






# Abstract Alloy Instances

Jan Oliver Ringert<sup>1</sup>  and Allison Sullivan<sup>2</sup>  

<sup>1</sup> Bauhaus-University Weimar, Weimar, Germany

<sup>2</sup> The University of Texas at Arlington, Arlington, TX, USA  
allison.sullivan@uta.edu



**Abstract.** Alloy is a textual modeling language for structures and behaviors of software designs. One of the reasons for Alloy to become a popular light-weight formal method is its support for automated, bounded analyses, which is provided through the Analyzer toolset. The Analyzer provides the means to compute, visualize, and browse instances that either satisfy a model or violate an assertion. Understanding instances for the given analysis often requires much effort and there is no guarantee on the order or level of information of computed instances. To help address this, we introduce the concept of abstract Alloy instances, which abstract information common to all instances, while preserving information specific to the analysis. Our abstraction is based on introducing lower and upper bounds for elements that may appear in Alloy's instances. We evaluate computation times and sizes of abstract instances on a set of benchmark Alloy models.

**Keywords:** Alloy analyzer · Instances · Relational logic · Abstraction

## 1 Introduction

Alloy [8–10] is a textual modeling language based on relational first-order logic. Alloy models declaratively express structures and behaviors of software designs. The Alloy Analyzer [2] provides various analyses for finding instances of Alloy models. This analysis is automated due to the use of a bounded scope and an automated translations to SAT solvers, making Alloy a popular light-weight formal method [10]. Alloy has been used to validate software designs [16, 31], to formalize class diagrams [4, 5, 12], to test and debug code [6, 13], to repair program states [21, 30] and to provide security analysis [1, 29].

Simplified, Alloy models consist of signatures, fields, and constraints. Intuitively, a signature introduces a set of atoms, a field relates atoms to other atoms, and constraints define valid configurations – instances – of atoms and their relations. Most Alloy analyses produce a very large numbers of instances, which can number in the hundreds or even thousands, even after automatically filtering symmetric instances [28]. These instances are presented to the user in the order the underlying SAT solver discovers them, which is effectively random. In the Analyzer, users can iterate over instances one by one, visually inspecting

them for correctness. However, given the size of instances and Alloy’s unordered enumeration, this inspection process places a high burden on the user [7, 14]. Therefore, recent work has looked to address this problem by trying to compute more informative, e.g., minimal, instances [15], analyzing “why” and “why not” questions for elements of instances [14], or providing a lightweight order to the enumeration by allowing the user to preserve or change elements of instances [23]. However, all of these approaches deal with valid, complete Alloy instances. Unfortunately, not everything present in an instance is there to satisfy the explicitly executed commands. Alloy instances must also satisfy global properties and no prior work separates the different origins of constraints that influence the shape of an instance.

To address this, we introduce the concept of abstract Alloy instances, a generalization over concrete Alloy instances that abstract away information common to all instances, while preserving information specific to a concrete outcome of the analysis. Our abstraction is based on introducing lower and upper bounds<sup>1</sup> for Alloy’s signatures and fields. The lower bound represents atoms and relations that must be contained in every Alloy instance that concretizes the abstract instance, while the upper bound captures possible additions of atoms and tuples. An abstract instance either represents multiple concrete Alloy instances – those in the upper and lower bounds – or the bounds coincide and the abstract instance is a concrete instance. Our abstraction of Alloy instances is specific to the analysis run by the user, e.g., an Alloy run command sampling specific instances or a check command looking for counterexamples of an assertion.

In this paper, we make the following contributions:

**Abstract Instances** We introduce abstract instances for Alloy that define lower and upper bounds that preserve information in the instance related to satisfying explicitly executed formulas of a command.

**Computing Maximal Abstract Instances** We present an algorithm to generate a *maximal abstract* Alloy instance, which is an abstract instance whose bounds maximize the number of concrete instances represented by the abstract instance.

**Evaluation** We evaluate different performance aspects related to generating abstract instances over a broad benchmark of Alloy models. Our results highlight that there is minor overhead to producing abstract instances, but these abstract instances successfully reduce the information presented to the user.

**Open Source** Our open-source implementation and evaluation materials are available on GitHub [20] and Zenodo [19].

## 2 Example

To introduce the basics of Alloy and computed instances, consider the model of a gradebook shown in Fig. 1. The model describes students, professors, classes, and assignments as well as their relations. Alloy’s main structural elements are

<sup>1</sup> The Alloy Analyzer requires analysis scopes as cardinalities for signatures. Our bounds are refinements of the bounds induced by those scopes, see Sect. 4.

```

1  abstract sig Person {}
2  sig Student, Professor extends Person {}
3  sig Class {instructor: one Professor, assistant: set Student}
4  sig Assignment {associated_with: set Class, assigned_to: some Student}
5  fact {all a : Assignment | one a.associated_with}
6  pred PolicyAllowsGrading(p: Person, a: Assignment) {
7    p in a.associated_with.assistant or
8    p in a.associated_with.instructor
9  }
10 assert NoOneCanGradeTheirOwnAssignment {
11   all p : Person { all a : Assignment {
12     PolicyAllowsGrading[p, a] implies not p in a.assigned_to
13   }}}
14 check NoOneCanGradeTheirOwnAssignment for 3

```

Fig. 1. Alloy model Gradebook from [15]

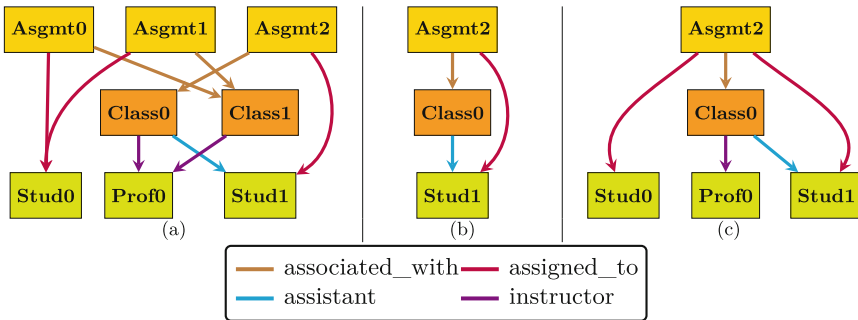


Fig. 2. Two concrete counterexamples (a) and (c) for the check command in Fig. 1 and an abstract instance (b) representing both (a) and (c)

signatures, e.g., signatures `Student` and `Professor`, which both inherit from signature `Person` (Fig. 1, ll. 1–2). Other signatures in the model are `Class` and `Assignment` (ll. 3–4). These signatures declare fields to express relations between the instances of signatures (called atoms). As an example, classes have one professor as instructor and a set of students as assistants (l. 3). Assignments are associated with a set of classes and assigned to at least one (`some`) student (l. 4). A fact restricts all assignments to be associated with exactly one class (l. 5).

