REACH: Refining Alloy Scenarios by Size (Tools and Artifact Track)

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

Allison Sullivan
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. A key strength of Alloy is its scenario-finding toolset, the Analyzer, which allows users to explore all valid scenarios that adhere to the model's constraints up to a user-provided scope. In Alloy, the Analyzer presents scenarios in the order they are discovered by a backend SAT solver, which is effectively unordered. This paper presents *Reach*, an enhancement to the Analyzer which allows users to explore scenarios by size. Experimental results reveal *Reach*'s enumeration improves performance while having the added benefit of maintaining a semi-sorted ordering of scenarios for the user. Moreover, we highlight *Reach*'s ability to improve the performance of Alloy's analysis when the user makes incremental changes to the scope.**

***Index Terms*—Alloy, SAT Solver, Scenario Enumeration**

I. Introduction

Our lives are increasingly dependent on software systems. However, these same systems, even the most safety-critical ones, are notoriously buggy. Therefore, there is a growing need to produce reliable software while keeping the cost low. One solution is to make use of declarative modeling languages to help improve software correctness. Alloy [9] is a first-order, relational modeling language. A key strength of Alloy is the ability to develop models in the Analyzer, a scenario enumeration toolset that lets users explore behavior enabled by their models. To achieve this, the Analyzer invokes off-the-shelf Boolean satisfiability (SAT) solvers to search for scenarios, which are assignments to the sets of the model such that all executed formulas hold. As output, the Analyzer produces a collection of scenarios the user can explore. Alloy scenarios have been used to validate software designs [13], [17], to test and debug code [5], [14], to repair program states [21], [29] and to synthesize security attacks [27]. However, effectively applying scenarios to improve the correctness of a software system, and thus improve the system's reliability, depends heavily on how easily users can discover scenarios of interest.

The Analyzer's enumeration process will explore all scenarios up to the user provided scope. Unfortunately, this results in the Analyzer finding hundreds of scenarios and there is nothing in Alloy's encoding that guarantees that the next scenario enumerated will be in any way related to the previous scenario. Instead, the order in which scenarios are discovered is based solely on the order that the back-end SAT solver discovers them, which appears as random. For instance, it is possible for the enumeration order to present a scenario of size 1 followed by a scenario of size 3 followed by a scenario of size 1 again. This places a high burden on the user to iterate over scenarios and find ones of value. Furthermore, in order to improve software reliability, the underlying model itself needs to be accurate. However, writing models correctly is a difficult and error-prone task. When a user is validating that their modeled constraints are correct, it is common for the user to desire to check the behavior of their model over smaller scenarios first, then move onto larger scenarios.

To address these limitations, this paper introduces *Reach*, an extension to the Analyzer which provides support for staged generation of scenarios by size. In this paper, we refer to the size of a scenario as the size of the largest signature set in the scenario. This view of size is based on how Alloy scopes, which outlines an upper bound on the size of all signatures, work. Rather than producing one enumeration across the entire scope, *Reach*'s execution results in a separate enumeration per size allowed by the scope. Therefore, *Reach* enables the user to intentionally explore scenarios of a specific size and to control when they move onto exploring scenarios of a different size. To achieve this, *Reach* (1) restricts the upper bound of the conjunctive normal form (CNF) encoding to match the size being enumerated and (2) modifies the CNF encoding to enforce that a scenario meets the required size because of a specific signature in the model, which results in a semi-sorted ordering of scenarios for the user.

Our experimental results show that *Reach*'s enumeration has no noticeable overhead and often improves the overall runtime. Moreover, in comparison to the Analyzer, *Reach* improves the feasibility of using Alloy models to increase software reliability in two ways. First, *Reach*'s staged generation by size provides users with an execution environment that directly enables them to efficiently validate their model. As a result, users can produce more accurate software models, which in turn, enables the user to verify the right software system. Second, the predictable order of *Reach*'s enumeration allows users to more easily navigate through the scenarios to discover key scenarios that can be used to validate the software system's design or implementation.

(a)

```
1. sig List {header: lone Node}
2. sig Node {link: lone Node}
3. pred acyclic {
4.   all l : List {
5.     all n : l.header.*link | n !in n.^link
6. }}
7. run {acyclic} for 3
```

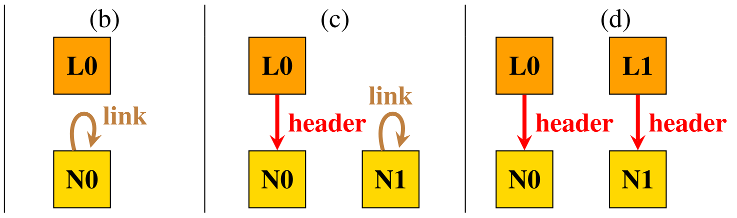


Fig. 1. Alloy Model and Scenarios of a Singly-Linked List

In this paper, we make the following contributions:

CNF Encoding for Size: We present a novel CNF encoding for scenario enumeration that both enforces a specific size of a scenario and outlines how this size is achieved.

Staged Generation: We modify the Analyzer's execution environment to stage generation of scenarios by size, including updating the Analyzer to support multiple concurrent enumerations, one for each size in scope.

Evaluation: We compare *Reach*'s performance to the default Analyzer enumeration and two baseline techniques using a variety of subject models used to evaluate recent advancements to Alloy. In addition, we explore how *Reach* can improve the performance of Alloy in an iterative deepening setting.

Open Source: We release our enumeration framework as an open-source extension to version 5.0.1 of the Analyzer. *Reach* can be found at: https://REACHExtension.github.io.

II. Background

A. Alloy

To highlight how modeling in Alloy works, Figure 1 (a) depicts a model of a singly-linked list with an acyclic constraint. Signature paragraphs introduce atoms and their relationships (lines 1 - 2). Line 1 introduces a named set `List` and uses the relation `header` to express that each `List` atom has zero or one header nodes (`lone`). Similarly, line 2 introduces the named set `Node` and uses the `link` relation to express that each `Node` atom points to zero or one other nodes. Predicate paragraphs introduce named first order logic formulas that can be invoked elsewhere (lines 3 - 6). The predicate `Acyclic` uses universal quantification (`all`), set exclusion (`!in`), relation join (`.`), transitive closure (`^`) and reflexive transitive closure (`*`) to express the idea that "for every list, all nodes in the list are not reachable from themselves. " Commands indicate which formulas to invoke and what scope to explore. The scope places an upper bound on the size of all signature sets. The command on line 7 asks the `Analyzer` to search for satisfying assignments to all the sets of the model (`List`, `header`, `Node`, and `link`) such that `Acyclic` is true using up to 3 `List` atoms and 3 `Node` atoms.

