Solution Enumeration Abstraction: A Modeling Idiom to Enhance a Lightweight Formal Method

Allison Sullivan[1], Darko Marinov[2], and Sarfraz Khurshid[3]

[1] North Carolina A&T State University, Greensboro, USA
aksullivan@ncat.edu
[2] University of Illinois at Urbana-Champaign, Urbana, USA
marinov@illinois.edu
[3] University of Texas at Austin, Austin, USA
khurshid@utexas.edu

Abstract. Formal methods are a key to engineering more reliable systems. In this paper, we focus on an important application of formal methods — enumerating solutions to logical formulas that encode properties of interest. Solution enumeration has many uses, e.g., in systematic software testing, model counting, or hardware analysis. We introduce *solution enumeration abstraction*, a novel idiom that allows users to define data abstractions to enhance solution enumeration by specifying how the solutions must differ, so enumeration creates a high quality set of solutions of a manageable size. We embody the idiom as a technique built on top of Alloy, a well-known lightweight formal method, which is comprised of a first-order relational logic with transitive closure, and a SAT-based analysis engine. Experimental results show that our technique supports a variety of data abstractions, and can substantially reduce the number of solutions enumerated and the time to enumerate them.

1 Introduction

Enumerating solutions to logical formulas that describe properties of interest is a highly useful application of formal methods in many domains. For example, solution enumeration enables validation of software designs [19,33,36,44], systematic testing of implementations [30,35], model counting for reliability analysis of systems [12], or program synthesis for security analysis of hardware [5,46,47]. While solution enumeration has found many uses, its effectiveness relies heavily on the quality and number of solutions enumerated. Creating too similar or too many solutions can lead to redundancy and inefficiency in the supported application, and harm scalability [5, 19, 30, 33, 35, 36, 44, 46, 47].

In this paper, we introduce *solution enumeration abstraction*, a novel idiom that allows users to define data abstractions to enhance solution enumeration by specifying how the solutions must differ. As a result, the collection of solutions enumerated is a tailored subset that focuses on solutions explicitly of value to the user. We implement our idiom for Alloy [19], a declarative, first-order modeling language that is deployed with the analyzer and a solution enumeration toolset. Given an Alloy model and a *scope*, i.e., bound on the universe of discourse,

the analyzer creates a constraint-solving problem in propositional logic and uses off-the-shelf SAT technology [9, 11, 16, 27, 42] to solve it.

Alloy has been used in academia and industry for design and modeling of software systems [3, 6, 20, 22, 48, 51], and for various forms of analyses of code, including deep static checking [13, 21], systematic testing [30], data structure repair [41, 50], and automated debugging [17]. To illustrate one application domain in more detail, Alloy has been recently used to model and analyze not only software but hardware systems. Trippel et al. [5, 46, 47] in the *CheckMate* project use Alloy to model program executions valid in a given microarchitecture in order to explore memory consistency and security properties of such microarchitecture. Their work found new variants of security exploits such as Meltdown and Spectre. From the Alloy perspective, their models are highly interesting as they employ some key structures. In particular, they build graphs (called μhb graphs—for "microarchitectural happens-before" graphs) that capture the precise valid ordering of events on a given microarchitecture. These graphs often give rise to structures that the domain modeling considers equivalent and, as such, it is not needed to explore all those equivalent structures[1].

Our idiom is founded on the principles of *data abstraction*, e.g., as embodied by *abstraction functions*, which map concrete data structures to abstract entities that the structures represent [28]. Abstraction functions naturally occur when abstract data types are used. To illustrate, consider a height-balanced binary search tree that implements a set of integers. An abstraction function can map trees to sets of integers, e.g., a tree with 3 nodes — where 2 is the value in the root, and 1 and 3 are, respectively, the values in the left and the right child of the root — can be mapped to the set {1,2,3}.

Traditional abstraction functions have many well-known uses. They document the key relationships that form the foundation of the implementation of the abstract data type; the implementation must provide behaviors that are correct with respect to the corresponding operations on the abstract data type. Moreover, abstraction functions facilitate analysis of code, e.g., using modular reasoning [26]. Furthermore, they enable synthesis of code, e.g., to synthesize *equals* or *hashCode* methods [38], or iterators over collections [39].

Our newly proposed idiom allows Alloy users to define abstraction functions in their models, and lays the foundation for a novel technique for *abstraction-directed solution enumeration* that restricts the enumeration to only create solutions that are mutually different at the level of the abstract domain, thereby providing the user vital control over solution enumeration. To illustrate, if a binary search tree implements a set, and two trees contain the same set of values, only one of them is generated. In general, an abstraction function maps many concrete structures to one abstract structure. Hence, enumerating (concrete) structures that map to unique abstract values can substantially reduce the number of solutions.

[1] We thank Caroline Trippel for pointing out specific examples of the equivalence properties in the domain of μhb graphs. We abstract these architecture-specific models into more general cases that are easier to present for a broader audience.

Our technique generalizes beyond traditional abstraction functions. For example, the user can simply enumerate solutions that differ with respect to a *subset* of existing relations in their model, e.g., creating a set of binary trees where no two trees have the exact same parent pointers. Another example is where the users want to reduce the number of solutions based on a criteria they desire, e.g., create graphs that do not have the same transitive closure in the context of hardware modeling[1]; the users can encode the criteria using our idiom, and then use our technique to focus enumeration on the relations that are introduced to define the criteria. Another example in the context of hardware modeling is when the user writes an alternative model with the goal to reduce the number of solutions even if doing so impacts some other quality attribute (e.g., readability) of the model[1]; the user can instead embed the alternative model in the abstraction and use it without modifying the original model.