The engineers developing the Alloy model want to make sure that no student grades their own assignment. They express a grading policy for persons `p` and assignments `a` in a predicate (ll. 6–8) that allows `p` to grade `a` iff `a` belongs to a class where `p` is an assistant (l. 7) or an instructor (l. 8). An assertion (ll. 10–12) quantifies over all persons `p` and all assignments `a` and asserts that if `p` can grade assignment `a` according to the policy expressed in the predicate then the assignment is not assigned to be solved by `p`.

The Alloy Analyzer allows the engineers to check the validity of the assertion in a bounded scope (l. 14, for up to 3 atoms of each of the signatures). It turns

out that the assertion is not valid and a counterexample is presented to the engineers. The counterexample in Fig. 2(a) is the one of the instances the Alloy Analyzer computes. It shows three assignments, two classes, two students, a professor, and their relations, e.g., `Prof0` is the professor of both classes. It is not easy for the engineers to spot the violation of their assertion, as the engineers need to try to determine which assignment(s) and grader(s) are relevant to the violation.

An abstract instance for Fig. 2(a) is shown in Fig. 2(b). Assignment `Asgmt2` is assigned to student `Stud1` who is also assistant in class `Class0` that the assignment is associated with, i.e., this student can mark their own assignment. Note that the abstract instance is much smaller than the concrete instance and focuses on the reason the assertion is violated, while abstracting away some elements, e.g., the information that `Prof0` is the instructor of the class or that there are multiple assignments not relevant to the violation. The abstract instance is not necessarily a complete Alloy instance, but it can be extended to many concrete instances by adding atoms and their relations. For instance, Fig. 2(c) shows a different concrete instance that extends the abstract instance.

## 3 Preliminaries

### 3.1 Alloy Semantics

We now sketch the semantics of Alloy models as sets of relations. Detailed definitions can be obtained by the descriptions of language elements in [3, 9].

The semantics of Alloy models is defined by a set  $R$  of  $n$ -ary relations  $r \in R$ . Intuitively each signature defines a unary relation and each field defines a relation of the arity of the field plus one. The domain of an  $n$ -ary relation is a subset of  $n$ -ary tuples over a universe `UNIV` of atoms, i.e.,  $dom(r) \subseteq UNIV^n$ . As an example, the domain of the relation for signature `Student` (Fig. 1, l. 2) is a set of atoms and the relation for field `instructor` (Fig. 1, l. 3) is a set of pairs of atoms from relations of signatures `Class` and `Professor`. The set  $R$  of all relations of an Alloy model is defined by the declared signatures, fields, and built-in signatures, e.g., built-in signature `Int`, whose atoms represent the in-scope integers.

Multiplicities of signatures and fields constrain the valuations of relations, e.g., the multiplicity `one` constrains the relation for field `instructor` (Fig. 1, l. 3) to include exactly one pair of `Class` and `Professor` atoms for every `Class` atom. The semantics of facts, predicates, assertions, and expressions are constraints over the tuples in relations  $R$  of the model. As an example, a fact in Fig. 1, l. 5 requires that for every atom in the relation for signature `Assignment` the relation for field `associated_with` contains exactly one tuple.

### 3.2 Alloy Analyses

The Alloy Analyzer enables automated analyses of Alloy models via `run` and `check` commands. `run` commands compute instances satisfying a predicate and `check` commands provide instances violating assertions, i.e., counterexamples.

The analysis of Alloy models by the Alloy Analyzer requires bounds  $B$  for relations  $R$ . Every  $r \in R$  has a lower bound  $LB_B(r) \subseteq dom(r)$  and an upper bound  $UB_B(r) \subseteq dom(r)$  with  $LB_B(r) \subseteq UB_B(r)$  (see [28]). Bounds are derived from user-defined scopes that determine the maximal numbers of atoms in relations for all signatures of the model. As an example, the check command in Fig. 1, l. 14 defines scope **3** setting  $|UB_B(r)| = 3$  for all relations  $r$  of signatures, e.g., the relation for signature **Student**.

We distinguish between two constraints  $M$  and  $C$  on the relations  $R$  of an Alloy model.  $M$  is the constraint defined by the semantics of the model (signatures and facts) and  $C$  is the constraint defined by a command (predicate or assertion). As an example, for the model in Fig. 1, the constraint  $M$  expresses the multiplicities and facts as sketched in Sect. 3.1 and the constraint  $C$  expresses the assertion in Fig. 1, l. 10–12. Thus, we define an Alloy instance as:

**Definition 1 (Alloy instance).** *An instance of an Alloy model is a valuation  $\mathcal{I}$  of relations  $r \in R$  within bounds  $B$  that satisfies the constraints  $M$  and  $C$  denoted by  $\forall r \in R : LB_B(r) \subseteq \mathcal{I}(r) \subseteq UB_B(r)$  and  $\mathcal{I} \models M \wedge C$ .*

Note that Definition 1 does not distinguish between run and check commands, as internally Alloy translates check commands to run commands by negating the assertion. The Alloy instance is then also called a counterexample.

## 4 Abstract Alloy Instances

To introduce abstract Alloy instances, we first define a partial order on bounds  $B$ , i.e., pairs of lower and upper bounds for relations  $R$ .

**Definition 2 (Partial order on bounds).** *Two bounds  $B$  and  $B'$  over relations  $R$  are in a partial order relation  $\preceq$  where  $B' \preceq B$  iff  $\forall r \in R : LB_B(r) \subseteq LB_{B'}(r) \wedge UB_{B'}(r) \subseteq UB_B(r)$ .*

The relation  $\preceq$  is reflexive, transitive, and antisymmetric (because subset inclusion  $\subseteq$  is a partial order). Intuitively, bound  $B$  is greater or equal to bound  $B'$  if  $B$  contains all bounds of  $B'$ , i.e., all lower bounds in  $B$  are smaller and all upper bounds are larger.

As an illustration, consider increasing the scope in Fig. 1, l. 14 from 3 to 5. The bounds have identical lower bounds (empty), but the upper bounds are equal or larger for when increasing scope 3 to scope 5. Typically, bounds for lower scopes are smaller with respect to  $\preceq$  than those obtained for larger scopes. We may write  $\mathcal{I} \preceq B$  for instances  $\mathcal{I}$  where we set  $LB_{\mathcal{I}}(r) = \mathcal{I}(r) = UB_{\mathcal{I}}(r)$  for all  $r \in R$ . Of note, our partial order on bounds is quite different from the partial order on instances defined for Aluminum [15]. First, their order does not include upper bounds, and second, their order is the reverse of ours for lower bounds.

Next, we define abstract instances for Alloy commands.

