REACH: Refining Alloy Scenarios by Scope

Ana Jovanovic¹ and Allison Sullivan¹

The University of Texas at Arlington, USA
ana.jovanovic@mavs.uta.edu, 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, it is common for users to desire to first validate smaller scenarios, then once confident, move onto validating larger scenarios. However, the Analyzer only presents scenarios in the order they are discovered by the SAT solver. This paper presents Reach, an extension 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 of the enumeration. Reach is available at: https://reachenumerator.github.io/

1 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 [11] is a first-order, relational logic that is used in industry and academia [17,10,26,22,9]. 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 translates Alloy models into KodKod [22] formulas and 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 iterate over.

Scenario enumeration is a highly useful application of formal methods, which has been used to validate software designs [21,30,31,24,12], provide model counting for reliability analysis of systems [8], and to synthesis security attacks for hardware architectures [33,34]. However, effectively applying scenario enumeration relies heavily on the quality of scenarios generated and how easily users
can explore the scenarios. Currently, the Analyzer’s enumeration process is robust: the Analyzer will enumerate all scenarios up to the user provided scope. However, the order in which scenarios are discovered is based only on the order that the back-end SAT solver discovers them. Therefore, there is no guarantee that the next scenario enumerated will be in any way related to the current scenario. This random ordering can discourage users from continuously interacting with the enumerator. Additionally, when a user is trying to validate that their modeled constraints are correct, it is common for the user to desire to move from smaller to larger scenarios. Unfortunately, there is currently no native support for sorting scenarios by size within the Analyzer.

To address this limitation, 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 in the scenario. 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 larger size. To achieve this, Reach restricts the upper bound of the conjunction normal form (CNF) encoding produced by KodKod to match the size being enumerated and modifies the CNF encoding to enforce that a scenario meets the required size because of a specific signature in the model. As a result, Reach additionally applies a semi-sorted ordering to scenarios for the user based on the order signatures are declared in the model. We evaluate Reach in two ways. First, we explore the efficiency of our encoding in comparison to the standard Alloy enumeration and a baseline technique. These experiments reveal that our encoding has no noticeable overhead and can even improve the overall runtime. Second, we perform a case study in which we apply Reach in an iterative deepening setting, where the scope of the command is gradually increased, to highlight how Reach’s functionality can improve Alloy’s analysis in this incremental setting.

In this paper, we make the following contributions:

- **Staged Generation.** We modify the Analyzer’s execution environment to stage generation of scenarios by size. Correspondingly, we update the Analyzer’s interfaces to support unique enumerations for each size.
- **Open Source.** We release our enumeration framework as an open-source extension of the latest stable release of the Analyzer. This allows users to gradually explore our new enumeration technique while still adhering to the original workflow of the Analyzer. Reach can be found at: [https://REACHEnumerator.github.io](https://REACHEnumerator.github.io)
- **Evaluation.** We present an experimental evaluation using a variety of subject models used to evaluate recent advancements to Alloy. We evaluate the overhead of Reach’s focuses search environment and compare Reach’s performance to the default Analyzer enumeration and a baseline technique.
- **Case Study.** We explore how Reach can improve the performance for enumerating scenarios in an iterative deepening setting, a common model development practice.
1. **sig** List {header: lone Node}
2. **sig** Node {link: lone Node}
3. **pred** acyclic {
   4. all l : List | all n : l.header.*link | n !in n.^link
5. }
6. **run** {acyclic} for 3

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

2 Illustrative Example

2.1 Alloy

To highlight how modeling in Alloy works, Figure 1 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. 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 - 5). The predicate Acyclic uses universal quantification (all), set exclusion (!in), transitive closure (^) and reflexive transitive closure (*) to express the idea that “for every list, all nodes in the list are not reachable from themselves following one or more traversal down their link relation.” Commands indicate which formulas to invoke and what scope, bound on the universe of discourse, to explore (line 6). The scope places an upper bound on the size of all signature sets. The command on line 6 asks the Analyzer to search for satisfying assignments to all the sets of the model (List, header, Node, and link) such that the Acyclic predicate is true using up to 3 List atoms and 3 Node atoms.

Figure 2 (a), (b) and (c) graphically displays the first three scenarios found by the Analyzer when the command at line 6 is executed: 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. These scenarios are iteratively discovered by the back-end SAT solver. In Alloy, scenarios are produced by KodKod, which generates a DIMACS formatted 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 atom assignments that can populate a set and ties those to unique integer values, which make up the primary variables of the CNF formula passed to the SAT solver. 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, 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 set List, primary variable “2” reasons over whether or not atom L1 is in set List,
and primary variable “3” reasons over whether or not atom L2 is in set List. If the SAT solver finds a solution in which “1” is positive, then L0 will be in set List for the scenario. 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.