Our technique is complementary to existing approaches for reducing the number of solutions. One such well-known approach is *symmetry breaking*, where additional constraints are added to the formula to remove *isomorphic* solutions to help the solvers prune more [8,23,43], e.g., to remove isomorphic graphs when enumerating binary search trees. Our enumeration technique allows defining and utilizing abstraction functions even in the presence of symmetry breaking constraints. Moreover, our technique can completely *subsume* symmetry breaking, and allows writing symmetry breaking constraints directly as abstractions.

Overall, our new technique enables a key *separation of concerns* in software modeling, where the user can build the model without worrying about refining it to facilitate solution enumeration, which can then be guided by defining an appropriate abstraction using our idiom. We make the following contributions:

- **Idiom**. We introduce an idiom to model abstraction functions in Alloy;
- **Abstraction-directed solution enumeration**. We present an abstraction technique to direct solution enumeration, so the solutions enumerated differ at the abstract level, or stated dually, some solutions that differ at the concrete level are not generated if they map to the same abstract values;
- **Generalization**. We present a generalization of our core technique to support various forms of abstractions to direct solution enumeration; and
- **Evaluation**. We present an experimental evaluation using several subject models; the results show that our technique can substantially reduce the number of solutions and the generation time. Our prototype and the subject models are available online: "`https://github.com/Allisonius/Seabs`".

Related work. Abstraction functions are a central concept in data abstraction [28]. They have been supported by many systems for writing formal specifications, e.g., by the Larch family [18, 25]. Various analyses leverage abstraction functions [26, 38, 39], or more general forms of abstraction [14, 29, 34, 37, 49] for increased efficacy. A key difference between previous work and this paper is our use of abstraction functions in the context of logical formulas to direct solution enumeration using propositional satisfiability solvers.

In the context of Alloy, solution enumeration is commonly used for *scenario exploration* where the user inspects the solutions to validate the Alloy models.

```
module list
one sig List { header: lone Node }
sig Node { elem: Int, link: lone Node }

pred Acyclic { /* no directed cycle */
  all n: List.header.*link | n !in n.^link }
pred NoRepetition { /* unique nodes have unique elements */
  all disj m, n: List.header.*link | m.elem != n.elem }
pred RepOk { Acyclic and NoRepetition }
fact Reachability { List.header.*link = Node } /* no disconnected node */

run RepOk for 3 but 2 int
```

Fig. 1: Alloy model of a singly-linked list of integers.

Several past projects improve solution enumeration by focusing it using different criteria, e.g., symmetry [23], minimality [33], field exhaustiveness [35], and coverage [36, 44]. Our approach is orthogonal to these techniques and can work in tandem with them, e.g., as we show for symmetry breaking (Section 3.3.2).

More generally, solution enumeration is a technique that enables a number of software analyses, e.g., test input generation for automated testing [30] and model counting for reliability analysis [12]. Researchers have developed various optimizations, e.g., dedicated search [4], mixing of generators and solvers [15,24], solver-aided languages [40], and sampling [10,31] for more effective enumeration. We believe our approach can also combine with some of these optimizations, and we plan to explore the integration in future work.

2 Overview

This section describes two illustrative examples to provide an overview of our approach for controlling solution enumeration in Alloy by utilizing abstraction functions. The first example shows a traditional abstraction function for an abstract data type (Section 2.1). The second example shows how our approach addresses a problem in the context of recent work [5, 46, 47] on hardware modeling using Alloy (Section 2.2). We describe the basics of Alloy as needed.

2.1 Singly-linked list and set

Consider modeling in Alloy an implementation of a set of integers using a singly-linked acyclic list of nodes that contain integers without repetition (Figure 1).

The `module` keyword names the model, which contains a set (`sig`) of lists (`List`) and a set of nodes (`Node`). Each element in an Alloy set is an atom. The keyword `one` declares the set of lists to contain only one element — each solution will contain exactly one list. The *field* `header` (in `List`) declares a binary relation of type `List` × `Node`. The keyword `lone` makes `header` a partial function; thus, each list has at most one header. The field `elem` (in `Node`) models the node elements and introduces a total function `Node` × `Int`, where `Int` is the built-in Alloy type that models primitive integers; `link` models the linking structure of the list and is a partial function of type `Node` × `Node`.

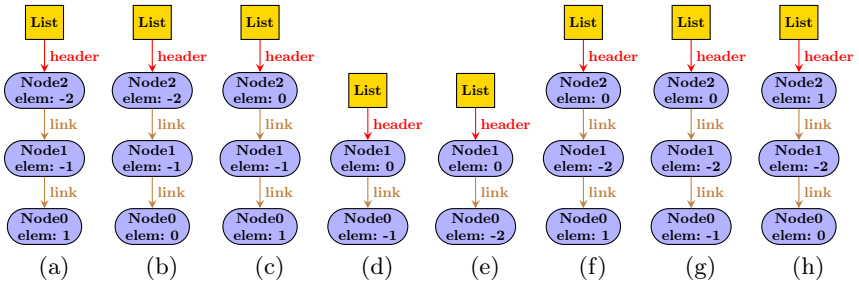

Fig. 2: First 8 solutions generated by the Alloy analyzer. For each structure, the square box is the list atom (*List*), and each ellipse is a node atom and is labeled with its identity (*Node*0, *Node*1, or *Node*2) and integer element (*elem*).