**Definition 3 (Abstract Instance).** *An abstract instance  $\mathcal{A}$  for model  $M$ , command  $C$ , and bounds  $B$  are bounds  $\mathcal{A} \preceq B$  s.t. all valuations  $\mathcal{I}$  in  $\mathcal{A}$  that satisfy  $M$  also satisfy  $C$ , formally  $\forall \mathcal{I}. \mathcal{I} \preceq \mathcal{A} : (\mathcal{I} \models M) \Rightarrow (\mathcal{I} \models C)$ .*

<pre> 1  sig Professor {} 2  run {one Professor} for 3 </pre> <p style="text-align: center;">(a)</p>	<pre> 1  abstract sig Person {} 2  sig Professor, Student extends Person{} 3  run {some Person} for 3 </pre> <p style="text-align: center;">(b)</p>
--	---

**Fig. 3.** Alloy models demonstrating interesting properties of abstract instances

It is important to define  $\mathcal{I}$  in Definition 3 again as valuations (as before in Definition 1) rather than Alloy instances. Alloy instances would need to satisfy both  $M$  and  $C$ , but for abstract instances the satisfaction of the command constraints  $C$  is only relevant if the model constraints  $M$  are satisfied.

By design, abstract instances abstract away the common constraints  $M$  of the model and preserve the reasons for satisfying commands  $C$ , i.e., all valid extensions (those satisfying the model) of the lower bounds up to the upper bounds must satisfy the analyzed command. As an example, consider the abstract instance in Fig. 2(b) where the lower bound consists of the displayed atoms and relations and the upper bound is unbounded ( $B$ ). Any valid extension of the lower bound, e.g., Fig. 2(a), violates the assertion, as a student grades their own assignment. We are interested in *maximal* abstract instance, i.e., an abstract instance  $\mathcal{A}$  that is maximal wrt.  $\preceq$  (there is no abstract instance  $\mathcal{A}'$  with  $\mathcal{A}' \neq \mathcal{A}$  and  $\mathcal{A} \preceq \mathcal{A}'$ ). A maximal abstract instance represents a maximal number of Alloy instances.

Torlak and Jackson [28] define *partial instances* for KodKod, which is the tool used by the Analyzer to translate the Alloy model into a boolean satisfiability problem, as the lower bounds of the relational problem. The purpose in [28] is to assist the solver. In contrast, our purpose is to provide information to engineers. Since our abstract instances contain lower bounds, they have a flavor of partial instances. However, the lower bounds of an abstract instance  $\mathcal{A}$  may be smaller than KodKod’s partial instances as  $M$  ensures that all represented instances  $\mathcal{I}$  include KodKod’s partial instances. The lower bounds of  $\mathcal{A}$  may also be larger than KodKod’s partial instances, if required for instances  $\mathcal{I}$  to satisfy  $C$ .

## 4.1 Properties of Abstract Instances

We now present six general properties of abstract instances.

First, every concrete instance  $\mathcal{I}$  from Definition 1 interpreted as bounds is also an abstract instance (again setting  $\text{LB}_{\mathcal{I}}(r) = \mathcal{I}(r) = \text{UB}_{\mathcal{I}}(r)$ ) because  $\mathcal{I} \models M \wedge C$ . We say that an abstract instance  $\mathcal{A}$  represents concrete instance  $\mathcal{I}$  iff  $\mathcal{I} \preceq \mathcal{A}$ . Every concrete instance seen as an abstract instance only represents itself, i.e., for all concrete instances  $\mathcal{I}$  and  $\mathcal{I}'$  we have  $\mathcal{I}' \preceq \mathcal{I} \Rightarrow \mathcal{I}' = \mathcal{I}$  (by unfolding the definitions). We are interested in generating abstract instances that represent many concrete instances.

Second, some maximal abstract instances  $\mathcal{A}$  are concrete instances, i.e., reducing any lower or increasing any upper bound of  $\mathcal{A}$  would allow for valuations  $\mathcal{I} \preceq \mathcal{A}$  where  $\mathcal{I} \models M$  but  $\mathcal{I} \not\models C$ . An example is shown in Fig. 3(a) where the instance consisting of one `Professor` atom is a maximal abstract instance.

Third, for a model  $M$ , command  $C$ , and bounds  $B$ , we typically have multiple maximal abstract instances (incomparable wrt. the partial order  $\preceq$ ). As an example, the run command of the model in Fig. 3(b) requires that instances contain at least one atom of type `Person`. We denote by  $s$  and  $p$  the relations defined by signatures `Student` and `Professor`. The abstract instances  $\mathcal{A}$  (at least one student) and  $\mathcal{A}'$  (at least one professor) where  $|\text{LB}_{\mathcal{A}}(s)| = 1$ ,  $\text{LB}_{\mathcal{A}'}(s) = \emptyset$ ,  $\text{LB}_{\mathcal{A}}(p) = \emptyset$ ,  $|\text{LB}_{\mathcal{A}'}(p)| = 1$ ,  $\text{UB}_{\mathcal{A}}(s) = \text{UB}_{\mathcal{A}'}(s) = \text{UB}_B(s)$ , and  $\text{UB}_{\mathcal{A}}(p) = \text{UB}_{\mathcal{A}'}(p) = \text{UB}_B(p)$  are both maximal abstract instances (reducing any lower bound would not ensure the existence of a `Person` atom and upper bounds are already maximal).<sup>2</sup>

Fourth, concrete instances may be represented by multiple maximal abstract instances. As an example, consider the model shown in Fig. 3(b) and the concrete instance consisting of both of a `Student` and a `Professor` atom. This concrete instance is represented by both of the incomparable abstract instances  $\mathcal{A}$  (at least one student) and  $\mathcal{A}'$  (at least one professor). This observation means that maximal abstract instances do not partition the set of instances they represent. There are however always partitions of the set of concrete instances by abstract instances, e.g., the trivial one where we treat concrete instances as abstract ones.

Fifth, from Definition 3, we can see that increasing a lower bound or decreasing an upper bound of an abstract instance  $\mathcal{A}$  (up to upper bounds in  $B$ ) preserves the abstract instance properties (as the set of valuations  $\mathcal{I} \preceq \mathcal{A}$  becomes smaller). In contrast, decreasing a lower bound or increasing an upper bound may allow for new valuations  $\mathcal{I}' \preceq \mathcal{A}$  that satisfy  $M$  but not  $C$ .

Finally, some maximal abstract instances have trivial bounds, e.g., when  $M$  implies  $C$  the requirement  $\mathcal{I} \models M \Rightarrow \mathcal{I} \models C$  from Definition 3 becomes true. Then all lower bounds of maximal abstract instances  $\mathcal{A}$  are empty ( $\forall r \in R : \text{LB}_{\mathcal{A}}(r) = \emptyset$ ) and all upper bounds correspond to upper bounds in  $B$  ( $\forall r \in R : \text{UB}_{\mathcal{A}}(r) = \text{UB}_B(r)$ ). A common example is where an Alloy user executes an empty run command to browse arbitrary instances. In this case, our abstraction, which focuses on the analysis of the command, has nothing to preserve.