### 2.2 Reach Enumeration

The Analyzer’s default enumeration process prevents repeatedly discovering duplicate scenarios; however, it does not guarantee any sort of ordering to the scenarios. Therefore, Reach modifies Alloy’s enumeration process to enumerate scenarios by size up to the user provided scope. In Reach, the user has the freedom to explore scenarios of any size and can choose when to switch between enumerating scenarios of different sizes. To illustrate Reach’s output, Figure 3 shows the first 8 scenarios enumerated by Reach if the user enumerates from the smallest size to the largest size. To start, the user can iterative over all scenarios of size 0, which is just Figure 3 (a). Then, the user can elect to iterate over all scenarios of size 1, which are the scenarios shown in Figure 3 (b) - (g). Next, the user can move onto scenarios of size 2, starting with the scenario in Figure 3 (h). For the model in Figure 1, Reach enumerates 1 scenarios of size 0, 6 scenarios of size 1, 38 scenarios of size 2 and 299 scenarios of size 3, with an average scenario discovery time of 2 milliseconds and a total runtime of 702 milliseconds.

Besides sorting scenarios by size, Reach also creates a semi-sorted ordering of scenarios by enumerating scenarios based on which signature establishes the size of the scenario. To illustrate, the scenarios in Figure 3 (b) - (e) are enumerated first, since Reach first explores all scenarios that are of size 1 because of the List signature. Then, once all those scenarios are enumerated, Reach explores all scenarios of size 1 because of the Node signature (Figure 3 (f) - (g)). In addition to appending CNF constraints which enforce the size of specific signatures, Reach still appends the CNF clause that prevents discovering duplicate scenarios. As a result, when exploring scenarios related to the signature Node, Reach does not rediscover scenarios Figure 3 (a), (d), and (e).

### 3 Technique

In this section, we describe Reach’s CNF encoding (Section 3.1) and how we apply this encoding to staged generation of scenarios by size (Section 3.2).
Reach creates new CNF clauses that enforce the size of a specific signature. There are two main benefits of this. First, our CNF encoding is simplified, as we don’t need to build a series of clauses to express the 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,” which is in a disjunctive normal form that is more complicated to translate into an equivalent CNF formula. Second, this allows us to automatically present scenarios in a semi-ordered format, as shown in Figure 3 (b) - (g), which can help users more confidently navigate the scenario space. To build our encoding, 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. To illustrate, to enumerate scenarios of size 3 due to the signature List using the primary variables outlined in Section 2, Reach would append the following clauses: \{1\}, \{2\} and \{3\}. These clauses will ensure that atoms L0, L1 and L2 are present in set List for any scenario enumerated.

As an optimization, since Reach creates a distinct enumeration for each size within scope, Reach restricts KodKod’s CNF generation to place an upper bound on all signatures that matches the target size being enumerated. Therefore, Reach’s CNF encoding does not need to append any constraints for the other signatures present in the model, as KodKod’s translation will ensure that the size of all other signatures falls between zero and the targeted size, inclusive. As an example, when enumerating scenarios of size 1 for the command in Figure 1, Reach tells KodKod to generate its CNF translation of the model with an upper bound of 1. As a result, KodKod will only create one primary variable for each signature. In contrast, the default translation, outlined in Section 2, would produce 3 primary variables for each signature. As a result, to enumerate only size 1 scenarios, Reach would need to append additional CNF clauses to prevent the size of all signatures from being too large.

---

### Fig. 3. Reach Enumeration

| (a) | (b) | (c) | (d) |
|-----|-----|-----|-----|
| ∅   | L0  | L0  | L0  |
| N0  | link| N0  | N0  |

### 3.1 CNF Encoding

Reach creates new CNF clauses that enforce the size of a specific signature. There are two main benefits of this. First, our CNF encoding is simplified, as we don’t need to build a series of clauses to express the 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,” which is in a disjunctive normal form that is more complicated to translate into an equivalent CNF formula. Second, this allows us to automatically present scenarios in a semi-ordered format, as shown in Figure 3 (b) - (g), which can help users more confidently navigate the scenario space. To build our encoding, 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. To illustrate, to enumerate scenarios of size 3 due to the signature List using the primary variables outlined in Section 2, Reach would append the following clauses: \{1\}, \{2\} and \{3\}. These clauses will ensure that atoms L0, L1 and L2 are present in set List for any scenario enumerated.

