A Proof Assistant for Alloy Specifications

Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, Mana Taghdiri

Karlsruhe Institute of Technology, Germany
{mulbrich,geilmann,elghazi,taghdiri}@ira.uka.de

Abstract. Alloy is a specification language based on a relational first-order logic with built-in operators for transitive closure, set cardinality, and integer arithmetic. The Alloy Analyzer checks Alloy specifications automatically with respect to bounded domains. Thus, while suitable for finding counterexamples, it cannot, in general, provide correctness proofs. This paper presents Kelloy, a tool for verifying Alloy specifications with respect to potentially infinite domains. It describes an automatic translation of the full Alloy language to the first-order logic of the KeY theorem prover, and an Alloy-specific extension to KeY's calculus. It discusses correctness and completeness conditions of the translation, and reports on our automatic and interactive experiments.

1 Introduction

Due to their expressive logics, theorem provers have been successfully used to prove detailed properties of complex system specifications. However, they are often considered too expensive to use frequently during software development. The cost is twofold: (1) the proof process is often interactive and requires the user to be an expert in both the problem domain being analyzed and the theorem prover being used, and (2) the input language is often a low-level logic that makes specifications unintuitive and error-prone.

Lightweight formal methods [17], on the other hand, promote checking software partially, yet frequently, during design and implementation. Alloy [16], for example, has been successfully used for checking several software systems against their requirements (see [7] for some examples). The main reasons for Alloy's popularity are its concise language, simple semantics, and fully automatic analyzer. Alloy provides a first-order logic based on relations that is augmented with built-in transitive closure (reachability), set cardinality, and basic integer arithmetic operators, which makes it suitable for specifying structure-rich systems. Alloy specifications are automatically analyzed using a SAT solver by requiring a bound on the number of elements of each relation. Consequently, while the counterexamples are non-spurious, lack of a counterexample, in general, does not constitute a proof of correctness. Thus, for safety-critical systems, the user must perform a second round of analysis in which he specifies the problem again, in the input language of a theorem prover for full verification.

This paper introduces Kelloy, an engine for verifying Alloy specifications, with the goal of bridging the gap between lightweight formal methods and theorem provers. To reduce the cost of using the underlying theorem prover, namely

KeY [4], it (1) provides a fully automatic translation of Alloy to KFOL—the first-order logic of KeY, (2) defines an Alloy-specific extension to KeY's calculus and a reasoning strategy that improves KeY's capability in finding proofs automatically, and (3) simplifies user interaction by generating intermediate proof obligations that are easy to understand.

KeY is attractive because it combines an interactive proof assistant with an automatic engine, and its calculus strategy is extensible: one can easily add new calculus rules and assign them to different proof search heuristic strategies. Kelloy translates Alloy's operators to function symbols, thus generating formulas that conform to the high-level structure of the analyzed Alloy specification. This makes the tool easy to interact with.

Our target logic is first-order because it is semi-decidable which indicates a higher automation potential in practice than higher order logics. KeY has built-in integer support and provides a set of rules implementing Peano arithmetic [5] extended to integers. Therefore, unlike previous approaches [2,13,14], we are able to translate the entire Alloy language including integer expressions, cardinality, and the ordering module (see Section 4.5). Such a translation cannot be complete because integers are not FOL-axiomatizable, but it suffices in almost all practical cases [10, p.153]. Although we target KFOL, our resulting formulas (with minor modifications) can be verified by any prover for first-order logic, which supports integers.

While some Alloy specifications can be verified automatically, specifications that extensively use quantifiers or transitive closure result in proof obligations that are too difficult to discharge fully automatically. In a user-guided, semi-automatic proof system such as KeY, however, it often suffices if the user provides only a few inputs (e.g. in the form of quantifier instantiations or induction hypotheses) to help the system find a proof. Since trying to prove an invalid assertion is particularly costly, we suggest Kelloy be used after the assertion has been checked by the Alloy Analyzer. The user can increase the analysis bounds to gain more confidence in the correctness of the assertion before using Kelloy for full verification.

In this paper, we describe a fully automatic translation of Alloy to KFOL, and establish that, the translation is correct (for finite and infinite domains) and complete (for finite domains). We also describe an Alloy-specific extension to KeY's calculus and reasoning strategy. We evaluate the approach by conducting both automatic and interactive proofs. Out of a total of 22 proved Alloy assertions in 10 specifications, 12 were proved without any user interaction. We also evaluate the impact of the user's experience on the interactive proof process.