Figure 1 (b), (c) and (d) displays the first three scenarios found by the Analyzer: an empty list with a disconnected node that has a cycle, a list with one node and a disconnected node that has a cycle, and two lists each with one node and no cycles. In Alloy, scenarios are produced by KodKod [26], which generates a CNF formula equivalent to the invoked constraints and uses a back-end SAT solver to find a satisfying solution. KodKod then translates this solution back into an Alloy scenario. Specifically, KodKod creates a list of all possible assignments that can populate a set and ties those to unique integer values, which make up the primary variables of the CNF formula. While solving the CNF formula, the SAT solver determines if each primary variable should be positive (true) or negative (false). A positive assignment means the atom is in the scenario.

To illustrate, for the command in Figure 1 (a), KodKod produces 24 primary variables for the CNF encoding: variables 1-3 relate to set `List`, 4-6 relate to set `Node`, 7-15 relate to set `header` and 16-24 relate to set `link`. For set `List`, the primary variable "1" reasons over whether or not atom `L0` is in `List`, primary variable "2" reasons over whether or not atom `L1` is in `List`, and primary variable "3" reasons over whether or not atom `L2` is in `List`. To create the scenario in Figure 1 (b), the SAT solver found a satisfying assignment for all 24 primary variables in which the following three primary variables were positive: "1" (`L0` in `List`), "4" (`N0` in `Node`), and "16" (`N0->N0` in `link`). The remaining primary variables were all assigned negative values. To enumerate a new scenario, KodKod instructs the SAT solver to find another satisfying assignment to the primary variables with an additional CNF clause that asserts that any new assignment found must differ from all previously discovered assignments by at least one primary variable.

B. *Reach Enumeration*

The Analyzer's default enumeration process prevents repeatedly discovering duplicate scenarios; however, it does not guarantee any sort of relationship between back-to-back scenarios. *Reach* serves to give the user a high level structure for which scenarios will be presented to them. Specifically, *Reach* modifies Alloy's enumeration process to create a separate enumeration for each size up to and including the user provided scope. To illustrate, for the model in Figure 1 (a), *Reach* creates four distinct enumerations which produce 1 scenarios of size 0, 6 scenarios of size 1, 38 scenarios of size 2 and 299 scenarios of size 3, respectively. Figure 2 shows the first 8 scenarios enumerated by *Reach* if the user decides to enumerate scenarios from the smallest size to the largest size. For scenarios of size 0, the user is presented with just Figure 2 (a). Then, for scenarios of size 1, the user will be presented with the scenarios shown in Figure 2 (b) - (g). Next, for scenarios of size 2, *Reach* will present the scenario in Figure 2 (h). From there, the user can continue to enumerate

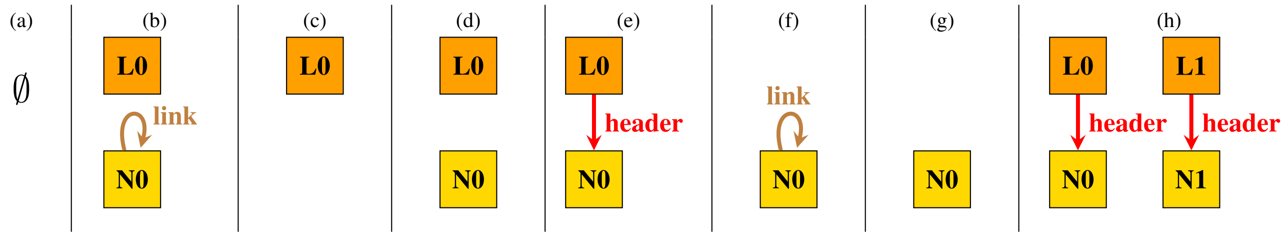


Fig. 2. Reach Enumeration

the remaining 37 scenarios of size 2. Importantly, users can switch between enumerations of different sizes at any point.

In addition to focusing on enumeration by size, *Reach*'s scenarios are generated in a semi-sorted order based on which signature determines the size of the scenario. To illustrate, consider the order of the size 1 scenarios in Figure 2. *Reach* first enumerates all scenarios that are of size 1 due to the `List` signature, (Figure 2 (b) - (e)). Then, *Reach* enumerates all of the scenarios that are of size 1 due to the `Node` signature (Figure 2 (f) - (g)). Of note, *Reach* still appends the CNF clause that prevents discovering duplicate scenarios from the default enumeration strategy. Accordingly, when exploring scenarios related to the signature `Node`, *Reach* does not rediscover the scenarios in Figure 2 (b), (d), and (e) even though these scenarios also have a `Node` set of size 1.

It is possible to enumerate scenarios by size using Alloy's first order logic. For instance, to enumerate scenarios of size 1, one could write the following sequence of commands:

```
run {Acyclic} for 1 but exactly 1 List
run {Acyclic and #List < 1} for 1 but exactly 1 Node
```

The first command uses `exactly` to ensure all scenarios are of size 1 because of the `List` signature. The second command additionally uses conjunction (`and`) and set cardinality (`#`) to ensure all scenarios are of size 1 because of the `Node` signature but avoids duplicate scenarios by preventing the `List` signature from being size 1. While creating this sequence of commands is trivial for our example, creating these commands quickly become a burden on the user as the number of signatures increases. A recent profiling of 1,652 Alloy models found that the median number of signatures in a model is 8 and that 75% of models range from 2 to 25 signatures [7]. In addition, prior work has shown that setting the size of signatures using the cardinality operator results in slower solving times, thus it is discouraged [9], and our experimentation revealed limitations of the `exactly` keyword.

C. Challenge: Augmenting Enumeration Without Loss of Information

The Analyzer's robust enumeration strategy often leads to hundreds or even thousands of scenarios. Therefore, there have been several efforts to augment the Analyzer's enumeration process [16], [25], [24], [23], [19], [18]. Overall, theses techniques aim to present a smaller collection of highly valuable scenarios to the user by applying additional criteria that a scenario must meet during generation. Unfortunately, a user study of different enumeration strategies for Alloy revealed that users are more likely to switch back to the default Analyzer [4]. One reason is that these techniques often produce significantly less scenarios for the user. To illustrate, for the singly-linked list model, there are 344 unique scenarios to discover. For this same model, using Aluminum [16], which enumerates minimal scenarios, produces just the empty scenario. As another example, using AUnit [25], which enumerates scenarios with different coverage, creates just 12 scenarios.

The decrease in scenarios created by these techniques can be counter to one of the most common uses of scenario finders: helping a user understand their system by exploring the concrete examples of behavior enabled by their system. While Alloy's enumeration can feel random, there is value to this: a user can be presented early with an unexpected scenario. Therefore, although ensuring that interesting scenarios are displayed early is good, if this is coupled with a tradeoff in the breadth of behavior presented, then the user may find these alternative enumeration strategies hinder their goals.

Reach aims to walk the line between adding structure to the enumeration process while not limiting the information given to the user. For the singly-linked list model, *Reach* will produce all 344 scenarios. Unlike the default enumeration, *Reach* will present the scenarios in a structured order based on size. However, since our only restriction is only on which signature is determining the size of the scenario, *Reach* is able to maintain randomness within each signature exploration for a given size. To illustrate, the first scenario of size 1 that *Reach* enumerates is a list with a disconnected cyclic node (Figure 1 (b)). As a result, even with *Reach*'s ordering, the user is still prompted early to determine if it is their intention to allow nodes to be disconnected from lists. We believe this strikes a balance between giving the user an expected high-level order while also still being able to surprise the user with interesting and potentially unexpected behavior early.