Each predicate (`pred`) defines a formula that can be *invoked* elsewhere. The predicate `Acyclic` defines acyclicity of a linked list using universal quantification (`all`). The expression `List.header.*link` uses relational composition ('`.`') and reflexive transitive closure ('`*`') to represent the set of all nodes reachable from the list's header node. The operator '`^`' is transitive closure; the expression `n.^link` represents the set of all nodes reachable from `n` following one or more traversals along the `link` field. The operator '`!`' is logical negation, and the keyword `in` represents the subset operation. Thus, the predicate `Acyclic` requires the list not to contain a directed cycle. The predicate `NoRepetition` also uses universal quantification; the keyword `disj` makes `m` and `n` distinct. Thus, the predicate `NoRepetition` requires distinct list nodes to contain unique elements and the list to not contain any duplicates. The predicate `RepOk` uses logical conjunction to require the list to be acyclic and free of duplicates.

Each fact defines a constraint that must be satisfied by every solution. The fact `Reachability` requires every node to be in the list so there are no disconnected components in any solution. This fact helps create more meaningful solutions that do not contain parts that are not relevant to the properties modeled.

The `run` command instructs the Alloy analyzer to create a solution with respect to the `RepOk` predicate, the predicates it transitively invokes, and the facts. The command specifies a scope of 3 for all the sigs in the model, i.e., *up to* 3 atoms in each sig, and a bit-width of 2 for integers, i.e., 4 integer values {-2, -1, 0, 1}. The analyzer can enumerate multiple (and if desired, all) solutions. Figure 2 shows the first 8 solutions created by the analyzer. In total, the analyzer creates 41 solutions for the given scope. All these solutions are *non-isomorphic* with respect to the identity of atoms. The Alloy analyzer automatically adds *symmetry-breaking predicates* [8, 43] which, in general, remove many but not all isomorphic solutions. For this scope, these predicates remove all symmetries. While these solutions are non-isomorphic, more than one solution contains the same *set* of node values. For example, the two lists in figures 2(f) and 2(h) represent the same set {-2, 0, 1} with 3 values. In fact, of the 41 solutions found, 6 represent the set {-2, 0, 1}.

```
module listAF
open list
one sig AbsFun { af: set Int }
fact AbsFunDef { AbsFun.af = List.header.*link.elem }
```

Fig. 3: An abstraction function modeled using our idiom..

2.1.1 Idiom for modeling abstraction functions Next, consider modeling the abstraction function for the list representing a set. The abstraction function $\alpha : \mathcal{C} \rightarrow \mathcal{A}$ maps each concrete data structure (in the concrete domain $\mathcal{C}$) to an abstract value (in the abstract domain $\mathcal{A}$). In general, each value in the abstract domain may itself be a structure. In this example, we describe our idiom for the special case when the abstract domain contains sets of integers; Section 3.1 presents a more general treatment. Our modeling idiom has 2 basic steps: 1) add a new singleton sig, e.g., called $AbsFun$, that introduces a field, e.g., af, that models $\mathcal{A}$; and 2) add a new fact that defines the value of the field af (in $AbsFun$) in terms of the fields that model the concrete structure.

Figure 3 shows an Alloy model that defines the abstraction function for our list example. The keyword `open` allows importing another model, which, in this case, is our list model (Figure 1). The sig `AbsFun` and field `af` model the abstraction function. Specifically, `af` introduces a binary relation `AbsFun` $\times$ `Int`; the keyword `set` declares `af` to be an arbitrary relation that maps to a *set* of integers. The expression `List.header.*link.elem` represents the set of all integer elements in the list nodes. The fact `AbsFunDef` constrains the field `af`'s value to equal the set of integers in the list and hence defines the abstraction function.

Our model of the abstraction function introduces a new sig and a new binary relation. Any solution for `RepOk` that is generated with respect to this new model contains a solution for the original model (`list`), i.e., in the concrete domain, and in addition, contains the corresponding value in the abstract domain (given by the value of the field `af`), which allows observing applications of the abstraction function (as well as inspecting concrete structures as before). The number of solutions for the old model (`list`, Figure 1) is the same as the number of solutions for new model (`listAF`, Figure 3) because each solution to `listAF` is a pair that contains a solution to `list` and its abstract value, and each abstract value has at least one corresponding concrete structure.

2.1.2 Using abstraction functions to direct solution enumeration Next, we describe how our idiom enables directing solution enumeration to reduce the number of solutions. Observe that many solutions to the original model (`list`, Figure 1) can map to the same abstract value, e.g., there are 6 lists $l_1, \ldots, l_6$ with exactly 3 nodes with elements -1, 0, and 1, and each l_i ($1 \leq i \leq 6$) maps to the same set {-1, 0, 1}. Our key insight is that if we require enumeration to create solutions that *differ with respect to the fields that model the abstraction function*, the set of all solutions created will not contain any two solutions that have the same value in the abstract domain. We embody this insight into a new solution enumeration technique built on top of the Alloy analyzer's Kodkod back-end (Section 3.2).