## 5 Computing Abstract Alloy Instances

We have seen in Sect. 4.1 that abstract instances are relatively easy to obtain by computing concrete instances and translating them into bounds. However, these abstract instances might not be very informative, as they represent a single concrete instance. We thus aim to compute maximal abstract instances.

Our algorithm for computing a maximal abstract instance is illustrated in Algorithm 1. First, a concrete instance  $\mathcal{I}$  satisfying the model and command constraints  $M \wedge C$  is computed by Alloy's regular solver shown as a call to `solve`( $M \wedge C, B$ ). From this concrete instance we start an iteration that increases the bounds  $\mathcal{A}$  (initialized as  $\mathcal{A} \leftarrow \mathcal{I}$ ) in every iteration of the while loop, i.e.,  $\mathcal{A}' \preceq \mathcal{A}$ . This iteration is necessary as upper and lower bounds may depend on each other. The iteration terminates as lower bounds may only shrink to the

<sup>2</sup> We oversimplify the case of inheritance and relations for illustrative purposes, see our implementation in Sect. 5.1 for a more thorough handling.

---

**Algorithm 1.** Computation of an abstract instance for model  $M$ , command  $C$  and bounds  $B$

---

```

1:  $\mathcal{I} \leftarrow \text{solve}(M \wedge C, B)$ 
2:  $\mathcal{A} \leftarrow \mathcal{I}$ 
3:  $\mathcal{A}' \leftarrow \emptyset$ 
4: while  $\mathcal{A} \neq \mathcal{A}'$  do
5:    $\mathcal{A}' \leftarrow \mathcal{A}$ 
6:    $\text{LB}_{\mathcal{A}} \leftarrow \text{minimize}(\text{LB}_{\mathcal{A}'})$  down to  $\emptyset$ 
7:    $\text{UB}_{\mathcal{A}} \leftarrow \text{maximize}(\text{UB}_{\mathcal{A}'})$  up to  $\text{UB}_B$ 
8: end while
9: return  $\mathcal{A}$ 

```

---



---

**Algorithm 2.** Computation of the `check` used for minimization in Algorithm 1 for  $\text{cand} \subset \text{LB}_{\mathcal{A}'}$  with bounds  $\mathcal{A}'$  and  $B$ , model  $M$ , and command  $C$  from Algorithm 1

---

```

1:  $M' \leftarrow M \cup \text{sig4Bounds}(\text{cand}, \text{UB}_{\mathcal{A}'})$ 
2:  $\text{bounds} \leftarrow \text{expr4Bounds}(\text{cand}, \text{UB}_{\mathcal{A}'})$ 
3: return  $(\text{solve}(M' \wedge \text{bounds} \wedge \neg C, B) == \text{UNSAT})$ 

```

---

empty set ( $\emptyset$ ) and upper bounds may grow at most up to  $B$ . The algorithm then returns a maximal abstract instance  $\mathcal{A}$  (by construction of the bounds).

To minimize and maximize bounds we use Delta Debugging [32]. Delta Debugging computes minimal subsets of a set that satisfy a `check` criterion. We can easily convert our bounds to sets (e.g.,  $\bigcup_{r \in R} \text{LB}_{\mathcal{A}'}(r)$  is a set of atoms and tuples) and back by tracking Alloy’s type information.

We show our implementation of `check(cand)` in Algorithm 2. A candidate  $\text{cand} \subset \bigcup_{r \in R} \text{LB}_{\mathcal{A}'}(r)$  is valid if the abstract instance criterion from Definition 3 is satisfied, i.e., for all  $\mathcal{I}'$  within the bounds of the abstract instance  $\mathcal{I}' \models M \Rightarrow \mathcal{I}' \models C$ . In Algorithm 2 the lower bounds we use for valuations  $\mathcal{I}'$  are  $\text{cand}$  and the upper bounds are  $\text{UB}_{\mathcal{A}'}$  (for maximizing  $\text{UB}_{\mathcal{A}'}$  `check` uses  $\text{cand}$  and  $\text{LB}_{\mathcal{A}'}$ ). We encode these as the constraint  $\text{bounds}$  (see Sect. 5.1). Finally, to evaluate the abstract instance criterion, we invoke the solver and convert the universal quantification over valuations  $\mathcal{I}'$  into an existential one that satisfies  $M$  and violates  $C$ .

## 5.1 Encoding of Bounds in Alloy

Ideally, we would like to pass bounds  $\mathcal{A}$  instead of  $B$  to Alloy’s solver KodKod [28]. However, the bounds used by KodKod are different from the ones indicated in Definition 1, Definition 2, and Definition 3, e.g., KodKod does not support inheritance and thus additional relations may be created in the translation to KodKod. Since our prototype implementation stays on the abstraction level of Alloy, we encode bounds as additional signatures (`sig4Bounds`) and constraints (`expr4Bounds`).

Method `sig4Bounds` creates signatures with multiplicity `lone` extending the primary signatures<sup>3</sup> of the model to represent atoms, e.g., signatures created for

---

<sup>3</sup> Alloy distinguishes between primary and subset signatures where atoms of subset signatures always also belong to primary signatures.



```

1 lone sig Asgmt0, Asgmt1, Asgmt2 extends Assignment {}
2 lone sig Class0, Class1 extends Class {}
3 ...
4 (one Asgmt2) and (one Class0) and (one Stud1)
5 (Class0 -> Stud1 in assistant) and (Asgmt2 -> Stud1 in assigned_to)
6 ...
7 (Person = Student + Professor) and (Student = Stud0 + Stud1) and ...
8 assistant in (Class0 -> Stud0 + Class0 -> Stud1)

```

**Fig. 4.** Excerpt of encoding of bounds from Fig. 2 via signatures and constraints

the atoms shown in Fig. 2(a) are declared in Fig. 4, ll. 1–2. Method `expr4Bounds` then uses this representation of atoms to express lower bounds by requiring the existence of the atoms and tuples, e.g., for the lower bound in Fig 2 (b) see Fig. 4, l. 4. Similarly, tuples are required by lower bounds, e.g., in Fig. 4, l. 5. Whereas the constraints of lower bounds are local for individual elements, upper bounds are global in the sense that we must constrain all atoms of a signature, e.g., Fig. 4, l. 7, and all tuples of a relation at once, e.g., Fig. 4, l. 8. The upper bound constraints in Fig. 4, ll. 7–8 are an excerpt of upper bounds initialized from the instance in Fig. 2(a).