III. Technique

In this section, we first describe *Reach*'s CNF encoding and then we explain how we apply this encoding to stage generation of scenarios by size.

A. CNF Encoding

During enumeration, *Reach* creates new CNF clauses that enforce the size of a *specific* signature. To achieve this, *Reach* appends a sequence of CNF clauses that asserts *each* primary variable mapped to the targeted signature is positive, meaning that the atom represented by the primary variable must appear in the signature for any scenario enumerated. There are two main benefits of this. First, our CNF encoding is simplified, as we don't need to build clauses to express the broader idea: "size of sig a is equal to Z or size of sig b is equal to Z or .... size of sig x is equal to Z." This concept is most directly captured as a disjunctive normal form formula, which is complicated to translate into an equivalent CNF formula. Second, the single signature focus of our encoding allows us to present scenarios in a semi-ordered format, which can help users more confidently navigate the scenario space.

Of note, our encoding does not place any restriction on the size of other signatures, giving the SAT solver the freedom to explore all possible behavior of the other sets in the model. For example, when enumerating scenarios of size 1 for the singly-linked list model, if the focus is on the `List` signature, the `Node` set is allowed to take any size, ranging from zero to one, inclusive. To keep our CNF encoding simple, *Reach* creates a distinct enumeration for each size within scope; therefore, *Reach* also modifies KodKod's CNF generation to place an upper bound on all signatures that matches the target size. KodKod's translation will take care of ensuring that the size of all other signatures falls between zero and the targeted size.

As an example, when enumerating scenarios of size 1 for the command in Figure 1 (a), *Reach* will first append the following CNF clause: $\{1\}$. This clause ensure that atom `L0` will appear in the set `List`. Since *Reach* tells KodKod to generate its CNF translation of the model with an upper bound of 1, KodKod will generate only 4 primary variables: "1" reasons over if `L0` is in `List`, "2" reasons over if `N0` is in `Node`, "3" reasons over if `N0` is in `L0`'s `header` relation and "4" reasons over if `N0` is in `N0`'s `link` relation. In contrast, using the Analyzer's default translation, KodKod would produce the 24 primary variables outlined in Section II, which includes 3 primary variables per signature. As a consequence, in order to enumerate scenarios of size 1, *Reach* would need to append additional CNF clauses to prevent the size of all signatures from being too large. For our example, *Reach* would need to append the following clauses: $\{1\}$, $\{-2\}$, $\{-3\}$, $\{-5\}$, and $\{-6\}$. Together, the first three clauses enforce the exact size of the set `List`: $\{1\}$ ensures that `L0` appears in `List` and both $\{-2\}$ and $\{-3\}$ ensure that `L1` and `L2` do not appear in `List`. The last two clauses, $\{-5\}$ amd $\{-6\}$, ensures that `N1` and `N2` do not appear in `Node`. Rather than bloat the encoding this way, *Reach*'s modification to the KodKod translation streamlines *Reach*'s encoding to just one clauses ($\{1\}$).

B. Staged Generation

To stage generation by size and to enforce an ordering on signatures, *Reach* modifies the Analyzer's enumeration algorithm in two ways. First, as mentioned, *Reach* creates a separate enumeration *per* size allowed by the scope. Therefore, *Reach* invokes KodKod multiple times, once for each size, to translate the model into a CNF formula. Second, when searching for scenarios, *Reach* adds the CNF clauses from Section III-A that explicitly enforce the size of a given signature. This results in an iterative process where *Reach*

Algorithm 1: Staged Generation of Scenarios

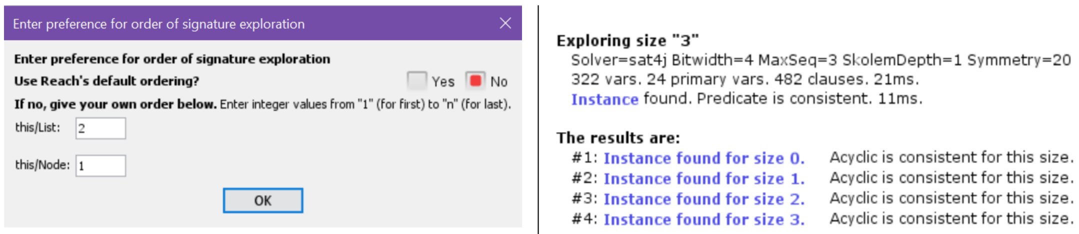
```
Input: Kodkod solver solver, Previous scenarios prev, Set of
       signatures sigs,
Output: Set of scenarios of the target size.
1  activeSig = 0, addClauses = true // Initialize
2  while activeSig != sigs.size() do
3      if addClauses then
4          // Enact Reach's CNF encoding over relevant
             primary variables
5          for $i \leftarrow 1$ to sigs.get(activeSig).getPVars().size() do
6              solver.addClause(sigs.get(activeSig).getPVars().get(i))
7          addClauses = false
8      solution = solver.solve()    // Find next scenario
9      if solution != null then
10         prev.add(solution.getPreventDuplicateClause())
11         cnf.addClause(solution.getPreventDuplicateClause())
12         output(solution)
13     if solution == null then
14         activeSig++
15         solver.cnfClear()    // Clear to remove previous
             active sig clauses
16         for $i \leftarrow 1$ to prev.size() do
17             solver.addClause(prev.get(i))    // Add clauses to
                 prevent dups
18         addClauses = true
```

may need to update the CNF clauses when transitioning from focusing on one signature to the next. Our algorithm takes as input a list signatures (`sigs`), allowing the user to provide any order they may desire for exploration.

Our modified enumeration approach is outlined in Algorithm 1. The variable `activeSig` is used to keep track of which signature is the current focus of the enumeration. The boolean flag `addClauses` is used to determine if *Reach* should append new CNF clauses that assert the size of the signature indicated by `activeSig`. The flag `addClauses` is set every time the active signature changes. The enumeration algorithm runs until all signatures in the model have been fully explored, which is controlled by the whole loop starting on line 2. When attempting to enumerate a new scenario, if `addClauses` is true, CNF clauses are added to the SAT solver following the encoding outlined in Section III-A (lines 4-7).

Then, *Reach* searches for the next scenario by invoking the SAT solver on line 8. If the result is satisfiable, *Reach* generates the CNF clause to prevent rediscovering this scenario and outputs the scenario (lines 9 - 12). Once the SAT solver returns an unsatisfiable result (line 13), *Reach* updates `activeSig` to move onto the next signature. When this happens, *Reach* removes all CNF clauses, so that *Reach* is not still enforcing the exact size of the previous signature. In addition, *Reach* adds back the CNF clauses which prevent duplicate scenarios from being found (lines 16 - 17). Then, *Reach* sets the flag `addClauses` to `true`, which will trigger *Reach* to add size-based clauses at the start of the next enumeration for the new signature indicated by `activeSig`.