As an optimization, since Reach creates a distinct enumeration for each size within scope, Reach restricts KodKod’s CNF generation to place an upper bound on all signatures that matches the target size being enumerated. Therefore, Reach’s CNF encoding does not need to append any constraints for the other signatures present in the model, as KodKod’s translation will ensure that the size of all other signatures falls between zero and the targeted size, inclusive. As an example, when enumerating scenarios of size 1 for the command in Figure 1, Reach tells KodKod to generate its CNF translation of the model with an upper bound of 1. As a result, KodKod will only create one primary variable for each signature. In contrast, the default translation, outlined in Section 2, would produce 3 primary variables for each signature. As a result, to enumerate only size 1 scenarios, Reach would need to append additional CNF clauses to prevent the size of all signatures from being too large.
Algorithm 1: Staged Generation by Signature

Input: Kodkod solver solver, Previous scenarios prev, Set of signatures sigs
Output: Set of scenarios of the target size.

1. activeSig = 0, addClauses = false // Initialize
2. while activeSig != sigs.size() do
3.  if addClauses then
4.   // Enact Reach’s CNF encoding over relevant primary variables
5.   for i ← 1 to sigs.get(activeSig).getPVariables().size() do
6.     solver.addClause(sigs.get(activeSig).getPVariables().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 ← 1 to prev.size() do
17.    solver.addClause(prev.get(i)) // Add clauses to prevent dups
18.  addClauses = true

3.2 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 3.1 that explicitly enforce the size a given signature. This results in an iterative process where Reach may need to update the CNF clauses when transitioning from focusing on one signature to the next. Currently, Reach prioritizes signatures based on the order in which they are declared in the model.

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, following the encoding outlined in Section 3.1 (lines 4-7). Then, to search for the next scenario, Reach invokes 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, removes all CNF clauses and 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 for
the next signature now indicated by \texttt{activeSig}. Once all signatures have been explored fully, the enumeration ends.

To illustrate, to enumerate scenarios of size 1 for the model in Figure 1, \textit{Reach} will translate the command into KodKod with an upper bound of 1, which means that KodKod will generate just 4 primary variables instead of 24: “1” reasons over if \texttt{L0} is in \texttt{List}, “2” reasons over if \texttt{N0} is in \texttt{Node}, “3” reasons over if \texttt{N0} is in \texttt{L0}’s \texttt{header} relation and “4” reasons over if \texttt{N0} is in \texttt{N0}’s \texttt{link} relation. Then, \textit{Reach} will append the following clause, which assert that the size of \texttt{List} must be 1: 
\{1\}. This will result in the scenarios (b) - (e) in Figure 3. Once the SAT solver is no longer able to find any scenarios, \textit{Reach} removes the clause \{1\} and append the clause \{2\}, which assert that \texttt{N0} must be in \texttt{Node}, resulting in the scenarios in Figure 3(f) and (g). Once the next iteration returns an unsatisfiable result, 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, \textit{Reach} would initiate a separate enumeration. This enumeration would be reasoning over the 12 primary variables generated by KodKod’s translation. \textit{Reach} would start by appending the following clauses, \{1\} and \{2\}, which would ensure that atoms \texttt{L0} and \texttt{L1} are in the set \texttt{List}, respectively (e.g. Figure 3(h)).

4 Implementation

\textit{Reach} is implemented as an extension to the Analyzer v.5.0.1 \[1\]. Importantly, since \textit{Reach} extends the main IDE for Alloy, users can directly explore the new functionality \textit{Reach} provides within their normal development process.

4.1 Updating Alloy’s Workflow

\textit{Reach} augments the enumeration workflow in the Analyzer, as seen in red in Figure 4 which is the result of two main changes. First, \textit{Reach} updates the communication between the Analyzer and KodKod, which is done through the Analyzer’s Reporter, to include information about the target size. This enables KodKod to change the upper bound for its CNF translation to match that of the size actively being enumerated. Additionally, the Enumerator’s iterator is updated to support the staged generation technique outlined in Algorithm 1.

Second, in the Analyzer, a user is only allowed to enumerate one command at a time. For \textit{Reach}, we support multiple simultaneous enumerations, which can be thought of as enumerating multiple commands. Therefore, \textit{Reach} modifies the interaction between the Reporter and the VizGUI interface to enable more than one concurrent enumeration. To support this, \textit{Reach} updates the Analyzer’s logging panel to display information per size, as shown in Figure 5. When the user clicks on any enumeration link, the Analyzer’s standard VizGUI interface appears, with the only modification being the functionality of the backend Enumerator. A user is able to switch between different enumerations by opening the respective VizGUI and a user can have multiple enumerations active at once.
As a result, Reach improves the functionality of the original Analyzer. Specifically, the original Analyzer supports an “execute all commands” functionality; however, the 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.