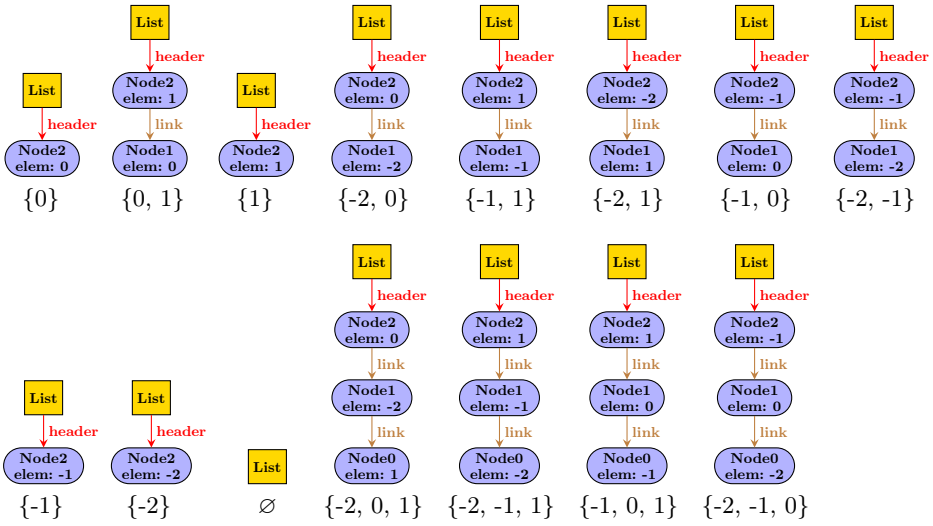

Fig. 4: All 15 solutions enumerated by our technique. Each solution has a linked list (in the concrete domain) and a set (in the abstract domain).

To illustrate, enumerating all solutions for the command "`run RepOk for 3 but 2 int`" with respect to the model `listAF` (Figure 3) using our new technique for directed enumeration creates 15 solutions (Figure 4) instead of the 41 that default enumeration creates for the model `list` (Figure 1). As the scope increases, the reduction in the number of solutions increases. For the command "`run RepOk for 6 but 3 int`" (i.e., up to 6 nodes and 8 integers {-4, -3, ..., 2, 3}), our directed approach creates 247 solutions whereas the default enumeration creates 28,961 solutions. Generating fewer solutions also takes much less time; for this latter command, our directed approach takes 1.2 seconds (total) whereas the default enumeration takes 35.1 seconds (total).

2.2 Graph and transitive closure

Recent work [5, 46, 47] used Alloy to model *microarchitectural happens-before* graphs in the context of hardware modeling, and introduced a number of custom techniques to reduce the number of solutions enumerated by the Alloy analyzer since each solution contributed to a security litmus test. Figure 5 shows a minimal Alloy model that represents the nodes and edges of the graph. For this model, the Alloy analyzer enumerates 152 solutions using the default scope of 3.

One reduction the authors desired was to create one representative graph from each class that has the same transitive closure[1]. Figure 6 shows how our technique allows defining an *abstraction function*, which basically *is* transitive closure, to direct enumeration as desired. Our technique enumerates only 59 solutions for this model (using the default scope), which reduces the number of enumerated solutions by over 2.5x.

Moreover, if self-loops are not relevant in differentiating solutions, the abstraction function can instead be the *reflexive* transitive closure: "`AbsFun.af =`

```
module graph
sig Node { edges: set Node }
```

Fig. 5: Alloy model of a graph simplified from CheckMate [46].

```
module graphAF
open graph
one sig AbsFun { af: Node -> Node }
fact AbsFunDef { AbsFun.af = ^edges }
```

Fig. 6: Directing enumeration to create one representative graph from each class that has the same transitive closure.

`*edges`". Our technique then enumerates 26 solutions for the resulting model (using the default scope), reducing the number over 5.8x over the original model.

3 Abstraction-directed Solution Enumeration

Our basic approach has two parts: 1) an idiom for writing an abstraction function in Alloy (Section 3.1); and 2) a technique for using it for solution enumeration (Section 3.2). Therefore, to utilize our approach, a user first writes an abstraction function for their model and then invokes our solution enumeration technique. While we focus on abstraction functions, our approach supports more general forms of abstractions to guide solutions enumeration (Section 3.3). In future work, we plan to generalize our approach to other solvers, e.g., SMT solvers that allow enumeration [32].

3.1 Idiom

The abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$ maps structures in the concrete domain $\mathcal{C}$ to values in the abstract domain $\mathcal{A}$. In general, each abstract value may itself be a structure. Assume the abstract domain is modeled using k relations $a_1, \ldots, a_k$ ($k \geq 1$). Our idiom for modeling the abstraction function has two basic steps:

1. Add a new singleton sig A with fields $a_1, \ldots, a_k$ that model $\mathcal{A}$; and
2. Add a new fact F to constrain $a_1, \ldots, a_k$ (in A) with respect to the relations that model the concrete domain (to define the abstraction).

Given an initial model m that characterizes the concrete domain, our idiom results in a model m' that consists of m and, in addition, has a new sig A, k new relations $a_1, \ldots, a_k$, and a new fact F. Some examples are shown in figures 3 and 6 in Section 2. Because F simply *defines* the values for the new relations in terms of the relations in m, and the abstraction function α should be total, any solution to m can be extended to a solution for m'. In other words, the number of solutions for m and m' is identical; there is a bijection between solutions of m (each solution is only a concrete structure) and solutions of m' (each solution is a *pair* of a concrete structure and an abstract value). Therefore, simply writing the abstraction function does *not* by itself reduce the number

of solutions enumerated using the Alloy analyzer. However, our new directed enumeration technique enables the reduction (Section 3.2).