2 Related Work

Several approaches address the verification of Alloy specifications. Prioni [2] is the closest to ours: it translates Alloy to a first-order logic in which function symbols represent Alloy operators. Prioni's theorem prover, Athena [3], has a polymorphic type system that allows a more succinct representation of operators,

but cannot handle infinite sets. Therefore, unlike Kelloy that verifies assertions in both finite and infinite domains, Prioni only analyzes finite domains.

Another approach [12] to verifying Alloy specifications is via a translation to omega closure fork algebras [11]. Since the target system is an equational calculus, the translation eliminates all Alloy quantifiers, leading to intermediate expressions that are extremely hard to understand [14]. To reduce the cost of user interaction, Dynamite [14] has been developed, which targets a calculus in fork algebra that supports quantifiers. Unlike Kelloy that uses a first-order logic, Dynamite uses the higher-order logic of the PVS theorem prover [20].

To our knowledge, unlike Kelloy, Prioni and Dynamite do not support the complete Alloy language; they cannot handle integers and set cardinality. However, they integrate the Alloy Analyzer in their interactive proof processes by checking user-provided intermediate hypotheses using the Analyzer first. A similar feature can be added to Kelloy as well.

In [8, 9], SMT solvers are used to verify Alloy specifications fully automatically. However, since Alloy is undecidable, in many cases, SMT solvers fail to verify valid specifications. As described in [7], these approaches are complementary to a full verification, but semi-automatic engine like Kelloy.

In [18], proof obligations for Event-B [1]—a set-theoretical language supporting integer expressions, cardinalities, and binary relations—are translated to KFOL. Similar to Kelloy, this approach targets a first-order theory that resembles the constructs of the source language. This work, however, targets an untyped logic, provides no calculus rules, no tool support, and no discussion of soundness and completeness of the translation. Furthermore, relations of higher arities and the transitive closure operator are not supported by Event-B and thus not covered by this work.

3 Background

3.1 Alloy and the Alloy Analyzer

Alloy [16] is a first-order relational logic with built-in transitive closure, set cardinality, and integer arithmetic operators. Our analysis introduces function symbols for the Alloy constructs of Fig. 1. In addition to the core Alloy logic, this subset of Alloy—called Alloy0—contains commonly-used Alloy constructs. It therefore enables us to generate formulas that closely conform to the structure of the Alloy specification being analyzed, and thus simplifies user interactions. Alloy constructs not present in Alloy0 are desugared by Kelloy.

As shown in Fig. 1, an Alloy0 problem consists of declarations and an assertion to check[1]. The Alloy Analyzer checks assertions with respect to a user-provided, *finite scope*—an upper bound on the number of elements of each type—fully automatically. While the reported counterexamples are guaranteed to be non-spurious, absence of a counterexample does not constitute proof.

[1] An Alloy problem with facts $f_1, \ldots, f_n$ and an assertion a is an Alloy0 problem with an assertion (f_1 `and` ... `and` f_n) `implies` a.

problem ::= *dcl** *assertion*
dcl ::= **sig** id [(**in** | **extends**) *type*]
 | *rel* : *type* [→ *type*]$^+$
assertion ::= *formula*
exp ::= *type* | *var* | *rel* | **none** | *exp* `+` *exp*
 | *exp* `-` *exp* | *exp* `&` *exp* | *exp*.*exp*
 | *exp* → *exp* | `~`*exp* | `^`*exp* | **Int** *intExp*
intExp ::= number | `#`*exp* | **int** *var*
 | *intExp* (`+` | `-` | `*`) *intExp*
 | `(`**sum** *var* : *exp* | *intExp*`)`

formula ::= *intExp intComp intExp*
 | *exp* (**in** | `=`) *exp* | **not** *formula*
 | *formula* (**and** | **or** | **implies**) *formula*
 | (**lone** | **some** | **one**) *exp*
 | **all** *var* : **set** *exp* | *formula*
 | **some** *var* : **set** *exp* | *formula*
intComp ::= `<` | `>` | `=`
type ::= id | **Int**
rel ::= id
var ::= id

Fig. 1. Abstract syntax for Alloy0

Declarations. Alloy0 types represent sets of uninterpreted atoms. The signature `sig A` declares `A` as a top-level type. `sig B in A` declares `B` as a subtype (subset) of `A`. The `extends` keyword has the same effect with the additional constraint that extensions of a type are mutually disjoint. Relations can have arbitrary arities and are declared as $f : A_1 \to \ldots \to A_n$.