The use of a generic minimizer in Algorithm 1, which is unaware of dependencies between tuples and atoms, may lead to cases where a tuple is present in the lower or upper bounds when one of its atoms is not. In both cases, `expr4Bounds` does not generate a constraint for the tuple, i.e., the constraint for the lower bound is weaker and might fail (the larger *cand* set with the missing atom will then be searched) and the constraint for the upper bound might be stronger and may succeed (the larger set with the additional atom will then also be checked).

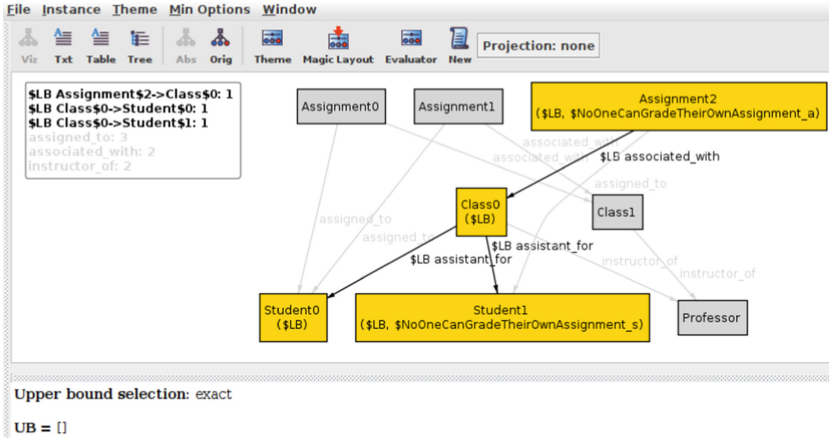
Note that our implementation uses APIs of the Alloy Analyzer and does not explicitly create the syntax shown in Fig. 4. This has two advantages: (1) we do not need to disambiguate fields with same names and (2) we can also constrain signatures marked as `private`, e.g., the signature `Ord` in Alloy’s `ordering` module.

## 5.2 Running Time Complexity

We estimate the running time complexity of the algorithm in terms of Alloy’s solver calls by Algorithm 1. Minimization and maximization with Delta Debugging has a running time in  $O(N^2)$ . The while loop in Algorithm 1 leads to an overall time complexity in  $O(N^4)$  (worst case where every iteration adds/removes only one element). In Algorithm 1, l. 6  $N = |\text{LB}_{\mathcal{A}'}|$  with  $|\text{LB}_{\mathcal{A}'}| \leq |\mathcal{I}|$ . In Algorithm 1, l. 7  $N = |\text{UB}_{\mathcal{A}'}|$  with  $|\text{UB}_{\mathcal{A}'}| \leq |\text{UB}_B|$ . In both cases,  $N \leq \sum_{r \in R} |\text{dom}(R)|$ . Looking at the structure of Alloy models with signatures *sigs*, fields *fields* and scope *maxScope*, we have  $N \in O(\text{maxScope} \cdot |\text{sigs}| + |\text{fields}| \cdot \text{maxScope}^{\text{maxArity}(\text{fields})})$ . Note that the size of  $\mathcal{I}$  is often much smaller, but this is not the case for  $|\text{UB}_B|$ .

## 5.3 Different Upper Bound Kinds

We have defined abstract instances in Definition 3 without any restriction on the shape of bounds. The running time analysis in Sect. 5.2 shows that restrictions



**Fig. 5.** Abstract instance visualized on top of a concrete instance (UB is unbounded)

on the kind of upper bounds we compute may improve running times. We have implemented four kinds of upper bounds and briefly describe these here.

**Exact.** Exact upper bounds are the most natural variant used in Sect. 5. Every atom and every tuple have to be considered when maximizing the upper bound of an abstract instance. The number of elements to find a maximal subset for is in  $O(\maxScope \cdot |\mathit{sigs}| + |\mathit{fields}| \cdot \maxScope^{\maxArity})$ .

**Instance or None.** The upper bound for each signature and field  $r \in R$  is as in the concrete instance  $UB_{\mathcal{A}}(r) = UB_{\mathcal{I}}(r)$  or unrestricted  $UB_{\mathcal{A}}(r) = UB_B(r)$ . The number of elements to find a maximal subset for is in  $O(|\mathit{sigs}| + |\mathit{fields}|)$ .

**Instance.** The upper bound is always the instance. There is no call to `maximize` in Algorithm 1, l. 7 and  $UB_{\mathcal{A}}$  remains as initialized from  $UB_{\mathcal{I}}$ .

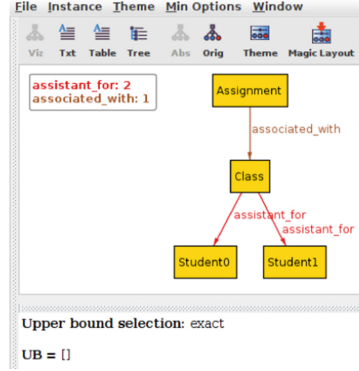
**None.** We do not consider any restriction of the upper bound. There is no call to `maximize` in Algorithm 1, l. 7 and  $UB_{\mathcal{A}}$  is instead treated as  $UB_B$ .

The latter two bound kinds reduce the overall running time complexity from  $O(N^4)$  to  $O(N^2)$ . For the first three kinds an abstract instance always exists (in the worst case it only represents  $\mathcal{I}$ ); however, kind *None* is incomplete, i.e., some concrete instances require upper bounds (see Fig. 3(a)).

## 5.4 Implementation and Visualization

We have implemented our work as an extension to the latest stable release of the Analyzer, version 6.0.0 [2] (our implementation is available from [20]). Importantly, since we extend the main IDE for Alloy, users can maintain their current workflow while gradually exploring the new functionality. Users can access abstract instances during the standard enumeration process, which occurs in the `VizGUI`. When viewing a specific instance, the user is able to select the “Abs” button which will update the active display to present the associated abstract instance. The lower bound of the abstract instance is displayed visually in the main panel, while the upper bound is conveyed textually below.

Users are given two display options. First, the “Over Instance” view will highlight the lower bound of the abstract instance, with any excluded portion of the Alloy instance grayed out. As an example, for the Gradebook model from Fig. 1, Fig. 5 shows a possible instance using the “Over Instance” visualization. Second, the “Independent” view which will visualize just the lower bound of the abstract instance. As an example, Fig. 6 shows the same instance as that in Fig. 5 but with the “Independent” view. In addition, users can also select which of the four upper bound kinds from Sect. 5.3 to use. The user can switch back to the original instance using the “Orig” button.



**Fig. 6.** Abstract instance visualized independently of any concrete instance (UB is unbounded)

## 6 Evaluation

To evaluate abstract instances, we use a collection of 78 benchmark Alloy models. We executed all experiments on Ubuntu 22.04 LTS (64 Bit) with an Intel Core i7-7700 K 4.20 GHz processor and 32 GB RAM. We use Alloy’s default options and selected MiniSatJNI as SAT solver.

