# Relational Reasoning via SMT Solving

Aboubakr Achraf El Ghazi and Mana Taghdiri

Karlsruhe Institute of Technology, Germany
{elghazi,mana.taghdiri}@kit.edu
http://asa.iti.kit.edu/

**Abstract.** This paper explores the idea of using a SAT Modulo Theories (SMT) solver for proving properties of relational specifications. The goal is to automatically establish or refute consistency of a set of constraints expressed in a first-order relational logic, namely Alloy, without limiting the analysis to a bounded scope. Existing analysis of relational constraints – as performed by the Alloy Analyzer – is based on SAT solving and thus requires finitizing the set of values that each relation can take. Our technique complements this approach by axiomatizing all relational operators in a first-order SMT logic, and taking advantage of the background theories supported by SMT solvers. Consequently, it can potentially prove that a formula is a tautology – a capability completely missing from the Alloy Analyzer – and generate a counterexample when the proof fails. We also report on our experiments of applying this technique to various systems specified in Alloy.

**Keywords:** First-order relational logic, SAT Modulo Theories, Z3, Alloy, Relational specification, Constraint solving.

## 1 Introduction

Many computational problems can be specified declaratively as a set of constraints expressed in a first-order relational logic. Safety properties of structure-rich systems, in particular, have been successfully expressed in Alloy [14], a typed, first-order relational logic with a built-in transitive closure operator. Due to its expressiveness and yet simplicity, Alloy has become a popular choice for describing high-level designs of various systems such as network configurations [24], naming architectures [17], and file-systems [15, 20]. It has also been used as an intermediate logic in many program checking tools such as Jalloy [27], JForge [7], Karun [25], and TestEra [16].

Besides its expressiveness and intuitive syntax, Alloy's fully automatic constraint solver – the Alloy Analyzer – is an important reason for its popularity. The Analyzer checks a collection of Alloy constraints, looking for an instance that satisfies all the constraints, but violates a property of interest. This analysis, however, is always performed with respect to a *bounded scope* in which only a finite number of values is considered for each type. This is because Alloy constraints are translated to a propositional logic and solved using a SAT solver. Therefore, although the Alloy Analyzer can produce counterexamples efficiently,

it can never *prove* the correctness of a property – not even for the simplest constraints. Furthermore, since arithmetic expressions in Alloy are directly translated to SAT via bit blasting, they can be analyzed with respect to only a few bits. Consequently, Alloy offers limited support for numerical constraints.

In order to overcome these limitations, we introduce a new approach in which Alloy constraints are analyzed using an SMT solver rather than a SAT solver. SMT solvers are particularly attractive because they can efficiently prove a rich combination of decidable background theories without sacrificing completeness or full automation. Furthermore, their increasing capability of handling quantifiers [5, 12, 13] supports an intuitive, non-finitized translation of first-order relational logic. Similar to SAT solvers, many SMT solvers can produce satisfying instances as well as unsatisfiable cores that improve their usability.

Our previous work [9] described how a subset of Alloy could be analyzed using the Yices SMT solver [8]. That analysis could prove properties of certain Alloy models, but it required type finitization for handling the transitive closure operator. Therefore, a complete proof was impossible in the presence of transitive closure. Our current technique, however, handles the whole Alloy language without requiring any type finitization and thus is potentially capable of proving properties of any Alloy model. Furthermore, it produces SMT formulas in the standard SMT-LIB language, so they can be analyzed by various SMT solvers.

We mitigate the bounded-analysis problem of Alloy by specifying all relational operators as first-order axioms in SMT2 – SMT-LIB, version 2.0 [4] – exploiting the increasing power of SMT solvers in handling quantifiers. However, since the Alloy logic is undecidable, axiomatizing certain Alloy constructs such as its hierarchical type system, transitive closure, set cardinality, and multiplicity keywords is particularly challenging; a naive translation can generate undecidable formulas that cannot be proven by SMT solvers. Therefore, we have carefully developed our translation rules to ensure that (1) the translation is always sound, and (2) it performs well in practice, i.e. the SMT formulas resulting from commonly-used Alloy idioms and patterns can be proven by the solver.

Due to our arbitrary use of quantifiers, our target logic is undecidable, and thus the instance returned by the SMT solver may be marked as "unknown". This indicates that the instance may be spurious, and must be double-checked. However, if the SMT solver outputs "unsat", it is guaranteed that the set of formulas is unsatisfiable. Consequently, our approach is a complement to the Alloy Analyzer: when the Alloy Analyzer fails to find a counterexample, our technique will translate the constraints to SMT2, aiming at proving the correctness of the property of interest. Therefore, the user can benefit from both the Alloy Analyzer's sound counterexamples, and the SMT solvers' sound proofs.

We report on the theoretical foundations of analyzing the Alloy relational logic using an SMT solver. We describe the translation rules in detail and report on our experiments of applying those rules to 8 systems already specified in Alloy. We checked a total of 20 assertions using the Z3 SMT solver [6] and the results are encouraging: out of the 15 valid assertions, 12 were successfully proven correct, and sound counterexamples were generated for 4 out of the 5 invalid assertions.

$problem$ ::= $typeDcl^* relDcl^* fact^* [assertion]$
$typeDcl$ ::= **sig** identifier [**in** $type$]
$relDcl$ ::= $rel$ : $type$ [[$mult$] → [$mult$] $type$]$^*$
$mult$ ::= **lone** | **some** | **one** | **set**
$fact$ ::= $formula$
$assertion$ ::= $formula$
$exp$ ::= $type$ | $var$ | $rel$ | **none** | $exp + exp$
| $exp$ & $exp$ | $exp - exp$ | $exp.exp$
| $exp \rightarrow exp$ |~$exp$ |^$exp$ | **Int** $intExp$
$intExp$ ::= number | #$exp$ | **int** $var$
| $intExp$ $intOp$ $intExp$ | (**sum** [$var$ : $exp$]$^+$ | $intExp$)

$formula$ ::= $exp$ **in** $exp$
| $intExp$ $intComp$ $intExp$
| **not** $formula$ | $formula$ **and** $formula$
| $formula$ **or** $formula$
| **all** $var$ : $exp$ | $formula$
| **some** $var$ : $exp$ | $formula$
$intOp$ ::= + | -
$intComp$ ::= < | > | =
$type$ ::= identifier | **Int**
$rel$ ::= identifier
$var$ ::= identifier

**Fig. 1.** Abstract syntax for the core Alloy logic

This suggests that although our motivation was to prove valid assertions, our technique can be useful for invalid assertions too. The analysis time in most cases was close to zero seconds, witnessing the efficiency of using SMT solvers.

## 2 Background

### 2.1 The Alloy Language

Alloy [14] is a typed, first order relational logic with an object-oriented-like syntax. As shown in Figure 1, a problem expressed in Alloy is a collection of type declarations, relation declarations, formulas marked as fact, and possibly an assertion to check. The Alloy Analyzer looks for an instance that satisfies all the facts, but violates the assertion. This analysis is performed with respect to a *finite scope*, an upper bound on the number of elements of each type, and thus absence of an instance does not constitute proof of correctness.