To illustrate, to enumerate scenarios of size 1 for the model in Figure 1 (a), *Reach* will translate the command into


Enter preference for order of signature exploration

Enter preference for order of signature exploration

Use Reach's default ordering? Yes [✓ No]

If no, give your own order below. Enter integer values from "1" (for first) to "n" (for last).

this/List: 2

this/Node: 1

OK

(a)

```
Exploring size "3"
   Solver=sat4j Bitwidth=4 MaxSeq=3 SkolemDepth=1 Symmetry=20
   322 vars. 24 primary vars. 482 clauses. 21ms.
   Instance found. Predicate is consistent. 11ms.

The results are:
   #1: Instance found for size 0.    Acyclic is consistent for this size.
   #2: Instance found for size 1.    Acyclic is consistent for this size.
   #3: Instance found for size 2.    Acyclic is consistent for this size.
   #4: Instance found for size 3.    Acyclic is consistent for this size.
```

(b)

Fig. 3. *Reach* Interfaces

KodKod with an upper bound of 1, which sets the primary variables returned by the `getPVars` call. The first active signature is `List`; therefore, *Reach* will append the following clause to assert the size of `List`: $\{1\}$. This will result in the scenarios (b) - (e) in Figure 2. Once the SAT solver is no longer able to find any scenarios, *Reach* removes the clause $\{1\}$ and append the clause $\{2\}$, which assert that `N0` must be in `Node`, resulting in the scenarios in Figure 2 (f) and (g). Once the SAT solver returns an unsatisfiable result, since all signatures have been explored, the enumeration ends and the user is informed that there are no more scenarios of size 1. Then, should the user wishes to explore scenarios of size 2, *Reach* would initiate a new enumeration. This enumeration would be reasoning over the 12 primary variables generated by KodKod's translation. *Reach* would start by appending the following clauses, $\{1\}$ and $\{2\}$, which would ensure that atoms `L0` and `L1` are in the set `List` (e.g. Figure 2 (h)).

IV. Implementation

This section outlines important implementation details of *Reach*, which is an extension to the Analyzer v.5.0.1 [1].

A. Updating Alloy's Workflow

Reach augments the enumeration workflow in the Analyzer in two main ways. First, *Reach* modifies the KodKod translation and enumeration strategy, as outlined in Section III. The bulk of implementing *Reach* involves updating the main command execution driver in Alloy to propagate the size under consideration and the order of signatures (`SimpleReporter`), the translation between Alloy and KodKod to continue to propagate the order of signatures and restrict the size of the KodKod translation (`TranslateAlloyToKodKod`), and the KodKod class that is responsible for incrementally invoking the SAT solver to both gather the relevant primary variables and to enforce Algorithm 1 when searching for the next scenario (`SolutionIterator`). By default, *Reach* will explore signatures in the order they are declared in the model, as developers tend to write the higher-level signatures first then components (i.e. creating `List` before `Node`). However, *Reach*'s backend algorithm can accommodate any order over the signatures. To support this, when the user executes a command, a pop-up menu, shown in Figure 3 (a), gives the user the option to provide their own ordering.

Second, *Reach* supports multiple simultaneous enumerations, which can be thought of as enumerating multiple commands. Previously, the Analyzer only allowed users to enumerate the last solved command. *Reach* removes the enumerator's singleton pattern, in order to allow there to be more than one active enumeration at a time and updates the visualization interface (`VizGUI`) to accept an enumerator object rather than use a global enumerator object. Then, to enable users to view multiple enumerations, *Reach* also updates the Analyzer's log panel to display information per size, as shown in Figure 3 (b). Each "Instance found" link opens its own visualization interface that can be simultaneously enumerated, As a result, *Reach* also improves the functionality of the original Analyzer, which has an "execute all commands" functionality. However, despite the name of the action, the base Analyzer only allows the user to enumeration that last solved command. Our support for multiple simultaneous enumerations enables the user to be able to enumerate any of the solved commands.

B. Supporting Alloy's Diverse Signature Grammar

Alloy allows for signatures to be abstract, extensions of other signatures and to have their own multiplicity constraints. Unfortunately, some of these constructs do not result in primary variables, which impacts our CNF encoding. In particular, if a signature has a singleton multiplicity constraint ("`one`"), no primary variables are generated for this signature, as there is nothing for the SAT solver to solve about which atoms should populate the signature's set. Unfortunately, this behavior results in scenarios found for a scope that may defy the user's intention. To illustrate, if you have a signature with a singleton constraint in your model and you execute a command with a scope of 0, the Analyzer will still find scenarios even though the signature with the singleton constraint will have a size of 1. To resolve this, *Reach* desugars the singleton multiplicity keyword into a fact in the model, which then triggers the generation of primary variables. As a result, *Reach* discovers the previously mentioned scenarios as scenarios of size of 1 rather than scenarios of size 0, which we believe better matches the user's expectation.

Additionally, for an abstract signature, any of the following may be true: (1) it has primary variables, (2) the primary variables are held by just the extension signatures, or (3) there may be no primary variables at all. The first case occurs when there are no extension signatures, in which case *Reach* follows its normal encoding. In the second case, unfortunately, an abstract signature can be extended by more than one signature. Accordingly, all atoms of any signature that extends the abstract signature count towards the size of the abstract signature, which does not work with *Reach*'s streamlined CNF encoding because we have to account for all the different combinations of atoms from all extension signatures. Therefore, *Reach* instead handles this situation at the Alloy level. When enumerating scenarios that are of the target size for an abstract signature, *Reach* appends a fact that asserts that the size of the abstract signature matches the target size. In the last case, this behavior occurs when an abstract signature is only extended by signatures which use the singleton multiplicity constraints. This latter case gets handled by the desugarring we do for the singleton multiplicity constraint.

V. Evaluation

We evaluate *Reach* over a benchmark of Alloy models collected from recent advancements made to Alloy [15], [25]. We address the following research questions:

- **RQ1:** Is *Reach*'s encoding efficient compared to using existing Alloy grammar?
- **RQ2:** What is the overhead of signature-focused enumeration by size?
- **RQ3:** How effective is *Reach* at improving the automated analysis for iterative deepening?

To evaluate *Reach*, we compare four strategies: (1) the Analyzer, (2) two baseline implementations and (3) *Reach*. To collect performance results, we automatically enumerate all solutions and add in a monitor to collect performance metrics, we use the default signature ordering, and we use the published scope. All experiments are performed on Linux Ubuntu 20.04 LTS with 1.8GHz Intel i7 CPU and 16 GB RAM.

We use a broad set of 12 models, including models of data structures (**arr**, **btree**, **dll**, **sll**), models of class diagrams (**cd**), models of puzzles (**nqueens**), and real world models that were generated automatically from software artifacts (**CD2A1**, **CD2A2**, **diff1**, **diff2**) and of surgical robots (**frank**, **tombot**). Table I gives an overview of the size of each model. **#Sig** is the number of signatures. As further context, columns **#Abs**, **#Ext** and **#One** show the number of signatures which are abstract, extensions of another signature, and/or use the singleton multiplicity constraint, respectively. Lastly, column **#Rel** shows the number of relations across all signatures.