There are other ways to model abstraction functions in Alloy. Perhaps the simplest is to use the *function* (`fun`) paragraph, which introduces a named expression. For example, for the singly-linked list model (Figure 1), we can write "`fun AbstractionFunction(): set Int { List.header.*link.elem }`" to define the abstraction function. An advantage is that no new sig (or field) must be added. A disadvantage is that the *return* type (i.e., the type of the expression in the function body), which models the abstract domain, can be just one relation (of arity 1, i.e., a set, or higher). This approach can be extended to support more general return types, e.g., by adding a new sig and fields that model the abstract domain, but doing so reduces this approach to our idiom.

3.2 Directed enumeration

We next describe our key technique for directing solution enumeration to reduce the number of solutions. Our insight is to require solution enumeration to create solutions that each *differ from all previous solutions with respect to the fields that model the abstraction function*, so the set of all solutions created will not contain two different solutions with the same value in the abstract domain.

In Alloy, solution enumeration is provided by the Kodkod [45] back-end, which uses enumerating SAT solvers [9, 11, 16, 27, 42]. When the user desires another solution after a solution, say s, is generated, Kodkod follows the standard practice in modern SAT solvers [11] for solution enumeration and adds a new clause c to the propositional formula in conjunctive normal form (CNF) such that any solution to the new formula will differ from s for at least one boolean variable. This difference is only with respect to the *primary variables*, which Kodkod creates when it translates the model m to a propositional formula p but before p is translated to a CNF formula, because the translation to CNF introduces auxiliary variables, and only the primary variables directly model the relations in m. Different assignments to auxiliary variables may represent the same assignment for primary variables. However, different assignments to primary variables always represent different solutions to the model.

To direct enumeration using the abstraction function, we adapt Kodkod's enumerator to *require the solutions to differ with respect to only the boolean variables that correspond to the fields that model the abstraction function* (and not all fields in the model as done traditionally). Algorithm 1 shows the pseudocode of our directed enumeration. The inputs are a formula ϕ, a scope s, and a set of all relations *Abs* that model the abstract domain. For each relation ρ, $primaryVariables$ (in Kodkod) returns the set of primary variables that model ρ; Kodkod represents each variable using a unique integer id. The method $solve()$ returns a solution if one exists and *null* otherwise. Lines 10–16 show the logic for adding a new clause $negSolAbsVars$ that ensures the next solution differs from the previous ones *with respect to the abstract domain*.

The guard on Line 13 is the key for restricting solutions to differ at the abstract level; without this guard, we get the Kodkod's traditional enumeration (hence we show the guard explicitly rather than iterating over $absVars$).

Algorithm 1: Abstraction-directed solution enumeration.

Input: Formula ϕ, Scope s, Set of relations Abs.
Output: Solutions enumerated with respect to the given abstraction.

```
1  absVars = {}   // empty set of unique ids for variables
2  foreach ρ ∈ Abs do
3      absVars = absVars ∪ primaryVariables(ρ)
4  solver = new Solver(φ, s)   // instatiate Kodkod for solution enumeration
5  while True do
6      solution = solver.solve()
7      if solution == null then break  // no (new) solution found
8      output(solution)
9      // add the negation of the current solution w.r.t. absVars
10     negSolAbsVars = new int[absVars.size()]
11     int j = 0
12     for i ← 1 to solver.numPrimaryVars do
13         if i ∈ absVars then
14             negSolAbsVars[j] = solultion.valueOf(i) ? -i : i
15             j++
16     solver.addClause(negSolAbsVars)
```

Kodkod's $numPrimaryVars$ returns the number of primary variables. The new clause only contains *literals* for the primary variables that represent the relations in Abs; for each such variable v, the clause contains literal v, resp. $!v$, if the value of v is $false$, resp. $true$ in the last solution.

3.3 Generalization

We next describe how our approach generalizes to support a wide range of scenarios for directing solution enumeration to create higher quality solutions. Our approach is not restricted to just abstraction functions. In fact, it does not even require the use of the idiom (Section 3.1) for modeling abstraction functions! In particular, our directed enumeration algorithm does not require the existence of an abstraction function in the Alloy model. The set of relations Abs can be *any* relations that already exist in the model. The user simply marks this set, e.g., in our current tool, as a comma-separated list of relations in the command-line arguments (e.g., `--absRels this/AbsFun,this/AbsFun.af`). Thus, our approach embodies a general technique for directed enumeration where the goal is to create solutions that must differ with respect to a given set of relations. Next, we briefly describe how our approach supports three scenarios that differ from traditional abstraction functions.

3.3.1 Focused enumeration Consider supporting a *goal* or *criterion*, e.g., a *test purpose*, that the enumerated solutions should meet [2]. For example, Alloy users often write additional constraints in their models to focus enumeration, say to create structures with no disconnected components (as illustrated in Figure 1). Our approach provides a new way for users to actively focus enumeration, where solution differences that do not matter can be explicitly defined and utilized.

To illustrate, the user can define the abstraction function `List.header.*link` for the model in Figure 1 to direct enumeration to not create two solutions where the list has the same set of nodes (regardless of whether or not the disconnected components differ).

As another example of focused enumeration, consider enumerating *red-black trees* that are height-balanced binary search trees where each node is colored either red or black [7]. Two red-black trees may be identical as binary search trees and differ only in the node colors. If it is desirable to create solutions that must differ as binary search trees modulo color, our approach directly supports this requirement by using the existing set of relations except color to define the abstraction and direct enumeration of desired red-black trees.