**Type Declarations.** Alloy types represent sets of atoms. The signature $sig\ A\{\}$ declares a top-level type named $A$ whereas $sig\ B\ in\ A\{\}$ declares a type $B$ as a subtype (subset) of the type $A$.

**Relation Declarations.** Relations are declared as fields of signatures. That is, $sig\ A\ \{r :\ B \rightarrow C\}$ declares $r$ as a relation of type $A \rightarrow B \rightarrow C$. A relation can be constrained by the multiplicity keywords *lone* (at most one), *some* (at least one), *one* (exactly one), and *set* (any number). A declaration $r : A\ m \rightarrow n\ B$ constrains $r$ to associate each element of $A$ with $n$ elements of $B$, and each element of $B$ with $m$ elements of $A$ where $m$ and $n$ are multiplicity keywords.

**Expressions.** Basic Alloy expressions are relations. Sets are unary relations, and scalars are singleton unary relations. The built-in relation *none* denotes the empty set. Set operators union, intersection, and difference are denoted by "+", "&", and "-" respectively. The "." operator denotes relational join: for two relations $p$ and $q$ with arities $m$ and $n$, the expression $p.q$ is defined as $\{(p_1, .., p_{m-1}, q_2, .., q_n) \mid (p_1, .., p_m) \in p \ \wedge\ (q_1, .., q_n) \in q \ \wedge\ p_m = q_1\}$. The expression $p \rightarrow q$ denotes Cartesian product of $p$ and $q$, and ~ represents the transpose of a binary relation, i.e. $\sim \mathtt{r} = \{(r_2, r_1) \mid (r_1, r_2) \in r\}$. The operator ^ denotes transitive closure, and is defined only on homogeneous binary relations.

Integer expressions denote primitive integers. The built-in type *Int* represents the set of all atoms carrying primitive integers. The expression *Int ie* denotes

the atom carrying the integer denoted by the integer expression $ie$, whereas $int\ v$ denotes the integer value of the atom represented by the variable $v$. Integer expressions are obtained from constant numbers and set cardinality `#`, and combined using arithmetic operators. These operators are distinguished from set operators using the type information. The expression $(sum\ x : A \mid ie)$ computes the sum of the values that the integer expression $ie$ can take for all distinct bindings of the variable $x$ in $A$.

**Formulas.** Basic Alloy formulas are formed using the subset operator $in$ and the integer comparison operators, and combined using logical operators. In a quantified formula $(Q\ x : e \mid F)$, the formula $F$ is based on $x$, the expression $e$ bounds the values of $x$, and $Q$ is either a universal or existential quantifier.

### 2.2 The SMT2 Language

We translate Alloy problems to SMT2 – the SMT-LIB standard, version 2.0 [4] as supported by the Z3 SMT solver[1] [6]. SMT-LIB supports various theories and defines a common language for SMT problems. Our generated formulas use the quantified theories of free sorts, linear integer arithmetic, and uninterpreted functions with equality, and thus fit in the AUFLIA logic [4].

**Declarations.** The logic underlying SMT2 is a many-sorted first-order logic with equality. It supports `Int, Real`, and `Bool`, and allows users to declare new sorts (types) using the `declare-sort` command.

Functions are the basic building blocks of SMT formulas. The command `(declare-fun` $f$ `(`$A_1, \cdots, A_{n-1}$`)` $A_n$`)` declares $f : A_1 \times \cdots \times A_{n-1} \rightarrow A_n$. All functions are *total*, i.e. they are defined for all elements of their domain. Constants are functions that take no arguments, i.e. a constant $v$ of type $A$ is declared as `(declare-fun` $v$ `()` $A$`)`.

**Assertions.** The command `(assert` $f$`)` asserts a formula $f$ in the current logical context. Basic formulas are function applications and can be combined using the boolean operators `and`, `or`, `not`, and `=>` (implies). Universal and existential quantifiers are denoted by `(forall (`$a_1$ $A_1$`)..(`$a_n$ $A_n$`)` $f$`)` and `(exists (`$a_1$ $A_1$`)..(`$a_n$ $A_n$`)` $f$`)` respectively.

**Analysis.** We use the `(set-logic` $l$`)` command to tell the solver what combination of theories is being used, and `(check-sat)` to instruct the solver to check whether the conjunction of the given assertions is satisfiable or not.

## 3 Approach

We translate well-typed Alloy problems to SMT2 by specifying the semantics of Alloy constructs as first-order axioms. Therefore, Alloy problems can be analyzed without type finitization or sacrificing full automation. However, due to Alloy's undecidability and our extensive use of quantifiers, the resulting SMT formulas can be undecidable. Thus the SMT solver may fail to establish or refute an

[1] The syntax of Z3 is slightly different from SMT-LIB in the use of parentheses.

```
sig Name {}
sig Address {}
sig Book {}
sig AddrBook in Book {
 addr: Name->lone Address
}
```

```
1. (declare-sort Name)
2. (declare-sort Address)
3. (declare-sort Book)
4. (declare-fun isName (Name) Bool)
5. (declare-fun isAddress (Address) Bool)
6. (declare-fun isBook (Book) Bool)
7. (declare-fun isAddrBook (Book) Bool)
8. (assert (forall (b Book) (=> (isAddrBook b) (isBook b))))
9. (declare-fun addr (Book Name Address) Bool)
10.(assert (forall (b Book)(n Name)(a Address)
  (=> (addr b n a) (and (isAddrBook b)(isName n)(isAddress t)))))
11.(declare-fun oneAddr (Book Name) Address)
12.(assert (forall (b Book)(n Name)(a Address)
  (=> (addr b n a) (= a (oneAddr b n)))))
```

**Fig. 2.** An example of translating Alloy declarations

assertion and can generate an "unknown" instance that may be invalid. We try to minimize the chances of producing invalid instances in practice by choosing an axiomatization that performs best for commonly-used Alloy patterns and idioms according to our experiments.

### 3.1 Type and Relation Declarations

Since SMT2 does not support subtype declarations, we translate Alloy's hierarchical type system implicitly. Top-level Alloy types are translated to uninterpreted SMT2 sorts, but subtypes are specified only through axioms. Extra axioms are needed for specifying those relations that are defined over subtypes.

Figure 2 provides an example. On the left, an address book is represented by an Alloy relation `addr: AddrBook -> Name -> lone Address` where `AddrBook` is a subtype of `Book`. On the right, our SMT2 translation is shown. The top-level types `Name`, `Address`, and `Book` are declared as uninterpreted sorts in Lines 1-3. Lines 4-7 declare an uninterpreted membership function for each Alloy type. A membership function `isT` is defined over the top-level, supertype $T'$ of a type $T$ to denote which elements of $T'$ belong to $T$. Membership functions are necessary for specifying the semantics of subtypes[2]. Line 8 specifies the subtype semantics, i.e. all elements of `AddrBook` should belong to `Book`.