4.2 Supporting Alloy’s Diverse Signature Grammar

Alloy’s robust grammar allows for signatures to be abstract, extensions of other signatures and to have their own multiplicity constraints. Some of these constructs do not result in primary variables for the SAT solver. In particular, if a signature has the singleton multiplicity constraint (“one”), the constraint always results in the same atom assignment to the signature’s set. Therefore, no primary variables are generated for this signature as there is nothing for the SAT solver to try and solve about what atoms should populate this signature. Unfortunately, this encoding behavior results in scenarios found for a scope that may defy user’s intention. To illustrate, if you have a signature with a singleton multiplicity 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 handle this, Reach appends a fact to the model which asserts the size of the signature is one, which prevents a scenario from being found of size 0. Therefore, Reach will discover these scenarios as scenarios of size of 1, which we believe better matches the user’s expectation.

Abstract signatures also do not generate primary variables, as an abstract signature can never directly have atoms of its own. Instead, all atoms of any signature that extends the abstract signature count towards the size of the abstract signature. Unfortunately, an abstract signature can be extended by more than one 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 and lets KodKod generate the CNF encoding.
5 Evaluation

We evaluate Reach over a benchmark of Alloy models collected from recent advancements made to Alloy [20,35,4]. We address the following research questions in this section:

- **RQ1**: What is the overhead of Reach’s signature-focused enumeration?
- **RQ2**: How does Reach’s enumeration perform in comparison to the Analyzer’s default enumeration and a baseline technique?

### 5.1 Set Up

To determine the runtime impact of Reach, we enumerate scenarios across different models using (1) the Analyzer, (2) a baseline implementation of Reach’s functionality and (3) Reach. To collect performance results, we automatically enumerate all solutions and add in a monitor to collect performance metrics related to size and time. All experiments are performed on Linux Ubuntu 20.04 LTS with 1.8GHz Intel i7 CPU and 16 GB RAM.

In our experiments, we use a broad mixture of 12 models, including models of data structures (arr, btree, dll, sll), models of java class diagram (cd), models of java graph (tombot), models of Constraint Satisfaction Problem (nqueens), and models of Java Collections (frankenvrep).

### Table 1. 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    |
| CD2Alloy1     | 16   | 4    | 12   | 8    | 1    |
| CD2Alloy2     | 10   | 4    | 6    | 2    | 1    |
| diff          | 17   | 4    | 13   | 8    | 1    |
| dll           | 2    | 0    | 0    | 1    | 4    |
| frankenvrep   | 52   | 17   | 35   | 28   | 16   |
| nqueens       | 1    | 0    | 0    | 0    | 2    |
| sll           | 2    | 0    | 0    | 0    | 2    |
| tombot        | 52   | 17   | 35   | 28   | 16   |

To collect performance results, we automatically enumerate all solutions and add in a monitor to collect performance metrics related to size and time. All experiments are performed on Linux Ubuntu 20.04 LTS with 1.8GHz Intel i7 CPU and 16 GB RAM. In our experiments, we use a broad mixture of 12 models, including models of data structures (arr, btree, dll, sll), models of java class diagram (cd), models of java graph (tombot), models of Constraint Satisfaction Problem (nqueens), and models of Java Collections (frankenvrep).
of puzzles (nqueens), and real world models that were generated automatically from software artifacts (CD2Alloy1, CD2Alloy2, diff1, diff2) and of surgical robots (frankenvrep, tombot). Table 1 gives an overview of the size of each model. #Sig is the number of signatures in the model. 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 declared in the model across all signatures. For each model, we select one existing command to execute and use the scope provided by the published model.

**Baseline** To help evaluate the impact of Reach’s encoding, we compare our CNF encoding to a baseline which uses Alloy’s first order logic to explicitly add constraints to the executed command to enforce size. For instance, for the model in Figure 1, to enumerate scenarios that are of size 2, we can append the following constraints:

\[(\#Node = 2 \text{ or } \#List = 2) \text{ and } (\#Node <= 2) \text{ and } (\#List <= 2)\]

Our baseline technique automatically generates formulas using the above strategy for all evaluation models.

5.2 Results

**RQ1: Overhead** Reach’s encoding focuses on exploring scenarios based on which signature results in the size of the scenario. In contrast, our baseline technique shows the performance for just enumerating scenarios by size. To explore what impact Reach’s CNF encoding has, Table 2 shows Reach and the baseline’s performance broken down by each size within scope. Column Model reflects the name of the model under evaluation. Column Sz is the size and column $S\text{Scr}$ is the number of scenarios enumerated at that size. The next five columns relate to the baseline technique’s performance. To show the size of the satisfiability problem, columns #PV, #Var, #Cls show the number of primary variables, the number of variables and the number of clauses generated. To show the overhead, columns $T_{avg}$ and $T_{tot}$ show the average time to discover a scenario and the total time to find all scenarios, respectively. All times are in milliseconds. The next five columns display the same performance information for Reach. If enumerating a size resulted in no scenarios, then that size is not included as a row in the table.