3.3.2 Symmetry breaking Symmetry breaking is a widely used technique for helping SAT solvers prune their search or create fewer solutions [8, 23, 43, 52]. Our approach has a three-fold interaction with symmetry breaking.

Abstraction functions in the presence of symmetry breaking constraints for the concrete domain — our idiom is orthogonal to the use of symmetry breaking and can be used regardless of whether the original Alloy model uses symmetry breaking constraints or not;

Symmetry breaking constraints for the abstract domain — our idiom allows defining symmetry breaking constraints for the abstract domain, e.g., to remove isomorphism at the abstract level. The user simply applies the standard practice of adding symmetry breaking constraints but does so only for the relations that model the abstract domain.

Symmetry breaking constraints as abstraction functions — a model m that has explicit symmetry breaking constraints, e.g., as a fact sb, can be augmented using our idiom such that 1) the abstract domain contains new relations that correspond to the relations that are originally in m, and 2) the abstraction function constrains the abstract domain values to equal the concrete domain structures, and lifts the symmetry breaking constraints sb to the abstract domain, which are no longer enforced at the concrete domain. Doing so gives a clean separation of symmetry breaking constraints from the base model because the purpose of these constraints is only to assist the back-end solvers and direct solution enumeration, and they would otherwise not be a part of the model.

3.3.3 Modeling alternatives An Alloy model typically evolves through different stages, some of which resemble how code evolves. Specifically, the Alloy user has to balance multiple concerns (correctness, analyzability, readability, etc.) when creating their model. Our approach allows a key separation of concerns that enables the user to consider analyzability — with regards to solution enumeration — as a separate concern when writing the model.

To illustrate, recent work on using Alloy for hardware modeling [46] introduced an initial model that is natural to write but leads to too many solutions. They then used an alternative model to make it more useful for solution enumeration, although the alternative made it cumbersome to write and reason about some key expressions that involved transitive closure[1]. The original

model used a binary relation "$edges : Node \times Node$" to model edges, where each node is an $\langle Event, Location \rangle$ pair; this model allows the user to simply write "$\hat{}edges$" for transitive closure. The alternative model removed the indirection of using node atoms in the definition of edges, and used a different relation "$edges' : Event \times Location \times Event \times Location$", to ensure the Alloy analyzer does not enumerate the many combinations that relate node atoms to $\langle Event, Location \rangle$ pairs. While the use of $edges'$ reduces the number of solutions enumerated, the use of transitive closure becomes cumbersome because it can only be applied to a homogeneous binary relation, which the user must now construct from $edges'$ before using the transitive closure operator.

With our approach, the user simply defines the alternative formulation using the abstraction without having to rewrite the original constraints, which are written using natural and intuitive formulas. To illustrate, for the *edges* and $edges'$ example, the user states how the *values* of *edges* and $edges'$ relate. (The model is available online: `https://github.com/Allisonius/Seabs`.) Thus, the abstraction function definition simply relates the structures in the original model to the values in the alternative model — without any need to transform or adapt the original structural constraints to the alternative model.

4 Evaluation

This section presents an experimental evaluation of our approach. We use a suite of 15 Alloy models, including data structures that implement abstract data types [7], models from the standard Alloy distribution [1], and models based on recent work that used Alloy for hardware security analysis[1] [5, 46, 47].

For each model, Table 1 lists the relations in the original model, the relations that define the abstract domain, and the form of abstraction used. The abstract domain relations are either a subset of the relations in the original model, e.g., for *rbt*, or new relations that we introduced for abstraction-directed enumeration and list with their types. The form of abstraction is either traditional abstraction function, e.g., a set implemented as a dynamic data structure (Section 2.1), focused enumeration (Section 3.3.1), symmetry breaking (Section 3.3.2), or modeling alternatives (Section 3.3.3).

The models include object arrays (*objarray*), multi-sets of integers (*multiset*), singly-linked lists (*list*, *listsymbr*), doubly-linked lists (*dll*), binary search trees (*bst*, *bstsymbr*), search trees with parent pointers (*bstp*), min-heaps (*minheap*), red-black trees (*rbt*), general directed trees with integers (*dtree*), general directed graphs (*graph* and *graph*2), and specialized modeling of edges as a map between $\langle Event, Location \rangle$ pairs (*graphsym* and *graphsym*2). The *graph*, *graph*2, *graphsym*, and *graphsym*2 subjects are based on models from CheckMate [46].

Table 2 presents the results of our experimental evaluation. We consider two versions of Alloy: 1) the latest stable release, i.e., Alloy 4.2; and 2) the latest (possibly unstable) build, i.e., Alloy 5.0 [1]. We use each version to compare, for each model, the two techniques: 1) Alloy analyzer's default enumeration for the original Alloy model (*Original*) and 2) our abstraction-directed enumeration for the augmented model that includes the desired abstraction (*Abstraction-directed*