We address the following research questions, where by abstract instance we always mean maximal abstract instance:

- **RQ1:** What is the time overhead of generating an abstract instance?
- **RQ2:** How do the sizes of abstract and concrete instances compare?
- **RQ3:** As concrete instances are enumerated, what is the diversity of the underlying abstract instance?
- **RQ4:** What is the time/size/diversity impact of the upper bound kind?

*Set Up.* To evaluate abstract instances, we rely on meaningful commands. Therefore, we focus on two collections of models used to illustrate how Alloy works: models from the Alloy textbook [9] (**Book**) and models included as

examples in the official Analyzer release (**Examples**). In addition, we include models used to evaluate recent automated repair work for Alloy (**ARepair**) whose commands execute faulty portions of the model. For each model, we consider every command present; however, we filter out commands that are: (1) empty (“run {}”), which only execute the facts of the model, (2) commands that produce no instances and (3) commands that use temporal logic, which is new

**Table 1.** Subjects

Subject	#M	Avg.S	Avg.R	#C	Avg.C
<b>ARepair</b>	33	4.27	2.91	36	1.10
<b>Book</b>	28	4.46	3.20	34	1.21
<b>Example</b>	17	6.71	7.76	41	2.41

to Alloy 6 and not currently supported by our implementation. After this filtering, we are left with 28 **Book** models, 17 **Example** models, and 33 **ARepair** models. For each collection of models, Table 1 gives the following information to convey the size and number of models in the benchmarks: Column **#M** shows the number of models, **#Avg\_S** is the average number of signatures per model, **#Avg\_R** is the average number of relations per model, **#C** is the total number of commands, and **#Avg\_C** is the average number of commands per model. For each command, we enumerate up to the first 10 instances, with an enumeration timeout of 10 min. For research questions 1–3, we use *Exact* upper bounds as a default.

### 6.1 RQ1: Overhead

Abstract instances are generated from an existing concrete instance that has been enumerated for a command. To explore the overhead of this process, Fig. 7(a) depicts a boxplot that shows the distribution of the ratio between the time it takes to generate the first abstract instance compared to the time to generate the first concrete instance. A ratio larger than 1 means the abstract instance took longer to produce than the paired concrete instance. We consider only the time to the first instance because the Analyzer uses incremental SAT solvers; therefore, the time to produce the first instance includes all the novel effort to resolve the executed constraints, while future instances are often quickly produced due to the ability to reuse previous work. There are 38, 33 and 34 abstract-concrete instances pairs in the boxplot for **ARepair**, **Book** and **Example** respectively. **Example** excludes two commands which timed out generating the first instance. The first quartile to third quartile ratios range from 2.14 to 4.5 for **ARepair**, from 1.84 to 11.81 for **Book** and from 5.35 to 62.10 for **Example**.

These results indicate that abstract instances frequently take longer to produce compared to their paired concrete instance. However, this does not mean abstract instances have a prohibitive overhead. In particular, finding concrete instances is quick: all concrete instances are produced in less than .5s. In comparison, 61 of the abstract instances take less than 2s to produce, while 34 abstract instances take between 2s and 10s to produce, which is a slight overhead but not unreasonable. However, 17 abstract instance take longer than 10s to produce, including 5 abstract instances that take longer than one minute. These 5 abstract instances all use a larger scope than the default scope (3) and include the “ordering” module. In fact, across all three data sets, all but two outliers capture abstract instances that come from models that uses the “ordering” module. While **ARepair** contains 1 abstract instance that includes the “ordering” module, **Book** has 10 and **Example** has 30, which directly translates into the increasingly larger ratios observed in Fig. 7(a).

On average, abstract instances have a minor overhead to produce; however, if the “ordering” module is present, the time overhead quickly increases. The “ordering” module bloats the time to generate an abstract instance because the module increases the size of the upper bound since it places an ordering on the atoms of a signature and all possible orders must be considered.

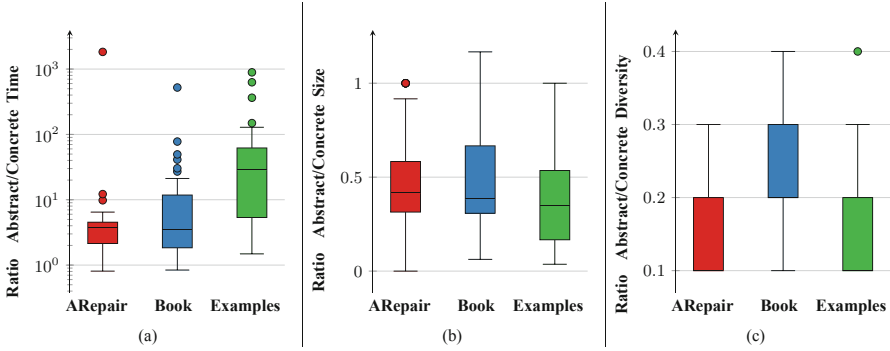


Fig. 7. Comparison of abstract instance to concrete instance performance

### 6.2 RQ2: Size Comparison

Given that abstract instances are meant to refine concrete instances, we expect that abstract instances are, on average, smaller than concrete instances. To explore if this holds, Fig. 7(b) depicts a boxplot showing the distribution of size ratios, which is calculated by taking the size of the abstract instance and dividing it by the size of the corresponding concrete instance used to produce the abstract instance. We define the size of a concrete instance as the number of its atoms and tuples and we define the size of an abstract instance as the number of atoms and tuples in the lower bound plus the number of relations constrained in by the upper bound. A ratio of less than 1 means the abstract instance is smaller than the paired concrete instance. There are 339, 253, and 282 abstract-instance pairs in the boxplot for **ARepair**, **Book** and **Examples** respectively. The first quartile to third quartile ratios range from 0.31 to 0.58 for **ARepair**, from 0.31 to 0.67 for **Book** and from 0.17 to 0.53 for **Example**.

The results highlight that on average the abstract instance is smaller than the concrete instance, and often the abstract instance reduces the size by at least half. Rarely, the abstract instance ends up the same size or larger than the concrete instance. This occurs just 6, 6, and 1 times for **ARepair**, **Book** and **Examples** respectively. In the opposite direction, for 30 **ARepair** instances, the abstract instance produced is an empty instance. This is expected as all of these instances are associated with the model “student16” that is under-constrained due to the student failure to write anything for the predicates. As a result, when the faulty predicates are run, only the facts of the model are enforced. The results also highlight that while models that use the “ordering” module will have longer abstract instance generation times, these models do not consistently produce larger abstract instances, as **Example** models has the smallest quartile 1 to quartile 3 range despite having the most models that use “ordering.”

We find that the abstract instance noticeably reduce the size of the concrete instance, highlighting that commonly half or more of the information in an instance is there regardless of the explicitly executed constraints of the command.