Since all SMT2 functions are total, arbitrary relations are specified using a function with an additional boolean column whose value is true for the tuples that are included in that relation, and false for all others. Line 9 declares `addr` as a boolean-valued function over its top-level types. Line 10 constrains `addr` to be defined only for the intended type of `AddrBook` $\times$ `Name` $\times$ `Address`. The multiplicity keyword `lone` is specified by Lines 11-12. Line 11 declares an uninterpreted function `oneAddr` that maps each element of `Book` $\times$ `Name` to exactly one element of `Address`. Line 12 constrains `addr` to be a subset of `oneAddr`, and thus to map every element of `Book` $\times$ `Name` to at most one address.

Figure 3 gives the translation rules for Alloy type and relation declarations. The main translation function $D$ generates a collection of SMT commands for an Alloy paragraph. This figure defines $D$ only for Alloy declarations; facts and

[2] Membership functions of top-level types are often avoidable. They are included in this example for uniformity.

$D : AlloyPar \rightarrow SMTCommand^*$ $\quad S : Alloy\ type$

$T_i : AlloyExpr \rightarrow SMTSort$ $\quad r : Alloy\ relation$

$E : AlloyExpr \times \overrightarrow{SMTVar} \rightarrow SMTFormula$ $\quad v : SMT\ variable$

$E[S, v] = (isName[S]\ v)$

$E[r, \texttt{<}v_1, .., v_n\texttt{>}] = (name[r]\ v_1\ ..\ v_n)$

$D[\textbf{sig}\ S] = \{(\textbf{declare-sort}\ name[S]),$ $\quad$ `If` $S$ `is top-level`

$\quad (\textbf{declare-fun}\ isName[S]\ (T_1[S])\ \textbf{Bool})\}$

$D[\textbf{sig}\ S_1\ \textbf{in}\ S_2] = \{D[\textbf{sig}\ S_1],\ (\textbf{assert}\ (\textbf{forall}\ (v\ T_1[S_1])\ (\texttt{=>}\ E[S_1, v]\ E[S_2, v])))\}$

$D[r : S_1\texttt{->} ..\ \texttt{->}S_n] = \{(\textbf{declare-fun}\ name[r]\ (T_1[S_1]\ ..\ T_1[S_n])\ \textbf{Bool}),$

$\quad (\textbf{assert}\ (\textbf{forall}\ (v_1\ T_1[S_1])\ ..\ (v_n\ T_1[S_n])(\texttt{=>}\ E[r, \texttt{<}v_1, .., v_n\texttt{>}]\ (\textbf{and}\ E[S_1,\ v_1]\ ..\ E[S_n,\ v_n])))))\}$

$D[r :\ S_1\texttt{->} ..\ \texttt{->}\ \textbf{set}\ S_n] = D[r :\ S_1\texttt{->} ..\ \texttt{->}\ S_n]$

$D[r :\ S_1\texttt{->} ..\ \texttt{->}\ \textbf{lone}\ S_n] = \{D[r :\ S_1\texttt{->} ..\ \texttt{->}\ S_n],$

$\quad (\textbf{declare-fun}\ oneName[r]\ (T_1[S_1]\ ..\ T_1[S_{n-1}])\ T_1[S_n]),$

$\quad (\textbf{assert}\ (\textbf{forall}\ (v_1\ T_1[S_1])\ ..\ (v_n\ T_1[S_n])(\texttt{=>}\ E[r, \texttt{<}v_1, .., v_n\texttt{>}]\ (\texttt{=}\ v_n\ (oneName[r]\ v_1..v_{n-1})))))\}$

$D[r :\ S_1\ \texttt{->} ..\ \texttt{->}\ \textbf{some}\ S_n] = \{D[r :\ S_1\ \texttt{->} ..\ \texttt{->}\ S_n],$

$\quad (\textbf{declare-fun}\ oneName[r]\ (T_1[S_1]\ ..\ T_1[S_{n-1}])\ T_1[S_n]),$

$\quad (\textbf{assert}\ (\textbf{forall}\ (v_1\ T_1[S_1])\ ..\ (v_{n-1}\ T_1[S_{n-1}])$

$\quad\quad (\texttt{=>}\ (\textbf{and}\ E[S_1, v_1]\ ..\ E[S_{n-1}, v_{n-1}])\ E[r,\ \texttt{<}v_1, .., v_{n-1}, (oneName[r]\ v_1\ ..\ v_{n-1})\texttt{>}])))\}$

$D[r :\ S_1\texttt{->} ..\ \texttt{->}\ \textbf{one}\ S_n] = \{D[r :\ S_1\texttt{->} ..\ \texttt{->}\ \textbf{lone}\ S_n],\ D[r :\ S_1\texttt{->} ..\ \texttt{->}\ \textbf{some}\ S_n]\}$

**Fig. 3.** Translation rules for Alloy type and relation declarations

assertions are covered in Sec. 3.2. For an Alloy expression $e$ of type $S_1 \times .. \times S_n$, the auxiliary function $T_i[e]$ returns the SMT sort that corresponds to the top-level, supertype of $S_i$. Function $E$ translates intermediate Alloy expressions. $E[e, \overrightarrow{v}]$ returns an SMT formula that encodes that a list of SMT variables $\overrightarrow{v}$ is included in the relation resulting from evaluating $e$. Figure 3 defines $E$ as needed by this set of rules. Other cases are covered in the next sections. Sorts and functions declared in SMT2 are named using the functions *name*, *isName*, and *oneName*. The function *name* returns a unique name for each Alloy type and relation. *isName* denotes the type membership function, and *oneName* denotes the helper function used for encoding multiplicity constraints.

As shown in Figure 3, an Alloy top-level type is translated to a sort in SMT2. A membership function is declared for each Alloy type to represent the elements that are included in that type. Subtypes are further constrained to be subsets of their immediate supertypes. An Alloy relation is translated to a boolean-valued SMT2 function. Since only top-level types are declared as sorts, this function is declared over top-level types. An extra constraint ensures that the relation is defined only for its intended types (and not their supertypes). Multiplicity keywords can be desugared to basic Alloy constraints. For example, $r : S_1 \rightarrow \texttt{lone}\ S_2$ is equivalent to $r : S_1 \rightarrow S_2$ with the additional constraint $\texttt{all}\ x : S_1,\ y, z : S_2 \mid ((y\ \texttt{in}\ x.r)\ \texttt{and}\ (z\ \texttt{in}\ x.r)) \Rightarrow (y = z)$. However, since multiplicity applied to the last column is widely-used in Alloy, we optimize this case. For a relation $r : S_1 \rightarrow .. \rightarrow \texttt{mult}\ S_n$ with a multiplicity keyword `mult`, we declare a function $oneName[r]$ that maps every tuple of $(T_1[S_1] \times .. \times T_1[S_{n-1}])$ to exactly one element of $T_1[S_n]$. For `lone`, elements of $r$ must be included in $oneName[r]$, for `some`, $r$ must include all elements of $oneName[r]$ that belong to the intended type of $S_1 \times .. \times S_n$, and for `one`, both conditions must hold.