In terms of the size of the satisfiability problem, Reach and the baseline often generate similar sized problems, with the baseline generating slightly larger problems with an average of 19 more variables and 92 more clauses, which, on average, makes the baseline problem 1.03× larger. Reach’s smaller sized satisfiability problems highlights how focusing on a specific signature can slightly decrease the complexity of the problem in comparison to the generic baseline encoding that needs to account for broader behavior. In terms of runtime, the two techniques are similar. For 24 and 47 models, Reach finishes faster while for
18 models the baseline finishes faster. Moreover, for 40 of the 47 enumerations depicted in Table 2, Reach and the baseline technique’s total runtime are within 3 seconds of each other. For the remaining seven executions, Reach finishes between 3.2 and 18 seconds faster than the baseline technique. These seven executions also highlight how small improvements in the average time to discover a scenario can have an impact when applied across the breadth of all discoverable scenarios. For CD2Alloy2, in which Reach finishes 18 seconds faster, Reach discovers scenarios on average just 5 milliseconds faster than the baseline.

While there is not a larger performance difference between the baseline and Reach, the results indicate that Reach’s signature-focused encoding can actually help the SAT solver’s search performance by having the size of a specific signature be a guiding focus, as indicated by the seven executions in which there was a noticeable difference in runtime performance favoring Reach. Notably, four of these seven executions in which Reach has a tangible performance increase over the baseline come from executions over real world models (CD2Alloy1, CD2Alloy1, frankenvrep, tombot). Overall, Reach performance in comparison to the baseline indicates that Reach’s semi-sorted ordering of signatures is able to both provide a known structure to the discovery of scenarios for the user while also often improving the overall runtime performance.

**RQ2: Comparison to the Analyzer’s Enumeration** Reach’s enumeration strategy is a tradeoff. We append additional CNF clauses that must be satisfied by any scenario enumerated, which increases the complexity of the satisfiability problem. However, in exchange, we are able to tailor the generated satisfiability problem to a specific size. To see the impact, Table 3 shows a comparison between the default Analyzer, the baseline technique and Reach. Column **Model** reflects the name of the model under evaluation. Column **Scr** is the size and column **Scp** is the number of scenarios enumerated at that size. The next five columns relate to the Analyzer’s performance. To show the size of the satisfiability problem, columns **#PV**, **#Var**, **#Cls** show the number of primary variables, the number of variables and the number of clauses generated. To show the overhead, 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. Under the baseline and Reach’s headers, the column **T_avg** is the average time to discover a scenario across all scenarios of all sizes and column **T_tot** reflects the total runtime summed across all sizes.

The total number of scenarios is constant across all three techniques, as all information is related to enumerating all scenarios from size 0 to the upper bound. For 10 of the 12 models, the total runtime of all three techniques is within 3 seconds of each other. Therefore, we do not suffer any negative performance overhead when enumerating scenarios by size rather than allowing the SAT solver to determine the order. The most noticeable difference in performance appears for model CD2Alloy1, which is an automatically generated model from a UML class diagram. For CD2Alloy1, both the baseline technique and Reach reduce the average time to discover a scenario, which has a ripple effect leading to the
Table 2. Comparison of Reach and the baseline broken down by each size in scope. All times are in milliseconds (ms). Sizes with zero scenarios are left out.