Expressions. Alloy0 expressions evaluate to relations. Sets are unary relations and scalars are singleton unary relations. The built-in relation `none` denotes the empty set. Operators `+`, `-`, and `&` denote union, difference, and intersection, respectively. For relations `r` and `s`, relational join (composition), Cartesian product, and transpose are denoted by `r.s`, `r -> s`, and `~r`, respectively. Transitive closure `^r` denotes the smallest transitive relation that contains `r`.

Integer expressions evaluate to integer values ($\mathbb{Z}$) and are constructed from numbers, arithmetic operators `+`, `-`, and `*`, set cardinality `#`, and `sum`. The built-in signature `Int` denotes the set of integer atoms. The cast operators `int` and `Int` give the integer value corresponding to an integer atom, and vice versa. The expression `(sum x: A | ie)` gives the sum of the integer expression `ie` for all distinct bindings of the variable `x` in the unary relation `A`.

Formulas. Basic Alloy0 formulas are constructed using the subset (`in`), equality, and integer comparison operators, and combined using the usual logical operators. The formulas `lone e`, `some e`, and `one e` constrain the cardinality of a relational expression `e` to be at most one, at least one, and exactly one, respectively. The quantifiers `all` and `some` denote the universal and existential quantifiers, and are supported by the Alloy Analyzer if the quantified variable is either a scalar or can be skolemized [16]. In our analysis, however, relations are first-order constructs, and thus quantifiers in the general form of $Q\ x : \texttt{set}\ e \mid F$ are allowed, where e is an expression of an arbitrary arity, x ranges over all subsets of e, and Q is either the universal or existential quantifier.

3.2 The KeY Proof System

KeY [4] is a deductive theorem prover based on a sequent calculus for JavaDL—a first-order dynamic logic for the Java programming language—which allows

both automatic and interactive proofs. Kelloy uses the many-sorted, first-order subset of JavaDL, called KFOL.

Declarations. KFOL declarations consist of a set of types, a set of function symbols, and a set of predicate symbols. We write $f : T_1 \times \ldots T_n \rightarrow T$ to declare an n-ary function symbol f that takes arguments of types $T_1, \ldots, T_n$, and returns a value of type T. A constant symbol c of type T is a function with no arguments, and is denoted by $c : T$. A predicate symbol p that takes arguments of types $T_1, \ldots, T_n$ is denoted by $p\colon T_1 \times \ldots \times T_n \rightarrow \texttt{Prop}$.

Expressions and Formulas. The set of all expressions for a declaration is denoted by `Expr`, and the set of all formulas by `Frml`. Expressions are constructed from function applications, and formulas from predicate applications. The equality predicate $=$ is built-in. In addition, KFOL provides boolean constants **true** and **false**, and the propositional connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, and the negation $\neg$ to combine formulas. Universal and existential quantificatiers are written, respectively, as $\forall x\colon T \mid \phi$ and $\exists x\colon T \mid \phi$ for a variable symbol x of type T and a formula ϕ. KFOL has a built-in type *int* along with the binary function symbols $+, -, * : \mathit{int} \times \mathit{int} \rightarrow \mathit{int}$, and the predicate symbol $< : \mathit{int} \times \mathit{int} \rightarrow \texttt{Prop}$ which are all written in infix notation.

For a KFOL declaration C, a set of formulas A and a formula f we write $A \models_C f$ to denote that f is a logical consequence of A in C.

4 Axiomatization of Alloy0

As shown in Fig. 2, Alloy0 specifications are translated to KFOL using the translation function T which takes two arguments: a well-typed Alloy0 problem P, and a set *fin* of signatures that are marked by the user to be considered as finite. This is because some specifications require a finite setting (see Section 4.3). The translation T[<p, *fin*>] returns a ternary tuple <C, k_d, k_a> where C denotes a set of KFOL constant declarations that represents the signatures and relations of P, k_d is a KFOL formula that encodes the declaration constraints and the finite types (using the finite_1 predicate described in Section 4.3), and k_a denotes a KFOL formula encoding the assertion. Instead of reducing Alloy0 constructs to their definitions, our translation uses function symbols. This increases the automation level, makes the formulas easy-to-understand, and clarifies their correspondence to the original Alloy0 formulas. The semantics of these KFOL functions are defined by a set of KFOL axioms Ax. To prove the intended assertion in P, we invoke KeY on the proof obligation for $Ax \models_C k_d \Rightarrow k_a$.