### 6.3 RQ3: Diversity

To gain insight into how many different abstract instances the user will encounter, Fig. 7(c) depicts a boxplot showing the distribution of diversity ratios, which is calculated by taking the number of unique abstract instance and dividing it by the number of concrete instances for each command. We include only those commands that were able to produce 10 concrete instances. A ratio of less than one means there were fewer unique abstract instances than concrete instances, with a ratio of 0.1 meaning all 10 concrete instances reduced to the same abstract instance. There are 33, 21 and 25 commands in the boxplot for **ARepair**, **Book** and **Examples** respectively. The first quartile to third quartile ratios range from 0.1 to 0.2 **ARepair**, from 0.2 to 0.3 **Book** and from 0.1 to 0.2 **Example**. The median is equivalent to the 1st quartile for all data sets.

The results demonstrate that **ARepair** and **Example** models frequently produce only 1 or 2 abstract instances for the first 10 instances enumerated. For both data sets, 17 of their commands produce a single abstract instance. In contrast, **Book** models have a little bit more diversity, with only 4 commands producing a single abstract instance. However, even for **Book**, no command produces more than 4 unique abstract instances. Since a user is likely to inspect the first few instances, but maybe not too many more, our results indicate that the user is often looking at instances that all satisfy the explicitly executed commands in the exact same way. Therefore, as future work, we plan to explore how to directly enumerate unique abstract instances, which will ensure users are able to quickly view diverse ways the command can be satisfied.

### 6.4 RQ4: Impact of Upper Bound Kind

As outlined in Sect. 5.3, abstract instances can be calculated with four different upper bounds. While *Exact* is the default, Fig. 8 compares the performance across all four upper bound kinds. In Fig. 8, **E** represents *Exact*, **I** represents *Instance*, **IoN** represents *Instance or None* and **N** represents *None*. Across the performance metrics, *None* consistently represents fewer data points as *None* is incomplete for 41 of the commands in the evaluation. For the other three bounds, there is a minor difference in the number of data points, as some of the more time expensive upper bound kinds occasionally timeout while enumerating instances.

Figure 8(a) compares the overhead of each upper bound kind by depicting the ratio between the time to generate the abstract instance and generate the concrete instance. We again look at the time to produce the first instance. There are 103, 105, 105 and 64 abstract-concrete pairs in the boxplot for *Exact*, *Instance*, *Instance or None* and *None* respectively. The results in Fig. 8(a) highlights that on average, *Exact* is the most expensive upper bound and *Instance* is the fastest upper bound, both of which is expected.

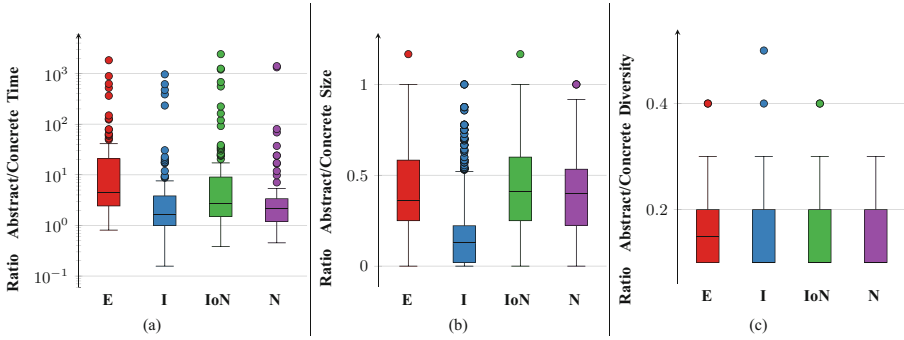


Fig. 8. Comparison of performance for different upper bounds

Figure 8(b) compares the size of the abstract instances produced by the different upper bound kinds. We again present size as a ratio of the size of the abstract instance divided by the size of the corresponding concrete instance. There are 871, 913, 901, 570 abstract-concrete pairs in the boxplot for *Exact*, *Instance*, *Instance or None* and *None* respectively. For *Exact*, *Instance or None* and *None*, the size performance is very similar. In contrast, *Instance* consistently produces smaller abstract instances than all other three upper bound kinds. While the other three produce just 30 empty abstract instances, all for the “student16” submission, *Instance* produces 210 abstract instances without lower bounds. This translates directly into the observed performance difference in size.

Figure 8(c) compares the diversity of generated abstract instances produced by the different upper bound kinds. We again present diversity as a ratio of the number of unique abstract instances divided by the number of unique concrete instances per command that do enumerate 10 concrete instances. There are 77, 86, 83 and 54 commands in the boxplot for *Exact*, *Instance*, *Instance or None* and *None* respectively. As Fig. 8(c) shows, the different upper bound kinds have very similar performance in terms of diversity. *Exact* upper bounds does produce slightly more abstract instances on average across the first 10 instance, with all other upper bounds having a median of 0.1, meaning only one unique abstract instance, while *Exact*’s median is 0.2.

### 6.5 Threats to Validity

There are two main threats to validity for our results. First, we selected our benchmark models to eliminate the likelihood of encountering trivial commands. Therefore, our results may not generalize to other Alloy models which may use different operators and signature constraints than those that appear in our evaluation models. Second, our implementation may have bugs. To mitigate this threat we have used existing components where possible, e.g., Delta Debugging [32] and Alloy’s APIs and solver (see Sect. 5.1). In addition, we have added

assertions and ran our algorithms on all available models. Before Algorithm 1, l. 4 we check whether  $\text{solve}((M \cup \text{sig}4\text{Bounds}(\mathcal{A})) \wedge \text{expr}4\text{Bounds}(\mathcal{A}) \wedge C, B)$  is satisfiable (otherwise  $\text{expr}4\text{Bounds}$  is incorrect as  $\mathcal{I}$  must be a solution). In Algorithm 2 we check that  $\text{solve}(M' \wedge \text{bounds} \wedge C, B)$  is satisfiable, i.e., that there are instances represented by the candidate.

## 7 Related Work

**Explaining Alloy Instances.** Our motivation for developing abstract instances is to help users understand why a given instance was generated by the Analyzer for an executed command. There have been two notable efforts related to helping explain Alloy instances. First, Amalgam is an extension to the Analyzer, which uses provenance chains to inform the user why a specific tuple does or does not appear in the scenario [14]. Unlike abstract instances, Amalgam’s provenance chain includes the facts of the model and thus it is possible for the provenance chain of a tuple to never reference the explicitly invoked formulas of the command. Second, recent work [7] explored how presenting novice users with a combination of instances and non-instances for a command can help the user understand a modeled constraint. This work uses tailored instances that were selected for the study and thus does not try to influence an active enumeration.