$F : AlloyFormula \rightarrow SMTFormula$ $\quad$ $fact, assertion, f : Alloy\ formula$
$D[fact] = (\textbf{assert}\ F[fact])$ $\quad$ $e : Alloy\ expression$
$D[assertion] = (\textbf{assert}\ F[\textbf{not}\ assertion])$ $\quad$ $x : Alloy\ variable$
$F[\textbf{not}\ f] = (\textbf{not}\ F[f])$ $\quad$ $v, w : SMT\ variable$
$F[f_1\ \textbf{and}\ f_2] = (\textbf{and}\ F[f_1]\ F[f_2])$
$F[f_1\ \textbf{or}\ f_2] = (\textbf{or}\ F[f_1]\ F[f_2])$
$F[\textbf{all}\ x : e \mid f] = (\textbf{forall}\ (v\ T_1[e])\ (\texttt{=>}\ E[e, v]\ F[f][v/x]))$
$F[\textbf{some}\ x : e \mid f] = (\textbf{exists}\ (v\ T_1[e])\ (\textbf{and}\ E[e, v]\ F[f][v/x]))$
$F[e_1\ \textbf{in}\ e_2] = (\textbf{forall}\ (v_1\ T_1[e_1])\ ..\ (v_n\ T_n[e_1])(\texttt{=>}\ E[e_1, \texttt{<}v_1, .., v_n\texttt{>}]\ E[e_2, \texttt{<}v_1, .., v_n\texttt{>}]))$
$E[\texttt{~}e, \texttt{<}v_1, v_2\texttt{>}] = E[e, \texttt{<}v_2, v_1\texttt{>}]$
$E[e_1\ \texttt{+}\ e_2, \texttt{<}v_1, .., v_n\texttt{>}] = (\textbf{or}\ E[e_1, \texttt{<}v_1, .., v_n\texttt{>}]\ E[e_2, \texttt{<}v_1, .., v_n\texttt{>}])$
$E[e_1\ \texttt{\&}\ e_2, \texttt{<}v_1, .., v_n\texttt{>}] = (\textbf{and}\ E[e_1, \texttt{<}v_1, .., v_n\texttt{>}]\ E[e_2, \texttt{<}v_1, .., v_n\texttt{>}])$
$E[e_1\ \texttt{-}\ e_2, \texttt{<}v_1, .., v_n\texttt{>}] = (\textbf{and}\ E[e_1, \texttt{<}v_1, .., v_n\texttt{>}]\ (\textbf{not}\ E[e_2, \texttt{<}v_1, .., v_n\texttt{>}]))$
$E[e_1\ \texttt{->}\ e_2, \texttt{<}v_1, .., v_n, .., v_{n+m}\texttt{>}] = (\textbf{and}\ E[e_1, \texttt{<}v_1, .., v_n\texttt{>}]\ E[e_2, \texttt{<}v_{n+1}, .., v_{n+m}\texttt{>}])$
$E[e_1.e_2, \texttt{<}v_1, .., v_{n-1}, v_{n+2}.., v_{n+m}\texttt{>}] = (\textbf{exists}\ (w\ T_n[e_1])$
$\qquad (\textbf{and}\ E[e_1, \texttt{<}v_1, .., v_{n-1}, w\texttt{>}]\ E[e_2, \texttt{<}w, v_{n+2}, .., v_{n+m}\texttt{>}]))$
$E[\textbf{none}, v] = \textbf{false}$
$E[x, v] = (\texttt{=}\ x\ v)$

**Fig. 4.** Translation rules for Alloy formulas

### 3.2 Formulas

Figure 4 gives the translation rules for Alloy facts (that are assumed to be true) and assertions (that are intended to be checked). We negate an assertion so that any instance found by the SMT solver will be a counterexample to the assertion. If the solver finds no instances, the assertion is proven correct.

In addition to the translation functions defined in Figure 3, Figure 4 uses the function $F$ to translate Alloy formulas. Negation, conjunction, and disjunction in Alloy are mapped to those in SMT2. A quantified Alloy formula ($\texttt{Q}\ x : e \mid f$) is translated to an SMT formula that bounds $x$ to $T_1[e]$, and uses either an implication (for universal quantifiers) or a conjunction (for existential quantifiers) of $e$ to constrain the values of $x$. The notation $[v/x]$ substitutes $v$ for all occurrences of $x$[3]. The Alloy formula ($e_1$ `in` $e_2$) is well-formed only when $arity[e_1] = arity[e_2]$ and is translated by specifying that every element of $e_1$ is included in $e_2$.

$E[e, \texttt{<}v_1, .., v_n\texttt{>}]$ produces an SMT formula that encodes that $\texttt{<}v_1, .., v_n\texttt{>}$ is included in the relation corresponding to $e$. Since the original Alloy constraints are well-typed, $n = arity[e]$. Defining $E$ for relational transpose, union, intersection, and difference is straightforward. An expression $e_1\texttt{->}e_2$ contains a tuple $\texttt{<}v_1, .., v_n, .., v_{n+m}\texttt{>}$ iff $e_1$ contains $\texttt{<}v_1, .., v_n\texttt{>}$ and $e_2$ contains $\texttt{<}v_{n+1}, .., v_{n+m}\texttt{>}$ where $n = arity[e_1]$ and $m = arity[e_2]$. Relational join is similar except that it requires an existentially quantified variable for the merged column of the two relations. $E[none, v] = false$ because *none* denotes the empty set, and the scalar case of $E[x, v]$ is defined as equality. Since in the expression $E[x, v]$, the variable $x$ is declared in Alloy and $v$ in SMT2, the formula $(= x\ v)$ is not well-formed by itself. However, the translation rules will substitute an SMT variable for $x$ after this formula is plugged in its enclosing formula.

### 3.3 Transitive Closure

The Alloy expression $\texttt{\^{}}r$ computes the smallest symmetric and transitive relation that contains $r$ where $r : S \rightarrow S$ is a binary homogeneous relation. Since the Alloy

[3] Alloy's universal quantifiers cannot be applied to non-unary relations, and existential quantifies over non-unary relations can be desugared using multiple unary relations.

1. (**declare-fun** $trName[r]$ (**Int** $T_1[r]$ $T_2[r]$) **Bool**)
2. (**assert** (**forall** ($i$ **Int**) ($v_1$ $T_1[r]$) ($v_2$ $T_2[r]$) (**=>** (**<** $i$ 1) (**not** ($trName[r]$ $i$ $v_1$ $v_2$)))))
3. (**assert** (**forall** ($v_1$ $T_1[r]$) ($v_2$ $T_2[r]$) (**=** ($trName[r]$ 1 $v_1$ $v_2$) $E[r, \texttt{<}v_1, v_2\texttt{>}]$)))
4. (**assert** (**forall** ($i$ **Int**) ($v_1$ $T_1[r]$) ($v_2$ $T_2[r]$) (**=>** (**>** $i$ 1)
   (**=** ($trName[r]$ $i$ $v_1$ $v_2$) (**or** ($trName[r]$ (**-** $i$ 1) $v_1$ $v_2$)
   (**exists** ($w$ $T_1[r]$) (**and** ($trName[r]$ (**-** $i$ 1) $v_1$ $w$)$E[r, \texttt{<}w, v_2\texttt{>}]$)))))))

$E[\texttt{\^{}}r,\ \texttt{<}v_1,\ v_2\texttt{>}] = (\textbf{exists}\ (i\ \textbf{Int})\ (trName[r]\ i\ v_1\ v_2))$

**Fig. 5.** Translation rules for the transitive closure of a relation $r$

Analyzer checks Alloy problems with respect to finite scopes, it soundly unrolls $\texttt{\^{}}r$ to $r+r.r+..+r^{(n)}$ where $n$ is the upper bound on the size of $S$. In our analysis, however, types are infinite and so any finite unrolling of transitive closure will be unsound. Figure 5 gives our axioms using an integer-based inductive definition. In the interest of space, here we only describe the translation of $\texttt{\^{}}r$ where $r$ is a relation explicitly declared in Alloy. The general case of $\texttt{\^{}}e$ requires normalizing the expression $e$ and applying a generalized version of these rules.