4.1 Declarations

Let max denote the maximum relation arity used in the analyzed Alloy0 problem. For every $1 \leq n \leq max$, we introduce a KFOL type Rel_n to denote the set of n-ary relations. Atoms of the universe are denoted by the KFOL type $Atom$.

For every arity n, the membership predicate $in_n\colon Atom^n \times Rel_n \rightarrow \texttt{Prop}$ allows the construction of the KFOL formula $in_n(a_1, \ldots, a_n, r)$ which denotes

$T : \mathit{problem} \times \mathcal{P}(\mathit{type}) \to \mathcal{P}(\text{KFOL-decl}) \times \texttt{Frml} \times \texttt{Frml}$
$S : \mathit{dcl} \to \text{KFOL-decl}$
$F : \mathit{dcl} \cup \mathit{formula} \to \texttt{Frml}$
$N : \mathit{type} \cup \mathit{rel} \cup \mathit{var} \to \text{KFOL-const} \cup \text{KFOL-var}$
$Ax : \mathcal{P}(\texttt{Frml})$

$T[\texttt{<}d_1 \ldots d_n, a\texttt{,}\ \mathit{fin}\texttt{>}] = \texttt{<}\bigcup_{i=1}^{i=n}\{S[d_i]\}\texttt{,}\ \bigwedge_{i=1}^{i=n} F[d_i] \wedge \bigwedge_{S \in \mathit{fin}} \mathit{finite}_1(N[S])\texttt{,}\ F[a]\texttt{>}$
$S[\texttt{sig}\ A] = N[A] : \mathit{Rel}_1$
$S[\texttt{sig}\ A\ (\texttt{in}|\texttt{extends})\ B] = N[A] : \mathit{Rel}_1$
$S[r\texttt{:}A_1 \to \ldots \to A_n] = N[r] : \mathit{Rel}_n$
$F[\texttt{sig}\ A] = \bigwedge_S \mathit{disj}_1(N[A], N[S])$ for any top-level signature $S \neq A$
$F[\texttt{sig}\ A\ \texttt{in}\ B] = \mathit{subset}_1(N[A], N[B])$
$F[\texttt{sig}\ A\ \texttt{extends}\ B] = F[\texttt{sig}\ A\ \texttt{in}\ B] \wedge$
$\bigwedge_S \mathit{disj}_1(N[A], N[S])$ for any extension S of B where $S \neq A$
$F[r\texttt{:}A_1 \to .. \to A_n] = \mathit{subset}_n(N[r], \mathit{prod}_{1\times(n-1)}(N[A_1], (.., \mathit{prod}_{1\times1}(N[A_{n-1}], N[A_n]))))$

KFOL Axioms:
$\forall r, s\text{:}\ \mathit{Rel}_n \mid \mathit{subset}_n(r, s) \Leftrightarrow \forall a_{1:n}\text{:}\ \mathit{Atom} \mid \mathit{in}_n(a_{1:n}, r) \Rightarrow \mathit{in}_n(a_{1:n}, s)$
$\forall r, s\text{:}\ \mathit{Rel}_n \mid \mathit{disj}_n(r, s) \Leftrightarrow \forall a_{1:n}\text{:}\ \mathit{Atom} \mid \neg(\mathit{in}_n(a_{1:n}, r) \wedge \mathit{in}_n(a_{1:n}, s))$

Fig. 2. Translation rules for Alloy0 declarations. $\mathcal{P}(S)$ denotes the powerset of a set S.

the membership of an n-ary tuple $\texttt{<}a_1, \ldots, a_n\texttt{>}$ in an n-ary relation r. We write $a_{i:j}$ as a shorthand for $a_i, \ldots, a_j$. The uninterpreted function $\mathit{sin}_1 : \mathit{Atom} \to \mathit{Rel}_1$ relates atoms and singleton sets.

In Fig. 2, the auxiliary translation function S translates any Alloy0 signature or relation r of arity n to a constant of type Rel_n with a unique name $N[r]$. The auxiliary translation function F constrains subtypes to be subsets of their parents using the uninterpreted predicate $\mathit{subset}_n\text{:}\ \mathit{Rel}_n \times \mathit{Rel}_n \to \texttt{Prop}$. Top-level signatures as well as extensions of a common signature are constrained to be mutually disjoint using the uninterpreted predicate $\mathit{disj}_n\text{:}\ \mathit{Rel}_n \times \mathit{Rel}_n \to \texttt{Prop}$. The semantics of these predicates are given by the axioms of Fig. 2. The type information of a relation r is encoded by constraining $N[r]$ to be a subset of the Cartesian product of its column types. Function $\mathit{prod}_{n\times m} : \mathit{Rel}_n \times \mathit{Rel}_m \to \mathit{Rel}_{n+m}$ denotes the Cartesian product of two relations of arities n and m, and is defined in Fig. 4.