A. *Baseline Technique*

Our two baseline techniques use Alloy's first order logic and existing scope grammar to enforce size and the semi-sorted ordering. First, **BaseE** uses the sequence of commands pattern outlined in Section II-B, which makes use of the "`exactly`" keyword from Alloy's scope grammar. We view this as the current native support within the Analyzer to enumerate scenarios of a specific size because of a signature. To illustrate, below is the sequence of commands for **BaseE** for our singly-linked list model:

```
run {Acyclic} for 2 but exactly 2 List
run {Acyclic and #List < 2} for 2 but exactly 2 Node
```

Unfortunately, when exploring our real world models, we discovered two shortcomings of the "`exactly`" keyword when

TABLE I
Size of evaluation models.

Model	#Sig	#Abs	#Ext	#One	#Rel
arr	2	0	0	1	2
btree	2	0	0	1	4
cd	2	0	1	1	1
CD2A1	16	4	12	8	1
CD2A2	10	4	6	2	1
diff1	17	4	13	8	1
diff2	17	4	13	8	1
dll	2	0	0	1	4
frank	52	17	35	28	16
nqueens	1	0	0	0	2
sll	2	0	0	0	2
tombot	52	17	35	28	16

applied to abstract signatures. First, if every extension signature is declared using the singleton multiplicity constraint, the "`exactly`" keyword will fail to detect the true size of the abstract signature set and will incorrectly produce scenarios that are too large. Consider the following:

```
abstract sig A {}
one sig B,C extends A {}
run {} for 1 but exactly 1 A
```

This run command should be unsatisfiable because signature `A` must have a size of two; however, the Analyzer will unfortunately find solutions for this command. Second, the "`exactly`" keyword applied to abstract signatures breaks Alloy's existing symmetry breaking optimizations. Therefore, **BaseE** will find substantially more scenarios that contain redundant structures. To illustrate the magnitude of this problem, for the **CD2A1** model and a size of 6, **BaseE** will produce 56,244 scenarios instead of just 268 scenarios. The loss of symmetry breaking also impacts the overall runtime, as exponentially more scenarios means an exponential increase in search times. **BaseE** took 26.2 minutes to find scenarios compared to *Reach*'s 5 seconds. All of the real world models in our experiments make heavy use of abstract signatures. As a result, **BaseE** did not feel like a proper baseline that really reflected how to enumerate by size within Alloy's existing infrastructure. Therefore, to still have a valid comparison, we developed a second baseline, **BaseC**, that uses the set cardinality operator. To illustrate, below is the sequence of commands for **BaseC** for our singly-linked list model:

```
run {Acyclic and #List = 2 } for 2
run {Acyclic and #List < 2 and #Node = 2 } for 2
```

B. *RQ1: CNF Encoding Efficiency*

Table II shows *Reach*'s and the baselines' performances broken down by each size within scope. Column **Model** reflects the model under evaluation, column **Sz** is the size and column **#Scr** is the number of scenarios. For each technique, we display 4 pieces of performance information. To show the size of the problem, columns **#PV** and **#Cls** show the number of primary variables and the number of clauses. To show the overhead, columns $\mathbf{T}_{avg}$ and $\mathbf{T}_{tot}$ show the average time to discover a scenario and the total time to find all scenarios in milliseconds. There are two summary sections (**R-BaseE** and **R-BaseC**) that compare *Reach* to **BaseC** and *Reach* to

TABLE II
COMPARISON OF *Reach* AND THE BASELINE BROKEN DOWN BY EACH SIZE IN SCOPE. ALL TIMES ARE IN MILLISECONDS.

Model	Scp	#Scr	Reach				BaseC				R-BaseC		BaseE				R-BaseE	
			#PV	#Cls	T_{avg}	T_{tot}	#PV	#Cls	T_{avg}	T_{tot}	$Diff_T$	$Diff_C$	#PV	#Cls	T_{avg}	T_{tot}	$Diff_T$	$Diff_C$
arr	1	255	33	664	6	1599	33	664	6	1524	75	0	32	664	5	1266	333	0
arr	2	1389	50	932	5	7058	50	938	6	8633	**-1575**	**-6**	48	786	4	5524	1534	146
arr	3	2127	67	1213	4	9004	67	1233	5	10298	**-1294**	**-20**	64	992	5	10373	**-1369**	221
btree	1	17	20	124	7	121	20	123	1	15	106	1	19	123	9	145	**-24**	1
btree	2	240	44	1605	5	1089	44	1612	3	826	263	**-7**	42	1455	8	1910	**-821**	150
btree	3	560	72	3321	4	2251	72	3342	3	1945	306	**-21**	69	2942	5	2785	**-534**	379
cd	1	1	1	1	7	7	1	1	1	1	6	0	1	1	1	1	6	0
cd	2	1	5	37	7	7	5	39	1	1	6	**-2**	3	23	1	1	6	14
cd	3	2	11	227	6	12	11	239	2	3	9	**-12**	8	185	11	22	**-10**	42
cd	4	5	19	541	5	25	19	536	3	15	10	5	15	433	13	66	**-41**	108
cd	5	14	29	1506	4	61	29	1510	4	56	5	**-4**	24	1360	15	208	**-147**	146
cd	6	51	41	2514	5	267	41	2511	6	323	**-56**	3	35	2298	16	797	**-530**	216
cd	7	190	55	3755	5	1010	55	3859	7	1398	**-388**	**-104**	48	3573	10	1955	**-945**	182
dll	1	17	20	1053	4	71	20	1053	2	31	40	0	19	1053	13	225	**-154**	0
dll	2	120	44	2603	4	455	44	2611	8	1002	**-547**	-8	42	2475	9	1075	**-620**	128
dll	3	560	72	4533	5	2769	72	4555	5	2908	**-139**	**-22**	69	4265	7	3797	**-1028**	268
nqueens	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
nqueens	1	1	33	1097	23	23	33	1097	6	6	17	0	32	1037	53	53	**-30**	60
nqueens	4	2	132	8036	95	190	132	8036	37	74	116	0	128	7627	74	147	43	409
nqueens	5	10	165	11362	52	517	165	11362	44	435	82	0	160	10707	71	707	**-190**	655
sll	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
sll	1	6	4	14	3	17	4	15	1	8	9	**-1**	3	9	8	45	**-28**	5
sll	2	38	12	156	2	78	12	174	8	291	**-213**	**-18**	10	134	7	259	**-181**	22
sll	3	299	24	482	2	607	24	527	2	654	**-47**	**-45**	21	446	4	1280	**-673**	36
CD2A1	5	180	350	12443	23	4093	350	12443	33	5921	**-1828**	0						
CD2A1	6	268	480	18016	20	5318	480	17644	33	8915	**-3597**	372						
CD2A1	7	568	630	24357	23	12866	630	23893	30	17255	**-4389**	464						
CD2A1	8	1171	800	32084	29	33834	800	31543	37	43024	**-9190**	541						
CD2A1	9	2433	990	40798	41	100781	990	40109	54	131019	**-30238**	689						
CD2A1	10	5038	1200	50254	57	285074	1200	49422	69	348739	**-63665**	832						
CD2A2	2	27	36	769	4	96	36	767	20	533	**-437**	2						
CD2A2	3	21	72	2032	16	343	72	1941	28	590	**-247**	91						
CD2A2	4	177	120	3819	5	918	120	3667	12	2078	**-1160**	152						
CD2A2	5	99	180	6047	9	918	180	5782	14	1391	**-473**	265						
CD2A2	6	607	252	8993	9	5300	252	8627	11	6690	**-1390**	366						
CD2A2	7	285	336	12405	12	3511	336	11938	21	6012	**-2501**	467						
CD2A2	8	1538	432	16218	13	20066	432	15684	14	21289	**-1223**	534						
CD2A2	9	646	540	20559	19	12458	540	19872	19	12320	138	687						
CD2A2	10	3255	660	25376	19	62257	660	24546	18	57065	5192	830						
diff1	5	150	355	15823	17	2480	355	15824	41	6216	**-3736**	**-1**						
diff1	6	235	486	22769	21	4897	486	22302	34	8077	**-3180**	467						
diff2	5	24	355	15801	40	967	355	15802	100	2393	**-1426**	**-1**						
diff2	6	46	486	22731	33	1508	486	22264	76	3484	**-1976**	467						
frank	4	1200	134	2316	5	5804	134	2286	9	10573	**-4769**	30						
frank	5	2000	172	3630	5	10654	172	3446	6	12560	**-1906**	184						
tombot	4	3000	134	2318	4	11668	134	2288	6	17377	**-5709**	30						
tombot	5	2760	172	3630	4	12401	172	3446	5	12800	**-399**	184						