**Instance Enumeration for Alloy.** Our technique is closely related to techniques which look to enhance the Analyzer’s instance enumeration process. One traditional approach is to reduce the number of instances through symmetry breaking, where the goal is to remove isomorphic instances [11, 22]. Beyond symmetry breaking, several past projects improve instance enumeration by (1) influencing the order of instances [24, 25] and (2) trying to narrow what scenarios are generated using a specific criteria, e.g., abstract functions [26], minimality [15], maximality [33], field exhaustiveness [17], and coverage [18, 27]. All of these techniques reduce the number of instances that are generated by applying additional criteria to how any new instance generated must differ from the previous set of instances. Of these, Aluminium, which enumerates minimal instances, is the most closely related to our technique. In contrast to abstract instances, Aluminium produces complete instances, which can prevent Aluminium from further reducing the information presented as there are lower bounds enforced by the constraints of the model that Aluminium will be required to meet to ensure the instance satisfies the facts of the model, in addition to the command.

## 8 Conclusion

This paper introduces the concept of abstract instances for the Alloy modeling language. These instances serve to remove information in the instance that is not directly relevant to the executed predicate or assertion invoked by the command. Our experimental results show that abstract instances can often be produced with a small overhead but do successfully reduce the information presented to



the user. In addition, our results reveal that an abstract instances often represent multiple concrete instances. As future work, we plan to conduct a user study to evaluate how abstract instances help users understand analysis results, explore how we can efficiently enumerate unique abstract instances, and extend our approach to handle Alloy's new temporal logic extension.

## References

1. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J.C., Song, D.: Towards a formal foundation of web security. In: Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, 17–19 July 2010, pp. 290–304. IEEE Computer Society (2010). <https://doi.org/10.1109/CSF.2010.27>
2. Alloy: Alloy Tools GitHub. <https://github.com/AlloyTools> (2022). Accessed 5 2022
3. Alloy 6 Language Reference. <https://alloytools.org/spec.html> (2022). Accessed 8 2022
4. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to alloy. *Softw. Syst. Model.* **9**(1), 69–86 (2010). <https://doi.org/10.1007/s10270-008-0110-3>
5. Cunha, A., Garis, A., Riesco, D.: Translating between Alloy specifications and UML class diagrams annotated with OCL. *Softw. Syst. Model.* **14**(1), 5–25 (2013). <https://doi.org/10.1007/s10270-013-0353-5>
6. Dini, N., Yelen, C., Alrmaih, Z., Kulkarni, A., Khurshid, S.: Korat-API: a framework to enhance Korat to better support testing and reliability techniques. In: SAC (2018)
7. Dyer, T., Nelson, T., Fisler, K., Krishnamurthi, S.: Applying cognitive principles to model-finding output: the positive value of negative information. *Proc. ACM Program. Lang.* **6**(OOPSLA), 1–29 (2022). <https://doi.org/10.1145/3527323>
8. Jackson, D.: Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* **11**(2), 256–290 (2002)
9. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge (2006)
10. Jackson, D.: Alloy: a language and tool for exploring software designs. *Commun. ACM* **62**(9), 66–76 (2019). <https://doi.org/10.1145/3338843>
11. Khurshid, S., Marinov, D., Shlyakhter, I., Jackson, D.: A case for efficient solution enumeration. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 272–286. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-24605-3\\_21](https://doi.org/10.1007/978-3-540-24605-3_21)
12. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: class diagrams analysis using alloy revisited. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 592–607. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-24485-8\\_44](https://doi.org/10.1007/978-3-642-24485-8_44)
13. Marinov, D., Khurshid, S.: TestEra: a novel framework for automated testing of Java programs. In: ASE (2001)
14. Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of “why” and “why not”: enriching scenario exploration with provenance. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, 4–8 September 2017, pp. 106–116. ACM (2017). <https://doi.org/10.1145/3106237.3106272>

15. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: principled scenario exploration through minimality. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE'13, San Francisco, CA, USA, 18–26 May 2013, pp. 232–241. IEEE Computer Society (2013). <https://doi.org/10.1109/ICSE.2013.6606569>
16. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: LISA (2010)
17. Ponzio, P., Aguirre, N., Frias, M.F., Visser, W.: Field-exhaustive testing. In: FSE (2016)
18. Porncharoenwase, S., Nelson, T., Krishnamurthi, S.: CompoSAT: specification-guided coverage for model finding. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 568–587. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-95582-7\\_34](https://doi.org/10.1007/978-3-319-95582-7_34)
19. Ringert, J.O., Sullivan, A.K.: Abstract alloy instances artefact (2022). <https://doi.org/10.5281/zenodo.7339931>
20. Ringert, J.O., Sullivan, A.K.: Abstract alloy instances code (2022). <https://github.com/jringert/alloy-absinst>
21. Samimi, H., Aung, E.D., Millstein, T.: Falling back on executable specifications. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 552–576. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-14107-2\\_26](https://doi.org/10.1007/978-3-642-14107-2_26)
22. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. In: SAT (2001)
23. Sullivan, A.: Hawkeye: user-guided enumeration of scenarios. In: Jin, Z., Li, X., Xiang, J., Mariani, L., Liu, T., Yu, X., Ivaki, N. (eds.) 32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, 25–28 October 2021, pp. 569–578. IEEE (2021). <https://doi.org/10.1109/ISSRE52982.2021.00064>
24. Sullivan, A.: Hawkeye: user guided enumeration of scenarios. In: ISSRE (2021)
25. Sullivan, A., Jovanovic, A.: Reach: refining alloy scenarios by size. In: ISSRE (2022)
26. Sullivan, A., Marinov, D., Khurshid, S.: Solution enumeration abstraction: a modeling idiom to enhance a lightweight formal method. In: Ait-Ameur, Y., Qin, S. (eds.) ICFEM 2019. LNCS, vol. 11852, pp. 336–352. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-32409-4\\_21](https://doi.org/10.1007/978-3-030-32409-4_21)
27. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for alloy. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, 13–17 March 2017, pp. 264–275. IEEE Computer Society (2017). <https://doi.org/10.1109/ICST.2017.31>
28. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-71209-1\\_49](https://doi.org/10.1007/978-3-540-71209-1_49)
29. Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: the CheckMate approach. *IEEE Micro* **39**(3), 84–93 (2019)
30. Nokhbeh Zaeem, R., Khurshid, S.: Contract-based data structure repair using alloy. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 577–598. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-14107-2\\_27](https://doi.org/10.1007/978-3-642-14107-2_27)
31. Zave, P.: Reasoning about identifier spaces: how to make chord correct. *IEEE Trans. Softw. Eng.* **43**(12), 1144–1156 (2017). <https://doi.org/10.1109/TSE.2017.2655056>

32. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. *IEEE Trans. Soft. Eng.* **28**(2), 183–200 (2002). <https://doi.org/10.1109/32.988498>
33. Zhang, C., et al.: Alloymax: bringing maximum satisfaction to relational specifications. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 155–167. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (2021)