Fig. 3 provides an example of translating Alloy0 declarations to KFOL. Fig. 3(a) gives a simple representation of a rooted, weighted, directed graph in Alloy0 and Fig. 3(b) gives its KFOL translation. The line numbers denote which Alloy0 statement produces each KFOL formula.

It should be noted that our type system is less precise than that of Alloy0; we encode some type-related properties as additional formulas that are incorporated as assumptions. Using the same type system would require a vast number of operators to be defined for all types, and a completely untyped system would not be compatible with our distinction of atoms and relations. Our calculus represents a useful compromise where arity information is captured syntactically by types, but the signature hierarchy is enforced semantically by formulas.

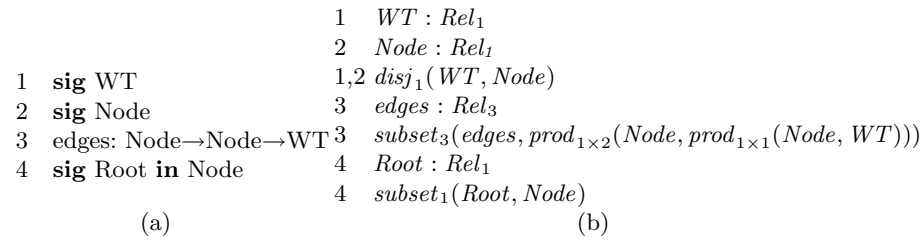

Fig. 3. An example of translating declarations: (a) Alloy0, (b) KFOL

4.2 Relational Expressions

We use the auxiliary translation function $E : exp \cup intExp \rightarrow \texttt{Expr}$ to translate Alloy expressions. A basic expression, namely a type, relation, or variable t is translated to its KFOL counterpart $N[t]$. The translation of other relational expressions is given in Fig. 4. Integer expressions are discussed in Section 4.3.

The Alloy0 relation `none` is translated to a KFOL constant $none_1 : Rel_1$ and is axiomatized to be empty. Relational operators are translated to KFOL functions whose names are subscripted by the arity information of their arguments. The semantics of these functions are defined by axioms over the predicates in_n.

Most axioms of Fig. 4 directly define the Alloy0 semantics of the corresponding operators. Due to the compactness of FOL, however, transitive closure cannot be characterized by a recursively enumerable set of first-order axioms [19]. Such an axiomatization is possible for finite interpretations [6], but because we are interested in infinite systems as well, those results are not applicable to our approach.

We define transitive closure using a primitive recursive function $itrJoin_2$ that uses the built-in integer type of KeY. This translation is comprehensible for users and allows us to define canonical induction calculus rules. As shown in Fig. 4, for a binary relation r and any integer $i \geq 0$, the KFOL expression $itrJoin_2(r, i)$ evaluates to a relation that contains the pairs (a, b) where b is reachable from a by following 0 to i steps in r.

4.3 Integer Expressions and Cardinality

The Alloy Analyzer calculates arithmetic expressions with respect to a fixed bitwidth, and thus calculations are subject to overflow. When verifying specifications, however, overflow is often not intended and integers are assumed to represent the infinite set of mathematical integers. Therefore, we translate Alloy0 integer expressions using KFOL's *int* type that models the semantics of mathematical integers, thus deliberately deviating from the Alloy semantics. Integer numbers, arithmetic expressions, and comparisons in Alloy0 are translated to their counterparts in KFOL.