| Model        | Sz | #/Scr | Baseline | Reach |
|--------------|----|-------|----------|-------|
|              |    | #PV   | #Var     | #Cls  | Tavg  | Ttot |
| arr          | 1  | 255   | 33       | 311   | 663   | 9    | 2543 |
|              | 2  | 1309  | 50       | 602   | 941   | 5    | 7176 |
| arr          | 3  | 2127  | 67       | 674   | 1234  | 4    | 9302 |
| btree        | 1  | 17    | 20       | 186   | 123   | 1    | 22   |
|              | 2  | 240   | 44       | 642   | 1614  | 3    | 861  |
| btree        | 3  | 560   | 72       | 1362  | 3342  | 4    | 2374 |
| cd           | 1  | 1     | 1        | 1     | 1     | 0    | 0    |
|              | 2  | 3     | 5       | 31    | 37    | 0    | 0    |
| cd           | 3  | 2     | 11      | 147   | 237   | 1    | 3    |
| cd           | 4  | 5     | 19      | 318   | 547   | 3    | 15   |
| cd           | 5  | 14    | 29      | 712   | 1514  | 5    | 70   |
| cd           | 6  | 51    | 41      | 1153  | 2521  | 8    | 424  |
| cd           | 7  | 190   | 55      | 1736  | 3857  | 10   | 1980 |
| CD2Alloy1    | 5  | 180   | 350     | 6194  | 12851 | 45   | 8224 |
| CD2Alloy1    | 6  | 208   | 480     | 8587  | 18118 | 37   | 10023|
| CD2Alloy1    | 7  | 568   | 630     | 11432 | 24363 | 32   | 18362|
| CD2Alloy1    | 8  | 1171  | 800     | 14845 | 32198 | 31   | 36946|
| CD2Alloy1    | 9  | 2433  | 990     | 18640 | 40888 | 41   | 100212|
| CD2Alloy1    | 10 | 5038  | 1200    | 22791 | 50386 | 56   | 282323|
| CD2Alloy2    | 2  | 27    | 36      | 515   | 809   | 1    | 49   |
| CD2Alloy2    | 3  | 21    | 72      | 1128  | 2036  | 5    | 121  |
| CD2Alloy2    | 4  | 177   | 120     | 2004  | 3906  | 4    | 713  |
| CD2Alloy2    | 5  | 99    | 180     | 3077  | 6112  | 9    | 924  |
| CD2Alloy2    | 6  | 667   | 232     | 4443  | 9106  | 8    | 4902 |
| CD2Alloy2    | 7  | 285   | 336     | 6006  | 12405 | 10   | 3047 |
| CD2Alloy2    | 8  | 1538  | 432     | 7819  | 16343 | 16   | 24654|
| CD2Alloy2    | 9  | 646   | 540     | 9857  | 20655 | 22   | 14253|
| CD2Alloy2    | 10 | 3255  | 660     | 12114 | 25520 | 25   | 83315|
| diff1        | 5  | 150   | 355     | 7262  | 16299 | 24   | 3662 |
| diff1        | 6  | 235   | 486     | 10022 | 22888 | 31   | 7413 |
| diff2        | 5  | 24    | 355     | 7262  | 16277 | 20   | 493  |
| diff2        | 6  | 46    | 486     | 10022 | 22850 | 35   | 1625 |
| dll          | 1  | 17    | 20      | 332   | 1052  | 2    | 43   |
| dll          | 2  | 120   | 44      | 897   | 2612  | 5    | 636  |
| dll          | 3  | 560   | 72      | 1588  | 4554  | 6    | 3711 |
| frankenvrep  | 4  | 1200  | 134     | 1372  | 2820  | 7    | 8727 |
| frankenvrep  | 5  | 2000  | 163     | 1785  | 3847  | 7    | 14253|
| nqueens      | 1  | 0     | 0       | 0     | 0     | 0    | 0    |
| nqueens      | 1  | 1     | 33      | 364   | 1097  | 8    | 8    |
| nqueens      | 4  | 2     | 132     | 2864  | 8042  | 76   | 152  |
| nqueens      | 5  | 10    | 165     | 4061  | 11365 | 57   | 576  |
| sil           | 1  | 0     | 1       | 0     | 0     | 0    | 0    |
| sil           | 1  | 6     | 4       | 14    | 15    | 1    | 6    |
| sil           | 2  | 38    | 12      | 124   | 176   | 1    | 71   |
| sil           | 3  | 299   | 24      | 339   | 527   | 2    | 655  |
| tombot       | 4  | 3000  | 134     | 1372  | 2822  | 6    | 19156|
| tombot       | 5  | 2760  | 163     | 1785  | 3847  | 5    | 16384|

to techniques finishing nearly 4 minutes faster than the default Analyzer. Similarly, for CD2Alloy2, both techniques finish about a minute faster the Analyzer. Of note, for the nqueens model, the baseline and Reach both noticeably reduce the average time to discover a scenario; however, the difference in overall runtime compared to the Analyzer is less than a second due to the small number of discoverable scenarios.
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. This iterative deepening process produces largely redundant work, as the scenarios found for smaller, previously explored scopes must be re-discovered when the scope increases. To illustrate, when a scope $s$ is increased to $s + 1$, the Analyzer regenerates all the scenarios related to scopes of size 0, 1, 2, ..., $s$ and additionally generates scenarios for scope $s + 1$. However, only the new scenarios of size $s + 1$ are actually new for the user.

A study of how users interact with the Analyzer reveals that users often perform consecutive executions with models that only vary slightly: 59.9% only differ by one formula, 17.6% only differ in the scope, and 11.8% are redundant to past executions [15]. Therefore, Reach’s ability to enumerate scenarios by size, if combined with a monitor to detect if no change or only a scope level change, can improve the Analyzer’s analysis for 26.4% of Alloy executions. To investigate how effective Reach can be at providing native support for iterative deepening, we simulate the process by using Reach’s infrastructure to enumerate scenarios for sizes $s + 1$ and $s + 2$ for our evaluation models from Section 5.