BaseE. In the summary, column **Diff**$_T$ displays the difference in runtime and column **Diff**$_C$ displays the difference in number of clauses. The differences are calculated by subtracting *Reach*'s value from the baseline value. Instances when *Reach* performs better are displayed in green. If enumerating a size resulted in no scenarios, then that size is not included.

In terms of the size of the satisfiability problem, *Reach* and the baseline generate similar problems that are minimally different. *Reach* and **BaseC** produce the same number of primary variables, which is expected, but **BaseE** generates less primary variables than both. In particular, similar to the backend encoding of the singleton multiplicity constraint, the encoding for `exactly` removes all primary variables related to the associated signature, which reduces the number of primary variables equal to the size being explored. *Reach* generates an almost equivalent number of clauses as **BaseC**, with *Reach* on average producing 0.98× the number of clauses as **BaseC**. In contrast, *Reach* often generates more clauses than the **BaseE**; however, the difference is overall is still minor, with *Reach* producing 1.14× more clauses than **BaseE**.

In terms of runtime performance, if a user is enumerating scenarios one by one, all of the average times to find a scenario (T_{avg}) are nominal, and an end user would not experience any difference. From a total runtime perspective, **BaseE** and *Reach*'s performance are nearly identical, with both tools finishing within 1.5 seconds of each other for every execution. Of these, *Reach* finishes faster for 17 of the 24 executions and is on average 1.22× faster. *Reach* and **BaseC** also have very

TABLE III
Comparison of performance across the entire scope. All times are in ms.

Model	Scp	#Scr	Alloy				Base-E		Base-C		Reach		Summary				
			#Pvar	#Cls	T$_{avg}$	T$_{tot}$	T$_{avg}$	T$_{tot}$	T$_{avg}$	T$_{tot}$	T$_{avg}$	T$_{tot}$	R-A	R-BE	R-BC	BE-A	BC-A
arr	3	3771	67	1210	4	18529	5	17163	5	20455	5	17661	**-868**	498	**-2794**	**-1366**	1926
btree	3	817	72	3318	4	4042	6	4840	3	2786	4	3461	**-581**	**-1379**	675	798	**-1256**
cd	7	264	55	3749	6	1834	12	3050	7	1797	5	1389	**-445**	**-1661**	**-408**	1216	**-37**
dll	3	697	72	4530	4	3083	7	5097	6	3941	5	3295	212	**-1802**	**-646**	2014	858
nqueens	5	14	165	11357	113	1595	65	907	37	515	52	730	**-865**	**-177**	215	**-688**	**-1080**
sll	3	344	24	485	1	581	5	1584	3	953	2	702	121	**-882**	**-251**	1003	372
CD2A1	10	9658	1200	48988	70	678509			57	554873	46	441966	**-236543**		**-112907**		**-123636**
CD2A2	10	6655	660	23889	26	176923			19	129257	16	105867	**-71056**		**-23390**		**-47666**
diff1	6	385	486	22097	21	8178			37	14293	19	7377	**-801**		**-6916**		6115
diff2	6	70	486	22059	33	2357			84	5877	35	2475	118		**-3402**		3520
frank	5	3200	163	3191	4	14802			7	23133	5	16458	1656		**-6675**		8331
tombot	5	5760	163	3191	5	28809			5	30177	4	24069	**-4740**		**-6108**		1368

similar performance across the smaller benchmark models. For 22 of the these 24 executions, the two techniques finish within half a second of each other. For **arr**, *Reach* sees the fastest time speedups, but the difference is only 1.2 to 1.5 seconds. Likewise, **BaseC** sees the largest magnitude speed up of 8× for **btree** size 1 and 7× for **cd** size 1, but the time difference is only 106 milliseconds and 6 milliseconds respectively. We do start of see a notable performance difference between *Reach* and **BaseC** when comparing real world models, in which the two techniques only finish within a second of each other 5 times. Across the 23 real world model executions, *Reach* finishes faster 21 times and is 0.68× faster than **BaseC** on average. For **CD2A1** *Reach* provides a notable improvement, finishing over a minute faster for the largest size, 10.

Given the overall improvements in runtime performance, especially for the real world models, we recommend the use of *Reach*. We also believe that the backend encodings for Alloy keywords that assert the exact size of a signature, such as `one` or `exactly`, can be improved by using our CNF encoding. The current encoding is less efficient and also does not fully support all of Alloy's robust grammar, as seen with combining `exactly` with abstract signatures. Moreover, we believe that the preservation of primary variables is helpful (1) to prevent issues, such as the issues outlined for the singleton multiplicity constraint in Section IV, that is a result of loosing this information and (2) to better integrate with applications, such as Hawkeye [23], that utilize primary variables.