Line 1 of Figure 5 declares a helper SMT function $trName[r]$ to compute transitive closure. For any integer $i$, $(trName[r]\ i)$ denotes the expression $r + r.r + .. + r^{(i)}$. This is specified inductively (on the value of $i$) using axioms 2-4. Line 2 specifies that $(trName[r]\ i)$ does not contain any elements if $i < 1$. Line 3 constrains the base case of $(trName[r]\ 1)$ to be equal to $r$, and Line 4 specifies $(trName[r]\ i)$ in terms of $(trName[r]\ (i-1))$ for $i > 1$. Finally, the definition of $E$ specifies that a pair $\texttt{<}v_1, v_2\texttt{>}$ is included in the relation resulting from evaluating $\texttt{\^{}}r$ iff $\texttt{<}v_1, v_2\texttt{>}$ is included in $(trName[r]\ i)$ for some integer $i$.

### 3.4 Integer Expressions

Arithmetic expressions in Alloy are handled by bit blasting, using a fixed, user-defined bitwidth (usually less than 7 [1]). Overflows are truncated silently. Better handling of arithmetic expressions was needed in many applications [26]. Thus we deviate from the Alloy's behavior and translate integer expressions using the SMT2 theory of linear integer arithmetic that supports infinite integers.

Figure 6 gives the rules. Function $I$ translates an Alloy integer expression to an SMT2 expression of type integer. Alloy's built-in type $Int$ is mapped to the SMT2's built-in sort $Int$. Unlike Alloy that distinguishes between integer atoms and primitive integers, the SMT logic allows a single integer type. Comparison and arithmetic operators in Alloy are translated to those in SMT2. $E[Int\ ie, v]$ specifies that a variable $v$ corresponds to the atom carrying $ie$ iff the (integer) values of $v$ and $I[ie]$ are equal. The $int$ operator becomes the identity function.

In the interest of space, we discuss the translation of $\texttt{\#}r$ where $r$ is a unary relation explicitly declared in Alloy. The general case of $\texttt{\#}e$ requires normalizing the expression $e$ and applying a generalized version of the rules. Our approach allows the cardinality of a (possibly cyclic) relation to be arbitrarily large (but finite). To compute $\texttt{\#}r$, we define a mapping $ordName[r]$ from every element of $r$ to one distinct integer $i \geq 1$. We constrain the integers to be consecutive so that $\texttt{\#}r$ is the largest integer used in the mapping. Lines 1-4 of Figure 6 define the helper functions. Line 5 specifies that $crdName[r] \geq 0$ and if it is 0, then $r$

$I : AlloyIntExpr \rightarrow SMTExpr$ $\quad ie : Alloy\ integer\ expression$
$T_1[\textbf{Int}] = \textbf{Int}$ $\quad n : Number$
$F[ie_1\ intComp\ ie_2] = (intComp\ I[ie_1]\ I[ie_2])$
$E[\textbf{Int}\ ie, v] = (\texttt{=}\ I[ie]\ v)$
$E[\textbf{Int}, v] = true$ `if` $v$ `is of type Int,` $false$ `otherwise`
$I[n] = n$
$I[\textbf{int}\ x] = x$
$I[ie_1\ intOp\ ie_2] = (intOp\ I[ie_1]\ I[ie_2])$
$I[\texttt{\#}r] = crdName[r]$
$I[(\textbf{sum}\ x : r \mid ie)] = (sumName[r]\ 1\ crdName[r])$
1.(**declare-fun** $crdName[r]$ () **Int**)
2.(**declare-fun** $ordName[r]$ ($T_1[r]$) **Int**)
3.(**declare-fun** $invName[r]$ (**Int**) $T_1[r]$)
4.(**declare-fun** $trgName[r]$ (**Int**) **Bool**)
5.(**assert** (**and** (>= $crdName[r]$ 0) (=> (= $crdName[r]$ 0)(**forall** ($v$ $T_1[r]$) (**not** $E[r, v]$)))))
6.(**assert** (**forall** ($v$ $T_1[r]$)(=> $E[r, v]$ (**and** (<= 1 ($ordName[r]$ $v$)) (<= ($ordName[r]$ $v$) $crdName[r]$)))))
7.(**assert** (**forall** ($v$ $T_1[r]$)(=> $E[r, v]$(= $v$ ($invName[r]$ ($ordName[r]$ $v$))))))
8.(**assert** (**forall** ($i$ **Int**)(=> (**and** (<= 1 $i$)(<= $i$ $crdName[r]$))(= $i$ ($ordName[r]$ ($invName[r]$ $i$))))))
9.(**assert** (**forall** ($i$ **Int**)(=> (**and** (<= 1 $i$)(<=$i$ $crdName[r]$)) $E[r, (invName[r]i)]$)**:pat** {($trgName[r]$ $i$)}))
10.(**assert** (=> (< 0 $crdName[r]$) ($trgName[r]$ 1)))
11.(**assert** (**forall** ($i$ **Int**)(=> (**and** (<= 1 $i$)(< $i$ $crdName[r]$))($trgName[r]$(+ $i$ 1))) **:pat** {($trgName[r]$ $i$)}))
12.(**declare-fun** $sumName[r]$ (**Int Int**) **Int**)

**Fig. 6.** Translation rules for Alloy integer expressions

must be empty. Lines 6-9 specify that $invName[r]$ is the inverse of $ordName[r]$ and that there is a one-to-one correspondence between the elements of $r$ and the integers $1 \leq i \leq crdName[r]$. Thus $crdName[r] = |r|$. (The existence of $crdName[r]$ ensures that $|r|$ is finite.) Since the Z3 SMT solver instantiates universal quantifiers based on the ground terms syntactically used in the formulas, we introduce the helper function $trgName[r]$ to ensure that the numeric axioms are sufficiently instantiated. Lines 10 and 11 constrain ($trgName[r]\ i$) to be $true$ for $1 \leq i \leq |r|$ which triggers the instantiation of Axiom 9, which in turn triggers the instantiation of the other axioms.