Model	Relations - Original Model	Relations - Abstract Domain	Abstraction
objarray	ObjectArray.array	AbsFun.af: set Object	Traditional – set of objects
list	List.header, Node.elem Node.link	AbsFun.af: set Int	Traditional – set of integers
bst	BST.root, BST.size, Node.key, Node.left, Node.right	AbsFun.af: set Int	Traditional – set of integers
minheap	MinHeap.root, Node.key, Node.left, Node.right	AbsFun.af: set Int	Traditional – set of integers
dll	DLL.header, Node.pre, Node.nxt, Node.elem	AbsFun.af: set Int	Traditional – set of integers
dtree	Tree.root, Node.edges, Node.elem,	AbsFun.af: set Int	Traditional – set of integers
graph	Node.edges	AbsFun.af: Node×Node	Focused enumeration – transitive closure
graph2	Node.edges	AbsFun.af: Node×Node	Focused enumeration – reflexive transitive closure
bstp	BST.root, BST.size, Node.key, Node.left, Node.right, Node.parent	Node.parent	Focused enumeration – parent must differ
rbt	RBT.root, RBT.size Node.key, Node.left, Node.right, Node.color	RBT.root, RBT.size Node.key, Node.left, Node.right	Focused enumeration – search tree must differ
listsymbr	List.header, Node.elem Node.link	AbsFun.af1: List×Node, AbsFun.af2: Node×Int, AbsFun.af3: Node×Node	Symmetry breaking – non-isomorphic structures
bstsymbr	BST.root, BST.size, Node.key, Node.left, Node.right	AbsFun.af: set Int	Symmetry breaking and traditional
multiset	MultiSet.array, MultiSet.length	AbsFun.array: Int×Int, AbsFun.length: Int	Modeling alternatives – sorted array of integers
graphsym	Node.event, Node.location, Node.edges	AbsFun.af: Event×Location× Event×Location	Modeling alternatives – map between $\langle E, L\rangle$ pairs
graphsym2	Node.event, Node.location Node.edges	AbsFun.af1: Event×Location AbsFun.af2: Event×Location× Event×Location	Modeling alternatives – two maps to allow isolated nodes

Table 1: Models used in our evaluation.

enumeration). For each technique, the table lists the number of all (boolean) variables in the SAT encoding ($\#Var$), the number of primary variables ($\#PVar$), the number of all clauses ($\#Cls$), the number of solutions ($\#Sol$), and the time taken to find all solutions ($T_{v.4}$ using Alloy 4.2 and $T_{v.5}$ using Alloy 5.0). For each model, the table also lists the scope (*Scope*), which we selected as the minimum of 10 and the largest scope for which the default enumeration can enumerate all solutions in under 1 minute (so that all experiments finish in a reasonable time).

For all but 2 cases, the number of primary variables is smaller for the original model than the model that includes the abstraction to guide solution enumeration. Being smaller is expected, as modeling the abstraction introduces a new sig and relation(s). For 2 cases (*bstp* and *rbt*), the numbers are the same because the abstraction is simply a subset of the *existing* relations.

As expected, the number of solutions enumerated by our technique is no more than the number enumerated by the default enumeration. For one case (*objarray*), the numbers are the same, because the Alloy's default symmetry breaking behaves the same as the abstraction function we defined. Across all cases, the number of solutions can be reduced by up to 405.3x (*dtree*).

Model	Original						Abstraction-directed enumeration						Scope
	#Var	#PVar	#Cls	#Sol	$T_{v.4}$	$T_{v.5}$	#Var	#PVar	#Cls	#Sol	$T_{v.4}$	$T_{v.5}$	
bst	14036	341	34936	2179	40.9	40.2	13779	357	34099	9	2.9	2.5	9
bstp	8200	290	18954	625	4.6	7.7	8018	290	18211	429	3.6	7.6	7
bstsymbr	12780	332	34220	626	13.4	8.5	12523	348	33383	9	4.0	13.9	9
dtree	1244	75	2751	88769	59.9	59.3	1319	83	3254	219	0.8	0.7	5
dll	2113	132	5119	28961	22.6	24.9	2196	140	5354	247	0.5	0.5	6
list	1874	96	4742	28961	24.0	24.7	1957	104	4977	247	0.5	0.6	6
listsymbr	1874	96	4742	28961	24.0	24.7	1696	180	4232	20160	9.1	8.9	6
minheap	2322	100	5033	15913	13.0	12.0	2397	108	5236	219	0.6	0.6	5
multiset	306	72	489	585	1.0	1.0	1662	144	4257	165	0.3	0.6	3
graph	138	20	200	6344	4.9	4.3	448	36	994	671	0.6	0.9	4
graph2	138	20	200	6344	4.9	4.3	460	36	994	190	0.3	0.9	4
graphsym	360	36	500	915	1.7	1.6	4323	126	7319	148	0.8	0.6	3
graphsym2	360	36	500	915	1.7	1.6	4515	126	7652	170	1.0	0.5	3
objarray	1398	110	3716	11	0.5	0.5	1478	120	3843	11	0.2	0.1	10
rbt	8639	255	20648	84	4.5	11.0	8373	255	19737	65	4.1	13.4	7

Table 2: Performance comparison between the techniques. Times are in seconds.

For Alloy 4.2 (the latest stable release), for all cases, enumerating all solutions using our technique takes less time than the default enumeration. The time speedup using our technique is between 1.1x (*rbt*) to 74.9x (*dtree*). For Alloy 5.0 (the latest, possibly unstable, build), the relative performance results are different for 2 cases (*bstsymbr* and *rbt*) where our technique has a slowdown of 1.6x and 2.9x for *bstsymbr* and *rbt*, respectively; however, the number of solutions is not impacted by the choice of the Alloy version and is still substantially reduced. Moreover, because each solution may be used for expensive post-processing [46] (e.g., to test long-running code executed on each solution), the number of solutions can be more important than the time to generate them. Across the remaining 13 cases, the time speedup using our technique is between 1.7x (*graphsym*2) to 84.7x (*dtree*). Overall, the performances of Alloy 4.2 and Alloy 5.0 are similar.