C. RQ2: Overhead of Enumerating by Size

Enumerating by size is a tradeoff. We create additional criteria that must be satisfied by any scenario enumerated; however, we gain an expected order to the enumeration. To see the impact of this tradeoff, Table III shows a comparison between the default Analyzer, the baseline techniques and *Reach* from the perspective of enumerating all scenarios up to and including the scope. Column **Model** is the model under evaluation. Column **Scp** is the size and column **#Scr** is the number of scenarios enumerated at that size. The next four columns relate to the Analyzer. Columns **#PV** and **#Cls** show the number of primary variables and the number of clauses. Columns **T**$_{avg}$ and **T**$_{tot}$ show the average time to discover a scenario and the total time to find all scenarios, respectively. For the baseline and *Reach*, we present summary information. The column **T**$_{avg}$ is the average time to discover a scenario across all sizes and column **T**$_{tot}$ reflects the total runtime summed across all sizes. All times are in milliseconds. The last 5 columns show the difference in total runtime for different combinations – "**R**" represents *Reach*, "**A**" represents the default Alloy Analyzer, "**BE**" represents **BaseE**, and "**BC**" represents **BaseC**. The difference are always the left hand side subtracted from the right hand side of the column name.

The total number of scenarios is constant across all three techniques, which is expected, as the information is related to enumerating all scenarios up to the upper bound. Across all techniques, the average time to discover a scenario is nominal if an end user is enumerating scenarios one at a time and inspecting them. However, the difference in average discovery times add up across the span of all scenarios. To illustrate, the most noticeable difference in performance appears for model **CD2A1** in which *Reach* reduces the average time to discover a scenario by 9 milliseconds compared to **BaseC** and 24 milliseconds compared to the Analyzer. When enumerating all solutions, this has a ripple effect leading to the *Reach* finishing 1.8 minutes faster than **BaseC** and 3.9 minutes faster than the Analyzer. All told, on average, when enumerating all scenarios, *Reach* finishes 0.87× faster than the Analyzer, 0.68× faster than **BaseE** and 0.83× **BaseC**. Therefore, we do not suffer any negative performance overhead using *Reach*.

In contrast, the two baseline techniques often take longer than the Analyzer, with **BaseE** finishing on average 1.5× slower and **BaseC** finishing on average 1.2× slower. As a result, with the baseline techniques, there is some overhead to enumerate by size. This overhead is not too significant and for **CD2A1** and **CD2A2**, **BaseC** achieves a speed up over Alloy of about 2 minutes and 48 seconds, respectively. In addition, while using the set cardinality operator is discouraged due to poor runtime performance, **BaseE** was slower than **BaseC** for 5 of the 6 models, implying that the discouraged practice is

TABLE IV
ITERATIVE DEEPENING RESULTS FOR THE ANALYZER AND *Reach*. ALL TIMES ARE IN MS.

Model	Scp	Reach				Alloy									Summary	
		#PV	#Cls	Scr	T$_{tot}$	#PV	#Cls	Scr	T$_{tot}$	1st	10%	25%	50%	100%	Diff$_T$	Diff$_S$
arr	4	84	1519	1206	7810	84	1473	4977	28314	3772	3892	4073	4375	4977	**-20504**	**-3771**
btree	4	104	5562	7280	38735	104	5516	8097	48858	818	1546	2638	4458	8097	**-10123**	**-817**
cd	8	71	5612	846	5431	71	5453	1110	9841	265	349	476	688	1110	**-4410**	**-264**
dll	4	104	6877	1820	9967	104	6831	2517	20475	698	880	1153	1608	2517	**-10508**	**-697**
nqueens	6	198	15099	4	770	198	15088	18	2572	15	15	16	17	18	**-1802**	**-14**
sll	4	40	1069	3081	7107	40	981	3425	12694	1	362	971	1885	3425	**-5587**	**-344**
CD2A1	11	1430	60244	10010	719812	1430	58798	19668	1755121	103	2460	5715	11088	19668	**-1035309**	**-9658**
CD2A2	11	792	30767	1265	42644	792	29069	7920	227959	31	1966	3978	7288	7920	**-185315**	**-6655**
diff1	7	637	30551	559	16111	637	29732	944	23303	24	136	329	571	944	**-7192**	**-385**
diff2	7	637	30493	10	697	637	29674	80	3641	8	19	30	55	80	**-2944**	**-70**
frank	6	214	4945	2560	14174	196	4234	5760	39462	3201	3457	3841	4481	5760	**-25288**	**-3200**
tombot	6	214	4945	3200	20200	196	4234	8960	55663	5761	6081	6561	7361	8960	**-35463**	**-5760**

more efficient than the native support within the Analyzer.

Based on these results, when considering applications of scenario finding that require an end user to enumerate all scenarios, such a test generation [5], [14], [27] or design validation [13], [17], we recommend *Reach* over other strategies.

D. RQ3: Iterative Deepening Application

Within Alloy, it is standard practice to start with a small scope, and then to increase the scope and re-run the Analyzer to build confidence in the model. A study of how user interact with the Analyzer reveals that iterative deepening executions account for 17.6% of Alloy executions [12]. Therefore, one of the main uses we envision of *Reach* is to support iterative deepening. To illustrate, consider when a scope is increased from s to $s+1$. The Analyzer will generate all the scenarios related to size 0, 1, 2, …, s, $s+1$. Unfortunately, all of these scenarios will be intermixed together. However, only the scenarios of size $s+1$ are actually *new* for the user. Notably, *Reach* already largely supports this activity, as we have a distinct enumeration per size. For example, users can re-run a command with scope $s+1$ and only enumerate the scenarios of size $s+1$. To get a sense of how beneficial *Reach* can be, we compare the performance of the Analyzer's default enumeration to using *Reach* to just enumerate scenarios for sizes $s+1$ for our evaluation models.

Table IV captures this comparison. Column **Model** is the model under evaluation and column **Scp** is the scope. The next 8 columns display performance information for each technique. Column **#PV** show the number of primary variables, column **#Cls** shows the number of clauses, column **#Scr** is the number of scenarios and column **T**$_{tot}$ show the total time to find all scenarios. All times are in milliseconds. The next 5 columns highlight the distribution of size $s+1$ scenarios in Alloy's enumeration. Column **1st** shows how many scenarios to explore before encountering the first scenario of size $s+1$, and columns **10%**, **25%**, **50%**, **100%** show how many scenarios are explored before discovering the corresponding percentage of scenarios of size $s+1$. The last two columns provide a summary of the difference: column **Diff**$_T$ is *Reach*'s runtime subtracted from the default Alloy execution and column **Diff**$_S$ is the number of scenarios found by *Reach* subtracted from the number the default Alloy execution finds.

The Analyzer takes on average 2.82× longer to run and discovers on average 3.04× more scenarios than *Reach*. This difference is expected as Alloy's scope is an upper bound. The Analyzer does create slightly smaller satisfiability problem using the same number of primary variables but less clauses. This does not lead to a better overall runtime due to the difference in perspectives on scope. However, the real benefit of *Reach* for iterative deepening is the ability to directly explore scenarios of the new size only. As shown in Column **1st**, for only one model (**sll**) the user would encounter a new scenario as the first enumerated scenarios. Often, the user ends up needing to explore over a hundred scenarios before first encountering a scenario of the new size. Across our benchmark models, we found that a user needs to explore on average 29.52% of the total scenarios before encountering a scenario of size $s+1$. To give a perspective on the distribution of $s+1$ scenarios, the user needs to enumerate on average 38.41% of the total scenarios to encounter 10% of the scenarios of size $s+1$, 50.99% to encounter 25% of the scenarios of size $s+1$, 71.32% to encounter 50% of the scenarios of size $s+1$ and all 100% to encounter all the scenarios of size $s+1$.