Leino et.al. [18] introduced efficient first-order axioms for comprehensions of the form $Q\{L \leq i < H,\ T\}$ where $Q$ is a function (e.g. sum, min), $L$ and $H$ are the lower and upper bounds on the integer $i$, and $T$ is an integer term based on $i$. Alloy's *sum* expressions are computed over integer-carrying relations. Thus no integer bounds are explicitly available. However, using our cardinality axioms, we have $(sum\ x : r \mid ie) = sum\{1 \leq i \leq |r|,\ I[ie][invName[i]/x]\}$ which makes Leino's axioms and patterns directly applicable. Figure 6 declares $sumName[r]$ to compute this sum expression for the required integer bounds. Definition of $sumName[r]$ is based on Leino's axioms and is skipped in the interest of space.

### 3.5 Simplifications

The SMT formulas generated by previous rules can be substantially simplified while their semantics is preserved. Out of the 12 Alloy assertions proven successfully in our experiments (see Sec. 4), only 3 can be proven before simplification.

The simplification rules are given in Figure 7. Rule 1 simplifies the expressions involving functional relations. For any Alloy relation $r : S_1 \rightarrow .. \rightarrow \textbf{one}\ S_n$, a tuple <$v_1, .., v_n$> is included in the corresponding function $name[r]$ iff $v_n = (oneName[r]\ v_1, .., v_{n-1})$. Rule 2 simplifies cardinality for the obvious cases of empty and singleton relations. This is determined syntactically based on the

1. $(name[r]\ v_1\ ..\ v_n) = (= v_n (oneName[r]\ v_1\ ..\ v_{n-1}))$ `if r is a function`
2. $cardName[none] = 0, \quad cardName[e] = 1$ `if e is a singleton relation`
3. (**forall** $(v\ Sort[w])$ (=> (= $v$ $w$) $f$)) $= f[w/v]$
4. (**exists** $(v\ Sort[w])$ (**and** (= $v$ $w$) $f$)) $= f[w/v]$
5. (**and** (**forall** $(v\ T)$(=> $f_1$ $f_2$)) (**forall** $(w\ T)$(=> $f_2$ $f_1$))) = (**forall** $(v\ T)$(= $f_1$ $f_2$))

**Fig. 7.** Simplification rules

type information. Rules 3 and 4 eliminate quantifiers based on the semantics of scalar values. They substitute the free variable $w$ for the quantified variable $v$ used in a formula $f$. These rules are valid because $w$ represents a single value in $Sort[w]$. Rule 3 holds since in any logical context, $(\forall v : Sort[w] \mid ((v = w) \implies f)) \equiv (\forall v \in \{w\} \mid f) \equiv f[w/v]$, and Rule 4 holds because $(\exists v : Sort[w] \mid ((v = w) \wedge f)) \equiv (\exists v \in \{w\} \mid f) \equiv f[w/v]$. Rule 5 converts the equality hidden in bidirectional implications to an explicit equality. It is applied when $f_1$ and $f_2$ are syntactically identical except possibly for the names of the bound variables; no decision procedure calls are involved.

Simplification is done in multiple passes. The first pass applies Rules 1 and 2 to all formulas. Consecutive passes apply Rules 3-5 iteratively until no more rules are applicable. Since these rules strictly reduce the number of quantifiers, this process terminates.

### 3.6 Correctness

An Alloy problem is a structure $AP =< T_{top}, T_{sub}, R, F >$ where $T_{top}$, $T_{sub}$, $R$, and $F$ respectively denote the set of top-level types, subtypes, relations, and formulas[4] declared in $AP$. An Alloy instance $I_a = (U_a, v_a)$ defines a universe of atoms $U_a$ and a valuation $v_a$ that maps every type and relation of $AP$ to a set of elements and tuples derived from $U_a$, respectively. $I_a$ satisfies $AP$ iff

- Types are well-formed. That is, (1) for $t \in T_{sub}$ that is a subtype of $t' \in T_{top} \cup T_{sub}$, we have $v_a(t) \subseteq v_a(t')$, and (2) for $t, t' \in T_{top}$, we have $v_a(t) \cap v_a(t') = \emptyset$.
- Relations are well-formed. That is, for $r \in R$ of type $t_1 \rightarrow .. \rightarrow [m]\ t_n$, we have $v_a(r) \subseteq v_a(t_1) \times .. \times v_a(t_n)$ and the multiplicity constraint of $m$ holds.
- Any $f \in F$ evaluates to true under $I_a$. That is, $[\![f]\!]^{I_a} = true$ where $[\![]\!]^{I_a}$ is defined inductively on the grammar of Figure 1 (see [14]).

Similarly, an SMT2 problem is a structure $SP =< S, G, A >$ where $S$, $G$, and $A$ respectively denote the set of sorts, functions, and assertions declared in $SP$. An SMT instance $I_s = (U_s, v_s)$ defines a universe of elements $U_s$ and a valuation $v_s$ that defines the values of sorts and functions. An instance $I_s$ satisfies $SP$ iff

- Sorts are well-formed. That is, for $s, s' \in S$, we have $v_s(s) \cap v_s(s') = \emptyset$.
- Functions are well-formed. That is, for $g \in G$ of type $(s_1\ ..\ s_n\ s)$, $v_s(g)$ gives a total function from $v_s(s_1) \times .. \times v_s(s_n)$ to $v_s(s)$.
- Any $a \in A$ evaluates to true under $I_s$. That is, $[\![a]\!]^{I_s} = true$ where $[\![]\!]^{I_s}$ is defined inductively for SMT2 formulas (see [4]).

[4] We assume that the assertion is negated and conjoined with the formula $F$.

Our analysis complements that of the Alloy Analyzer by providing proof capability. Thus to show its soundness, it is sufficient to show that for any Alloy problem $AP$, if our SMT2 translation $D[AP]$ is unsatisfiable, implying that the assertion in $AP$ is a tautology, then $AP$ is unsatisfiable too. But since Alloy computes arithmetic with respect to a fixed bitwidth, mathematically valid numeric formulas (based on infinite integers) may be invalid in Alloy due to overflows[5]. Thus unsatisfiability of $D[AP]$ implies unsatisfiability of $AP$ only in the absence of integer overflows, or equivalently, the following theorem holds:

**Theorem 1.** *If an Alloy problem $AP =< T_{top}, T_{sub}, R, F >$ has a satisfying instance for which none of the arithmetic computations overflow, its translation $D[AP] =< S, G, A >$ has a satisfying instance too.*

