocations for complex APIs. Test-Driven Synthesis builds a C# program such that it satisfies all tests [28]. While *SketchGen*^{2} is not a synthesis technique, *SketchGen*^{2} is designed to generate test cases that are valuable for sketching, as each test distinguishes between at least two expressions which are non-equivalent.

7 CONCLUSION

While software models are a valuable resource to create more reliable software systems, models are notoriously difficult to write correctly. *ASketch* introduces a framework for partial synthesis of Alloy models through sketching. Unfortunately, *ASketch* still requires users to write a valuable regular expression and a robust test suite in order to generate an Alloy model that matches the user's expectation. This paper introduces *SketchGen*^{2}, which automates a majority of the input generation needed for *ASketch*. Experimental results reveal that *SketchGen*^{2} is able to efficiently generate a list of expressions for synthesis while strengthening the user's test suite to handle the broad list of expressions produced by *SketchGen*^{2}. In particular, we show that the starting with a small collection of strong tests makes *SketchGen*^{2} efficient and results in a good test suite for use by *ASketch*.

ACKNOWLEDGMENTS

The work is partially supported by the National Science Foundation under Grant No. CCF-2123341.

REFERENCES

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*.

[2] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *TACAS*.

[3] Rastislav Bodík and Barbara Jobstmann. 2013. Algorithmic program synthesis: Introduction. *STTT* (2013).

[4] CheckMate GitHub. 2019. https://github.com/ctrippel/checkmate. (2019).

[5] Yves Deville and Kung-Kiu Lau. 1994. Logic program synthesis. *The Journal of Logic Programming* 19 (1994).

[6] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based Synthesis for Complex APIs. In *POPL*.

[7] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI*.

[8] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *ICSE*.

[9] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *IEEE TSE* 39, 9 (2013).

[10] Juan P. Galeotti, Nicolás Rosner, Carlos G. López Pombo, and Marcelo F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *TSE* (2013).

[11] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-Based Program Repair Using SAT. In *TACAS*.

[12] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. 2011. Interactive Synthesis of Code Snippets. In *CAV* (Snowbird, UT).

[13] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *TSE* (2002).

[14] Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[15] Daniel Jackson and Mandana Vaziri. 2000. Finding Bugs with a Constraint Solver. In *ISSTA*.

[16] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: Sketching for Java. In *FSE*.

[17] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *OOPSLA*.

[18] Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. 2016. An Update on Deductive Synthesis and Repair in the Leon Tool. In *SYNT Workshop*.

[19] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete Functional Synthesis. *SIGPLAN Not.* 45, 6 (2010).

[20] N. Lazaar, A. Gotlieb, and Y. Lebbah. 2010. Fault Localization in Constraint Programs. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, Vol. 1. 61–67. https://doi.org/10.1109/ICTAI.2010.18

[21] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. 2010. *Principles and Practice of Constraint Programming – CP 2010: 16th International Conference, CP 2010, St. Andrews, Scotland, September 6-10, 2010. Proceedings*. Chapter On Testing Constraint Programs.

[22] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. *PLDI* (2005).

[23] Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *ASE*.

[24] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-purpose Higher-order Relational Constraint Solver. In *Proc. 37th International Conference on Software Engineering - Volume 1*.

[25] Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. 2017. The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance. In *FSE*.

[26] Tim Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2013. Aluminum: Principled Scenario Exploration Through Minimality. In *ICSE*.

[27] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. *SIGPLAN Not.* 50, 6 (2015).

[28] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven Synthesis. *PLDI* (2014).

[29] Sorawee Porncharoenwase, Tim Nelson, and Shriram Krishnamurthi. 2018. CompoSAT: Specification-Guided Coverage for Model Finding. In *FM*.

[30] Hesam Samimi, Ei Darli Aung, and Todd D. Millstein. 2010. Falling Back on Executable Specifications. In *ECOOP*.

[31] Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *CAV*.

[32] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *FSE*.

[33] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. 2010. Verifying UML/OCL models using Boolean satisfiability. In *Design, Automation Test in Europe Conference Exhibition*.

[34] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. University of California, Berkeley.

[35] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching Stencils. *SIGPLAN Not.* 42, 6 (June 2007), 167–178.

[36] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching Concurrent Data Structures. In *PLDI*.

[37] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *ASPLOS*.

[38] Allison Sullivan. 2017. *Automated Testing and Sketching of Alloy Models*. Ph. D. Dissertation. University of Texas at Austin.

[39] Allison Sullivan, Kaiyuan Wang, Sarfraz Khurshid, and Darko Marinov. 2017. Evaluating State Modeling Techniques in Alloy. In *SQAMIA*.

[40] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *ICST*.

[41] Alloy Team. [n. d.]. http://alloy.mit.edu/alloy/documentation/alloy4-grammar.txt.

[42] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In *MICRO*.

[43] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2019. Security Verification via Automatic Hardware-Aware Exploit Synthesis: The CheckMate Approach. *IEEE Micro* (2019).

[44] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying protocols with concolic snippets. In *PLDI*.

[45] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. ASketch: A Sketching Framework for Alloy. In *Proceedings of the 2018 26th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE Demo)*. 916–919. https://doi.org/10.1145/3236024.3264594

[46] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2020. Fault Localization for Declarative Models in Alloy. In *ISSRE*.

[47] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. 2018. Systematic Generation of Non-equivalent Expressions for Relational Algebra. In *International ABZ Conference ASM, Alloy, B, TLA, VDM, Z*.

[48] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. Solver-Based Sketching of Alloy Models Using Test Valuations. In *International ABZ Conference ASM, Alloy, B, TLA, VDM, Z*.

[49] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. 2010. Contract-Based Data Structure Repair Using Alloy. In *ECOOP*.

[50] Pamela Zave. 2012. Using Lightweight Modeling to Understand Chord. *SIGCOMM Comput. Commun. Rev.* 42, 2 (2012), 49–57.