### 6.1 Results

Table 4 shows the comparison between the Analyzer’s enumeration, the baseline technique and Reach’s performance in an iterative deepening setting. Column **Model** reflects the name of the model under evaluation and column **Scp** is the scope. The next 18 columns display performance information for the different techniques. Specifically, performance information for each technique is given in 6 columns with the following meanings: columns **#PV**, **#Var**, **#Cls** show the number of primary variables, the number of variables and the number of clauses generated; column **#Scr** is the number of scenarios enumerated; and to show the overhead, columns **$T_{avg}$** and **$T_{tot}$** show the average time to discover a scenario and the total time to find all scenarios, respectively. All times are in milliseconds.
Table 4. Performance comparison for baseline, streaming, and CD2Alloy algorithms. All times are in ms.

| Dataset | Baseline | Streaming | CD2Alloy |
|---------|----------|-----------|----------|
| #Var    | #PV      | #Cls      | #Scn     |
| Avogadro | 3        | 4         | 5        |
| #ot      | T        | T         | T        |
| #ot      | T        | T         | T        |
| #ot      | T        | T         | T        |

1. All times are in ms.
Since the Analyzer can only treat the scope of a command as an upper bound, for each model, the Analyzer discovers on average $3.0 \times$ more scenarios and takes on average $3.6 \times$ longer than both the baseline and Reach. The Analyzer does create the smallest satisfiability problem, although both the baseline and Reach only create on average $1.02 \times$ and $1.05 \times$ larger models. Across most models, the baseline technique and Reach have similar performance. By design, both techniques generate the same number of scenarios. For 19 of the 24 executions, Reach and baseline finish within 3 seconds of each other. However, there are two notable points of difference. First, for models with a very large number of scenarios, Reach outperforms the baseline. To highlight, for the data structure models (btree, sll), Reach finishes 12.6 and 77.2 seconds faster. For both of these models, the constraints allow for a very broad range of behavior, as indicated by the extremely large number of scenarios discovered (26208 and 42199, respectively). Additionally, for both iterative deepening explorations of the CD2Alloy1 model, Reach finishes 13 and 17 seconds faster than the baseline. The first iterative deepening generates just over 10,000 scenarios and the second generates just under 20,000 scenarios. Second, for one model, CD2Alloy2, the baseline finds all 1265 scenarios of size 11 about 7 seconds faster than Reach.

Overall, the results re-iterate that Reach’s encoding provides an efficient enumeration despite adding constraints to enforce size through targeting specific signatures. At the same time, the results indicate that as the space of valid scenarios increases, Reach enforcement of a soft ordering on signatures can have a positive impact on the overall enumeration runtime. As a result, Reach’s enumeration strategy is a good candidate to support iterative deepening.

7 Related Work

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 [6,27,13]. 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 [29], minimality [21], field exhaustiveness [23], and coverage [31,24]. 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 direct the scenario enumeration process by specifying how the next scenario should differ with respect to the current scenario [28]. Our approach can work in tandem with these enumeration strategies to take advantage of their reduction in the scenario search space while Reach provides a semi-sorted order and efficient staged generation.
Our case study of Reach targets improving the performance of the Analyzer in an iterative deepening setting, which is an incremental analysis problem. 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, with a focus on formula level changes [36]. Third, Platinum slices Alloy models at a boolean level using the CNF formula and reuses scenarios if a redundant CNF slice is detected, and focuses on model structure changes [37]. All three efforts are intended to be more generic than Reach, working with a broader range of incremental changes. In contrast, Reach is specialized to focus on streamlining the Analyzer’s incremental performance related to scope-level changes. However, 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.

In general, beyond Alloy, researchers have focused on improving scenario enumeration strategies, e.g. dedicated search [3], mixing of generators and solvers [10,14], solver-aided languages [24], and sampling [19,7]. We believe that improvements to the scenarios that get generated by the Analyzer could be refined by some of these approaches, and further combined with Reach. Additionally, researchers have developed novel verification efforts which utilize scenario enumerating toolsets, e.g. automated test input generation [18], 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 automated verification efforts, in particular in an incremental setting by allowing the user to discover scenarios of specific sizes for testing.