*Proof. For any instance $I_a = (U_a, v_a)$ that satisfies $AP$, we construct an instance $I_s = (U_s, v_s)$ that satisfies $D[AP]$. Without loss of generality, we define $U_s = U_a$, and define $v_s$ as follows. For $s \in S$ corresponding to $t \in T_{top}$, define $v_s(s) = v_a(t)$. For $g \in G$, (1) if $g$ is a membership function for $t \in T_{top} \cup T_{sub}$, then $(v_s(g)[u] = true) \Leftrightarrow (u \in v_a(t))$ for all $u \in U_a$, (2) if $g$ is a boolean-valued function for $r \in R$, then $(v_s(g)[u_1, .., u_n] = true) \Leftrightarrow (\texttt{<}u_1, .., u_n\texttt{>} \in v_a(r))$, (3) if $g$ is a multiplicity function for $r \in R$, then $(v_s(g)[u_1, .., u_{n-1}] = u_n) \Rightarrow (\texttt{<}u_1, .., u_n\texttt{>} \in v_a(r))$ for multiplicities "some" and "one", and $(\texttt{<}u_1, .., u_n\texttt{>} \in v_a(r)) \Rightarrow (v_s(g)[u_1, .., u_{n-1}] = u_n)$ for "lone", (4) if $g$ corresponds to $\hat{}r$, then for $1 \leq i \leq |U_a|$, define $v_s(g)[i, u_1, u_2] = true \Leftrightarrow \texttt{<}u_1, u_2\texttt{>} \in v_a(r) \cup v_a(r).v_a(r) \cup .. \cup v_a(r)^{(i)}$. For $i > |U_a|$, define $v_s(g)[i, u_1, u_2] = v_s(g)[|U_a|, u_1, u_2]$, and (5) for cardinality-related $g$, let $v_s(g) = |v_a(r)|$ if $g$ is $crdName[r]$, $(v_s(g)[u_i] = i) \Leftrightarrow (v_a(r) = \{u_1, .., u_n\})$ if $g$ is $ordName[r]$, $(v_s(g)[i] = u_i) \Leftrightarrow (v_a(r) = \{u_1, .., u_n\})$ if $g$ is $invName[r]$, and $(v_s(g)[i] = true) \Leftrightarrow (1 \leq i \leq |v_a(r)|)$ if $g$ is $trgName[r]$. Sorts and functions are well-formed under $I_s$ because types and relations are well-formed under $I_a$. The property $[\![a]\!]^{I_s} = true$ is proved by cases: it holds for the assertions produced by each translation rule based on the semantics of Alloy and SMT2. Absence of integer overflows ensures that any arithmetic computation yields the same result in both logics. Details are skipped in the interest of space.*

## 4 Experiments

We have evaluated our technique by checking 20 assertions in 8 Alloy problems[6]: the address book of an email client where aliases and groups are allowed, the query interface and aggregation mechanism of Microsoft COM, the operations of a memory accessed by abstract addresses, a system for managing media files, the mark and sweep garbage collection algorithm, the own-grandpa puzzle, and a hand shaking protocol among spouses. The Alloy models of these problems are included in the Alloy 4 distribution, and represent various combinations of hierarchical types, nested relational joins, transitive closure, nested quantifiers,

[5] For example, $2 + 2 > 2$ does not hold in Alloy with a bitwidth of 3.

[6] available at `http://www.rz.uni-karlsruhe.de/~ kh133/alloyToSMT/`

| | | Alloy Analyzer | | Our Analysis by Z3 | |
|---|---|---|---|---|---|
| Problem | Assertion | Scope | Time (sec) | Time (sec) | Result |
| address book | delUndoesAdd | 31 | 80.91 | 0.00 | proved |
| | addIdempotent | 31 | 112.66 | 0.01 | proved |
| COM | theorem1 | 14 | 175.46 | 0.00 | proved |
| | theorem2 | 14 | 177.97 | 0.00 | proved |
| | theorem3 | 14 | 168.51 | 0.00 | proved |
| | theorem4a | 14 | 174.89 | 0.00 | proved |
| | theorem4b | 14 | 166.68 | 0.00 | proved |
| abstract memory | writeRead | 44 | 179.44 | 0.00 | proved |
| | writeIdempotent | 29 | 98.67 | 0.03 | proved |
| media assets | hidePreservesInv | 87 | 86.03 | 0.00 | proved |
| | pasteAffectsHidden | 29 | 138.34 | 0.00 | proved |
| mark sweep | soundness1 | 9 | 81.52 | 0.12 | false CE |
| | soundness2 | 8 | 28.84 | 0.11 | false CE |
| | completeness | 7 | 32.52 | 0.14 | false CE |
| nQueen | solCondition | 73 | 173.51 | 0.05 | proved |
| address book | addLocal | 3 | 0.05 | 0.10 | sound CE |
| media assets | cutPaste | 3 | 0.19 | 0.06 | sount CE |
| own grandpa | ownGrandpa | 4 | 0.01 | 0.12 | sound CE |
| nQueen | 15Queens | 15 | 4.95 | 13.53 | sound CE |
| handshake | puzzle | 10 | 2.47 | time out | N/A |

**Table 1.** Evaluation results

set cardinality, and arithmetic operations[7]. To further check our arithmetic rules, we also translated the queens' arrangement puzzle for an $n \times n$ chessboard [1].

We applied our translation and simplification rules to these models and used Z3 2.16 to solve the resulting SMT formulas. Table 1 gives the results. It also reports on the performance of the Alloy Analyzer 4 (AA). The time (in second) is measured on an Intel Core2Quad, 2.8GHz, 8GB memory. The Alloy analysis time is the total of the time spent on generating CNF and solving it using the SAT4J solver. The Z3 analysis time is what it reports using the *-st* option.

The assertions in the top part of the table are expected to be valid, i.e. their Alloy models contain developers' comments that no counterexamples are expected. The scope column in this case denotes the maximum scope for which AA can check the assertion before reaching the time-out of 180 seconds. The result column gives the outcome of running Z3: *proved* if it returns "unsat" when looking for a counterexample, implying that the assertion is successfully proven, and *false CE* if it returns a spurious counterexample. Out of the 15 valid assertions, 12 were proven correct by our analysis. However, none of the assertions of *mark sweep* could be proven. As the scope column suggests, this problem is structurally more complex than the other problems; AA cannot check those assertions even for a scope of 10 before reaching time-out. This problem is particularly difficult because it simulates the recursion involved in the mark and sweep algorithm by applying transitive closure to union and join of multiple relations. These expressions occur within nested quantifiers or in both sides of the subset

[7] Currently we do not support models that use Alloy's utility library.

or intersection operators. Since such structures create deeply-nested quantifiers in our translation, Z3 cannot readily prove those assertions. We are investigating other translation possibilities to reduce the complexity of such cases.

The assertions in the bottom part of the table are invalid, i.e. AA generates sound counterexamples for them. The scope column in this case gives the smallest scope required by AA to find a counterexample. Although the main goal of our approach is to prove valid assertions, we analyzed these invalid assertions to evaluate our technique in case of a counterexample. For the first 4 assertions, Z3 is capable of producing an instance that although marked as "unknown", it demonstrates a true counterexample (denoted by *sound CE*).

Since our approach requires no type finitization, its performance is always independent of scope. Exceptions are the *15Queens* and *puzzle* assertions that hard-code the scope using set cardinality. We have chosen our translation rules so that the generated SMT formulas are easy to solve, witnessed by the fact that the Z3 analysis time in most cases is close to zero. However, when producing a satisfying instance for formulas containing cardinality, Z3 has to deeply instantiate all the cardinality helper functions. Therefore, its runtime for $15Queen$ is worse than AA, and it times out for *puzzle*. This is not necessarily true for provable assertions as witnessed by *solCondition* which also involves cardinality constraints. Since AA performs well in finding small counterexamples, we suggest that the user checks his intended assertion using AA first, and then runs our analysis to prove potentially valid assertions.