Given how common iterative deepening is, we believe that *Reach* can greatly improve this process for the user. In particular, the user is able to more directly explore the new size, which our results highlight is scattered throughout the distribution of the default Analyzer's enumeration.

VI. RELATED WORK

Scenario Enumeration for Alloy. Our technique is closely related to techniques which looks to enhance the Analyzer's scenario enumeration process. One traditional approach is to reduce the number of scenarios through symmetry breaking, where constraints are added to the formula to remove isomorphic solutions [22], [10]. Beyond symmetry breaking, several past projects improve scenario enumeration by trying to narrow what scenarios are generated using a specific criteria, e.g., abstract functions [24], minimality [16], field exhaustiveness [18], and coverage [25], [19]. All of these techniques reduce the number of scenarios by applying some criteria across the entire enumeration. However, the order of scenarios generated is still dictated by the SAT solver's order of discovery. Another approach to enhancing scenario enumeration is to focus on the order in which scenarios are presented. Hawkeye is a tool which allows for users to give on-the-fly guidance to specify how the next scenario should differ with respect to the current scenario [23]. Our approach can work in tandem with all of these enumeration strategies.

Scenario Enumeration at Large. Beyond Alloy, researchers have focused on improving scenario enumeration strategies, .e.g. dedicated search [3], mixing of generators and solvers [11], solver-aided languages [20], and sampling [6]. We believe that the Analyzer generates could be refined by some of these approaches, and further combined with $Reach$. Researchers have also developed verification efforts which utilize scenario enumerating toolsets, e.g. automated test input generation [14], and model counting for reliability analysis [8]. In future work, we plan to explore how $Reach$'s enumeration strategy can benefit the broader adoption of these efforts by allowing the user to discover scenarios of specific sizes for testing. In particular, TestEra and Korat, which already utilizes elements of Alloy [14], [5].

Incremental Analysis for Alloy. Our exploration of $Reach$ in an iterative deepening setting targets improving the performance of the Analyzer in an incremental analysis setting. There are three main bodies of work related to incremental Alloy analysis. First, Titanium uses all the scenarios of the previous model version to calculate tighter bounds for relational variables for the next iteration [2]. Second, iAlloy uses static analysis to determine which commands to avoid re-executing and determine which scenarios to reuse [28]. Third, Platinum slices Alloy models at a boolean level using the CNF formula and reuses scenarios if a redundant CNF slice is detected [30]. Since our technique is built into the CNF encoding used to execute commands, we believe $Reach$ can be integrated with these techniques to help improve their incremental analysis related to scope-level changes.

VII. Conclusion and Future Work

This paper introduces $Reach$, a novel framework for enumerating Alloy scenarios by size. $Reach$ strengthens Alloy's default enumeration by preserving all the scenarios, while at the same time creating an expected order. To achieve this, $Reach$ utilizes a CNF encoding that prioritizes the size of specific signatures and uses this encoding to stage generation by size. Our experimental results highlight that our modified CNF encoding does not create any overhead and, even addresses the limitations of using Alloy's grammar to achieve the same effect. Furthermore, we highlight how $Reach$'s functionality can be applied to efficiently provide an iterative deepening execution environment.As future work, we plan to address refining scenarios by size for Alloy 6, which introduces temporal logic to Alloy and enables *mutable* signatures, which means that the size of signatures can change across different states in the same scenario. We plan to explore how to modify our CNF encoding to enumerate by size in the presence of mutable signatures as well as how to account for the new types of enumeration in Alloy 6.

References

[1] Alloy analyzer Website: http://alloytools.org (2019)

[2] Bagheri, H., Malek, S.: Titanium: efficient analysis of evolving alloy specifications. In: FSE (2016)

[3] Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: ISSTA (2002)

[4] Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J.: User studies of principled model finder output. In: SEFM (2017)

[5] Dini, N., Yelen, C., Alrmaih, Z., Kulkarni, A., Khurshid, S.: Korat-API: A framework to enhance Korat to better support testing and reliability techniques. In: SAC (2018)

[6] Dutra, R., Bachrach, J., Sen, K.: SMTSampler: Efficient stimulus generation from complex SMT constraints. In: ICCAD (2018)

[7] Eid, E., Day, N.A.: Static profiling alloy models. IEEE Transactions on Software Engineering pp. 1–1 (2022)

[8] Filieri, A., Pasareanu, C.S., Visser, W.: Reliability analysis in Symbolic PathFinder. In: ICSE (2013)

[9] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)

[10] Khurshid, S., Marinov, D., Shlyakhter, I., Jackson, D.: A case for efficient solution enumeration. In: SAT (2003)

[11] Kuraj, I., Kuncak, V., Jackson, D.: Programming with enumerable sets of structures. In: OOPSLA (2015)

[12] Li, X., Shannon, D., Walker, J., Khurshid, S., Marinov, D.: Analyzing the uses of a software modeling tool. Electron. Notes Theor. Comput. Sci. **164**(2), 3–18 (2006)

[13] Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using Alloy revisited. In: MODELS (2011)

[14] Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: ASE (2001)

[15] Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": Enriching scenario exploration with provenance. In: FSE (2017)

[16] Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: ICSE (2013)

[17] Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: LISA (2010)

[18] Ponzio, P., Aguirre, N., Frias, M.F., Visser, W.: Field-exhaustive testing. In: FSE (2016)

[19] Porncharoenwase, S., Nelson, T., Krishnamurthi, S.: CompoSAT: Specification-guided coverage for model finding. In: FM (2018)

[20] Ringer, T., Grossman, D., Schwartz-Narbonne, D., Tasiran, S.: A solver-aided language for test input generation. OOPLSA (2017)

[21] Samimi, H., Aung, E.D., Millstein, T.D.: Falling back on executable specifications. In: ECOOP. pp. 552–576 (2010)

[22] Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. In: SAT (2001)

[23] Sullivan, A.: Hawkeye: User guided enumeration of scenarios. In: ISSRE (2021)

[24] Sullivan, A., Marinov, D., Khurshid, S.: Solution enumeration abstraction - A modeling idiom to enhance a lightweight formal method. In: ICFEM (2019)

[25] Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST (2017)

[26] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS (2007)

[27] Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. IEEE Micro (2019)

[28] Wang, W., Wang, K., Gligoric, M., Khurshid, S.: Incremental analysis of evolving alloy models. In: Vojnar, T., Zhang, L. (eds.) TACAS (2019)

[29] Zaeem, R.N., Khurshid, S.: Contract-based data structure repair using Alloy. In: ECOOP. pp. 577–598 (2010)

[30] Zheng, G., Bagheri, H., Rothermel, G., Wang, J.: Platinum: Reusing constraint solutions in bounded analysis of relational logic. In: FASE (2020)