8 Conclusion and Future Work

A key strength of Alloy is its scenario enumeration toolset the Analyzer. Unfortunately, for a scenario finding toolset to be valuable, the user needs to be empowered to efficiently explore scenarios. The Analyzer’s enumeration process is only guided by a back-end SAT solver. To address this limitation, this paper introduces Reach, a novel framework for enumerating Alloy scenarios by size. To achieve this, Reach utilizes a CNF encoding that prioritizes the size of specific signatures, which helps to further order the scenarios. Our experimental results highlight that our modified CNF encoding does not create a burdensome overhead and, for models with a large scenario space, Reach’s encoding noticeably improves the discovery time. Furthermore, our case study explores how Reach’s functionality can be applied to efficiently provide an execution environment that supports iterative deepening, which is a common model development process in Alloy. In the future, we plan to combine Reach with our command dependency graph from our Alloy repair work in order to be able to automatically detect when a given execution is performing iterative deepening and to the allow users to specify their own ordering to the signatures.
References

1. Alloy analyzer Website: http://alloytools.org (2019)
2. Bagheri, H., Malek, S.: Titanium: efficient analysis of evolving alloy specifications. In: Proceedings of the 2016 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). pp. 27–38 (2016)
3. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: ISSTA (2002)
4. Brida, S.G., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.F.: Bounded exhaustive search of alloy specification repairs. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. pp. 1135–1147 (2021)
5. CheckMate GitHub: https://github.com/ctrippel/checkmate (2019)
6. Crawford, J.: A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In: AAAI-92 Workshop on Tractable Reasoning (1992)
7. Dutra, R., Bachrach, J., Sen, K.: SMTSampler: Efficient stimulus generation from complex SMT constraints. In: ICCAD (2018)
8. Fillieri, A., Pasareanu, C.S., Visser, W.: Reliability analysis in Symbolic PathFinder. In: ICSE (2013)
9. Frias, M.F., Galeotti, J.P., Pombo, C.G.L., Aguirre, N.M.: DynAlloy: Upgrading Alloy with actions. In: ICSE (2005)
10. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: ICSE (2010)
11. Jackson, D.: Alloy: A lightweight object modelling notation. IEEE Transactions on Software Engineering (TSE) 11, 256–290 (2002)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
13. Khurshid, S., Marinov, D., Shlyakhter, I., Jackson, D.: A case for efficient solution enumeration. In: SAT (2003)
14. Kuraj, I., Kuncak, V., Jackson, D.: Programming with enumerable sets of structures. In: OOPSLA (2015)
15. 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)
16. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using Alloy revisited. In: MODELS (2011)
17. Maoz, S., Ringert, J.O., Rumpe, B.: CCDiff: Semantic differencing for class diagrams. In: ECOOP (2011)
18. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: ASE (2001)
19. Meel, K.S., Vardi, M.Y., Chakraborty, S., Fremont, D.J., Seshia, S.A., Fried, D., Ivrii, A., Malik, S.: Constrained sampling and counting: Universal hashing meets SAT solving. In: Beyond NP, Papers from the 2016 AAAI Workshop (2016)
20. Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": Enriching scenario exploration with provenance. In: FSE (2017)
21. Nelson, T., Saghaﬁ, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: ICSE (2013)
22. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: LISA (2010)
23. Poncio, P., Aguirre, N., Frias, M.F., Visser, W.: Field-exhaustive testing. In: FSE (2016)
24. Porncharoenwase, S., Nelson, T., Krishnamurthi, S.: CompoSAT: Specification-guided coverage for model finding. In: FM (2018)
25. Ringer, T., Grossman, D., Schwartz-Narbonne, D., Tasiran, S.: A solver-aided language for test input generation. OOPSLA (2017)
26. Ruchansky, N., Proserpio, D.: A (not) NICE way to verify the Openflow switch specification: Formal modelling of the Openflow switch using Alloy. SIGCOMM (2013)
27. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. In: SAT (2001)
28. Sullivan, A.: Hawkeye: User guided enumeration of scenarios. In: ISSRE (2021)
29. Sullivan, A., Marinov, D., Khurshid, S.: Solution enumeration abstraction - A modeling idiom to enhance a lightweight formal method. In: ICFEM (2019)
30. Sullivan, A., Wang, K., Khurshid, S., Marinov, D.: Evaluating state modeling techniques in alloy. In: SQAMIA (2017). http://ceur-ws.org/Vol-1938/paper-sul.pdf
31. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST (2017)
32. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS (2007)
33. Trippel, C., Lustig, D., Martonosi, M.: CheckMate: Automated synthesis of hardware exploits and security litmus tests. In: MICRO (2018)
34. Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. IEEE Micro (2019)
35. Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for Alloy. In: ASE (2018). https://doi.org/10.1145/3238147.3238162
36. Wang, W., Wang, K., Gligoric, M., Khurshid, S.: Incremental analysis of evolving alloy models. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 174–191 (2019)
37. Zheng, G., Bagheri, H., Rothermel, G., Wang, J.: Platinum: Reusing constraint solutions in bounded analysis of relational logic. In: Fundamental Approaches to Software Engineering. pp. 29–52 (2020)