## 5 Related Work

Previous attempts to prove Alloy properties used interactive theorem provers. Dynamite [11] proves properties of Alloy specifications using the PVS theorem prover [21], via a translation to fork algebra. It introduces a PVS pretty-printer that shows proof steps in Alloy, reducing the burden of guiding the prover. Prioni [3] integrates the Alloy Analyzer with the Athena theorem prover. To overcome the challenge of finding proofs, Prioni provides a lemma library that captures commonly-used Alloy patterns.

Compared to theorem provers that perform a complete analysis but are not fully automatic, SMT solvers are fully automatic, but may fail to prove quantified formulas. Recent SMT solvers, however, have shown significant advances in handling quantifiers. Z3 integrates the superposition calculus in the DPLL framework [5, 13], and CVC3 uses improved E-matching instantiation strategies [12]. SMT solvers have been used to increase the automation level of many theorem provers. The PVS [21] and Isabelle/HOL [10] logics, e.g., have been translated to Yices input language [8]. Although such translations address higher-order logics with a rich combination of types predicates, recursive data types, records, etc., they do not support constructs such as transitive closure and set cardinality.

Abadi, et. al. [2] verified some Alloy problems while identifying decidable fragments of many-sorted first-order logic. However, they only support a restricted form of transitive closure, and no integer arithmetic or cardinality. Lev-Ami,

et. al. [19] introduced a method for simulating reachability properties that arise in program verification. Similar to our technique, they specify the semantics of transitive closure using first-order axioms. However, they use additional (coloring) axioms to aid the underlying prover (SPASS [22]). The coloring axioms are either provided by users or generated by heuristics. Although not immediately clear, a similar approach may be applicable to translating Alloy's transitive closure. Automated theorem provers (ATP) such as SPASS provide an unbounded analysis based on superposition calculus, but their lack of support for linear arithmetic makes them less attractive for reasoning about a rich logic like Alloy.

Suter, et. al. [23] presented a decision procedure for the quantifier-free Boolean Algebra with Presburger Arithmetic (QFBAPA) capable of handling sets and their cardinalities. They reduce QFBAPA to integer linear arithmetic (QFPA) which is solved by the decision procedures of Z3. Set cardinality is computed using the integers that represent the cardinality of Venn regions – the regions built by the maximal overlapping degree of a finite collection of sets. Since Alloy cardinality can be applied to arbitrary expressions (possibly containing variables) with arbitrary arities, this technique is not readily applicable to our translation.

## 6 Conclusions

We presented a new approach for analyzing problems expressed in Alloy, a first-order relational logic. Its main advantage is the ability to prove an assertion correct, a capability totally missing from the Alloy Analyzer (AA). We suggest our analysis be used to complement AA: when AA fails to find a counterexample, our tool can be used to prove the assertion correct. We avoid type finitization altogether and use the theories supported by SMT solvers instead.

Due to Alloy's undecidability and our arbitrary use of quantifiers, resulting SMT formulas can be undecidable. However, among different ways of axiomatizing an Alloy construct, we have carefully chosen the one that performs best in practice. While more experiments on larger Alloy models are needed to fully evaluate our technique, current results show that Z3 can correctly handle most of the valid and invalid properties, witnessing the effectiveness of the approach. Improving the cases that Z3 failed to handle is left for future work.

Although we focused on Alloy, our translation rules demonstrate a general approach that can be applied in various contexts. In particular, we described how to specify multiplicity constraints using uninterpreted functions, transitive closure using the theory of linear integer arithmetic, and cardinality of (possibly cyclic) relations using bijective integer functions.

AA provides some predefined library functions (e.g. ordering) that trigger special optimizations in AA. Investigating an efficient translation of widely-used Alloy libraries (e.g. ordering, graph, and relation) is left for future work. We will also investigate how to use SMT solvers' unsatisfiable cores and next satisfying solution to improve the usability of our technique. Our current translation deviates from Alloy semantics in handling arithmetic using infinite integers. While

we believe that this is more suitable for most system descriptions, we will also provide an alternative fixed bitwidth arithmetic using bit-vectors in the future.

## References


1. The Alloy community. http://alloy.mit.edu/community/.
2. A. Abadi, A. Rabinovich, and M. Sagiv. Decidable fragments of many-sorted logic. *Preprint submitted to Elsevier*, 2009.
3. K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating model checking and theorem proving for relational reasoning. *RELMICS*, pages 21–33, 2003.
4. The satisfiability modulo theories library. http://goedel.cs.uiowa.edu/smtib.
5. M. P. Bonacina, C. Lynch, and L. Moura. On deciding satisfiability by DPLL and unsound theorem proving. In *CADE*, pages 35–50, 2009.
6. L. de Moura and N. Bjorner. Z3 efficient SMT solver. In *TACAS'08*, pages 337–340.
7. G. Dennis, F. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA*, pages 109–120, 2006.
8. B. Dutertre and L. de Moura. The Yices SMT solver. *tool document*, 2006.
9. A. A. El Ghazi and M. Taghdiri. Analyzing Alloy constraints using an SMT solver: A case study. In *AFM*, Edinburgh, United Kingdom, 2010.
10. L. Erkök and J. Matthews. Using Yices as an automated solver in Isabelle/HOL. In *AFM*, 2008.
11. M. F. Frias, C. G. L. Pombo, and M. M. Moscato. Alloy Analyzer+PVS in the analysis and verification of alloy specifications. In *TACAS*, pages 587–601, 2007.
12. Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *AMAI*, 55(1):101–122, Feb. 2009.
13. Y. Ge and L. Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV*, pages 306–320, 2009.
14. D. Jackson. *Software Abstractions: Logic, Lang. and Analysis*. MIT Press, 2006.
15. E. Kang and D. Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In *ABZ*, 2008.
16. S. Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, MIT, 2003.
17. S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *ASE*, 2000.
18. R. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In *SAC*, pages 615–622, 2009.
19. T. Lev-ami, N. Immerman, T. Reps, M. Sagiv, and et al. Simulating reachability using first-order logic. *IN CADE-20*, pages 99–115, 2005.
20. T. Nolte. Exploring filesystem synchronization with lightweight modeling and analysis. Master's thesis, MIT, 2002.
21. S. Owre, N. Shankar, and J. Rushby. PVS: A prototype verification system. In *CADE-11*, 1992.
22. Spass: Automated prover for FOL with equality. http://www.spass-prover.org/.
23. P. Suter, R. Steiger, and V. Kuncak. Sets with cardinality constraints in satisfiability modulo theories. In *VMCAI*, 2011.
24. M. Taghdiri and D. Jackson. A lightweight formal analysis of a multicast key management scheme. In *FORTE*, pages 240–256, 2003.
25. M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *JASE*, 14(1):87–121, 2007.
26. E. Torlak. *A Constraint Solver for Software Engineering*. PhD thesis, MIT, 2009.
27. M. Vaziri. *Finding Bugs in Software with Constraint Solver*. PhD thesis, 2004.