5 Conclusions

We introduced solution enumeration abstraction, a new modeling idiom that allows Alloy users to define abstractions to enhance solution enumeration. The user specifies how the solutions must differ, so enumeration creates a high quality set of solutions of a manageable size. We implemented our technique on top of the Alloy tool-set and evaluated using a variety of abstractions to show the generality and usefulness of the proposed idiom. The experimental results show that the technique can substantially reduce the number of solutions and the time taken to enumerate them.

Acknowledgments. We thank Caroline Trippel for sharing some of her excellent Alloy models and commenting on an earlier paper draft. This work was partially supported by NSF grants. CNS-1646305, CCF-1718903, CNS-1740916, and CCF-1918189, and an Intel ISRA grant for research on hardware security.

References

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

2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
3. Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the Android permission system. Formal Asp. Comput. (2018)
4. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: ISSTA (2002)
5. CheckMate GitHub: `https://github.com/ctrippel/checkmate` (2019)
6. Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, Power, ARM, and C++. In: PLDI (2018)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press (2009)
8. Crawford, J.: A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In: AAAI-92 Workshop on Tractable Reasoning (1992)
9. CryptoMiniSat Solver Website: `https://www.msoos.org/cryptominisat5/` (2019)
10. Dutra, R., Bachrach, J., Sen, K.: SMTSampler: Efficient stimulus generation from complex SMT constraints. In: ICCAD (2018)
11. Een, N., Sorensson, N.: An extensible SAT-solver. In: SAT (2003)
12. Filieri, A., Pasareanu, C.S., Visser, W.: Reliability analysis in Symbolic PathFinder. In: ICSE (2013)
13. Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: Efficient SAT-based bounded verification using symmetry breaking and tight bounds. TSE (2013)
14. Ghiya, R., Hendren, L.J.: Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: POPL (1996)
15. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: ICSE (2010)
16. Glucose Solver Website: `https://www.labri.fr/perso/lsimon/glucose/` (2019)
17. Gopinath, D., Malik, M.Z., Khurshid, S.: Specification-based program repair using SAT. In: TACAS (2011)
18. Guttag, J.V., Horning, J.J.: Larch: Languages and Tools for Formal Specification (1993)
19. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
20. Jackson, D., Sullivan, K.J.: COM revisited: Tool-assisted modelling of an architectural framework. In: SIGSOFT FSE (2000)
21. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: ISSTA (2000)
22. Khurshid, S., Jackson, D.: Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In: ASE (2000)
23. Khurshid, S., Marinov, D., Shlyakhter, I., Jackson, D.: A case for efficient solution enumeration. In: SAT (2003)
24. Kuraj, I., Kuncak, V., Jackson, D.: Programming with enumerable sets of structures. In: OOPSLA
25. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Softw. Eng. Notes **31**(3) (2006)
26. Leino, K.R.M., Müller, P.: A verification methodology for model fields. In: ESOP (2006)
27. Lingeling, Plingeling, and Treengeling Website: `http://fmv.jku.at/lingeling/` (2019)
28. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification, and Object-Oriented Design (2000)
29. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: VMCAI. pp. 181–198 (2005)

30. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: ASE (2001)
31. 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, AAAI Workshop (2016)
32. de Moura, L., Bjorner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
33. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: ICSE. pp. 232–241 (2013)
34. Pacheco, C., Ernst, M.D.: Randoop: Feedback-directed random testing for Java. In: OOPSLA Companion. pp. 815–816 (2007)
35. Ponzio, P., Aguirre, N., Frias, M.F., Visser, W.: Field-exhaustive testing. In: SIGSOFT FSE (2016)
36. Porncharoenwase, S., Nelson, T., Krishnamurthi, S.: CompoSAT: Specification-guided coverage for model finding. In: FM (2018)
37. Păsăreanu, C.S., Pelánek, R., Visser, W.: Concrete model checking with abstract matching and refinement. In: CAV (2005)
38. Rayside, D., Benjamin, Z., Singh, R., Near, J.P., Milicevic, A., Jackson, D.: Equality and hashing for (almost) free: Generating implementations from abstraction functions. In: ICSE (2009)
39. Rayside, D., Montaghami, V., Leung, F., Yuen, A., Xu, K., Jackson, D.: Synthesizing iterators from abstraction functions. In: GPCE (2012)
40. Ringer, T., Grossman, D., Schwartz-Narbonne, D., Tasiran, S.: A solver-aided language for test input generation. PACMPL OOPSLA (2017)
41. Samimi, H., Aung, E.D., Millstein, T.D.: Falling back on executable specifications. In: ECOOP (2010)
42. SAT4J Solver Website: `https://www.sat4j.org/` (2019)
43. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. In: SAT (2001)
44. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST (2017)
45. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS (2007)
46. Trippel, C., Lustig, D., Martonosi, M.: CheckMate: Automated synthesis of hardware exploits and security litmus tests. In: MICRO (2018)
47. Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. IEEE Micro (2019)
48. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: POPL (2017)
49. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: ASE (2004)
50. Zaeem, R.N., Khurshid, S.: Contract-based data structure repair using Alloy. In: ECOOP (2010)
51. Zave, P.: Reasoning about identifier spaces: How to make chord correct. IEEE Transactions on Software Engineering (2017)
52. Zhang, J.: The generation and application of finite models. Ph.D. thesis, Institute of Software, Academia Sinica, Beijing (1994)