The Alloy Analyzer requires all relations to be finite, and thus the cardinality operator is defined for all expressions. In our translation, however, relations

$E : exp \cup intExp \to \texttt{Expr}$

$E[\texttt{none}] = none_1$ $\qquad$ $E[\texttt{\~}x_2] = transpose_2(E[x_2])$

$E[x_n \texttt{ + } y_n] = union_n(E[x_n], E[y_n])$ $\qquad$ $E[\texttt{\^}x_2] = tc_2(E[x_2])$

$E[x_n \texttt{ - } y_n] = diff_n(E[x_n], E[y_n])$ $\qquad$ $E[\texttt{Int } ie] = sin_1(i2a(E[ie]))$

$E[x_n \texttt{ \& } y_n] = intersect_n(E[x_n], E[y_n])$ $\qquad$ $E[\texttt{int } v] = a2i(ordInv_1(E[v], 1))$

$E[x_n \texttt{ . } y_m] = join_{n\times m}(E[x_n], E[y_m])$ $\qquad$ $E[i_1 \stackrel{*}{+} i_2] = E[i_1] \stackrel{*}{+} E[i_2]$

$E[x_n \texttt{ -> } y_m] = prod_{n\times m}(E[x_n], E[y_m])$ $\qquad$ $E[\texttt{\#}x_n] = card_n(E[x_n])$

$E[(\texttt{sum } v : x_1 \mid ie)] = \Sigma_{i=1}^{i=card_1(E[x_1])} E[ie][sin_1(ordInv_1(E[x_1], i))/N[v]]$

`KFOL Axioms:`

$\forall a\colon Atom \mid in_1(a, none_1) \Leftrightarrow \textbf{false}$

$\forall a, b\colon Atom \mid in_1(b, sin_1(a)) \Leftrightarrow a = b$

$\forall r, s\colon Rel_n, a_{1:n}\colon Atom \mid in_n(a_{1:n}, union_n(r, s)) \Leftrightarrow in_n(a_{1:n}, r) \vee in_n(a_{1:n}, s)$

$\forall r, s\colon Rel_n, a_{1:n}\colon Atom \mid in_n(a_{1:n}, diff_n(r, s)) \Leftrightarrow in_n(a_{1:n}, r) \wedge \neg in_n(a_{1:n}, s)$

$\forall r, s\colon Rel_n, a_{1:n}\colon Atom \mid in_n(a_{1:n}, intersect_n(r, s)) \Leftrightarrow in_n(a_{1:n}, r) \wedge in_n(a_{1:n}, s)$

$\forall r\colon Rel_n, s\colon Rel_m, a_{1:n+m-2}\colon Atom \mid in_{n+m-2}(a_{1:n+m-2}, join_{n\times m}(r, s))$
$\Leftrightarrow (\exists b\colon Atom \mid in_n(a_{1:n-1}, b, r) \wedge in_m(b, a_{n:n+m-2}, s))$

$\forall r\colon Rel_n, s\colon Rel_m, a_{1:n+m}\colon Atom \mid$
$in_{n+m}(a_{1:n+m}, prod_{n\times m}(r, s)) \Leftrightarrow in_n(a_{1:n}, r) \wedge in_m(a_{n+1:n+m}, s)$

$\forall r\colon Rel_2, a_1, a_2\colon Atom \mid in_2(a_1, a_2, transpose_2(r)) \Leftrightarrow in_2(a_2, a_1, r)$

$\forall r\colon Rel_2, a_{1:2}\colon Atom \mid in_2(a_{1:2}, tc_2(r)) \Leftrightarrow \exists i\colon int \mid i \geq 0 \wedge in_2(a_{1:2}, itrJoin_2(r, i))$

$\forall r\colon Rel_2, i\colon int^{\geq 0} \mid itrJoin_2(r, 0) = r \ \wedge$
$itrJoin_2(r, i+1) = union_2(itrJoin_2(r, i), join_{2\times 2}(r, itrJoin_2(r, i))))$

$\forall r\colon Rel_n, a_{1:n}\colon Atom \mid (finite_n(r) \wedge in_n(a_{1:n}, r)) \Rightarrow 1 \leq ord_n(r, a_{1:n}) \leq card_n(r)$

$\forall r\colon Rel_n, i\colon int \mid (finite_n(r) \wedge 1 \leq i \leq card_n(r))$
$\Rightarrow \exists a_{1:n}\colon Atom \mid in_n(a_{1:n}, r) \wedge ord_n(r, a_{1:n}) = i$

$\forall r\colon Rel_n, a_{1:n}, b_{1:n}\colon Atom \mid (finite_n(r) \wedge in_n(a_{1:n}, r) \wedge in_n(b_{1:n}, r)$
$\wedge ord_n(r, a_{1:n}) = ord_n(r, b_{1:n})) \Rightarrow (a_1 = b_1 \wedge \ldots \wedge a_n = b_n)$

$\forall r\colon Rel_1, a\colon Atom \mid in_1(a, r) \Rightarrow ordInv_1(r, ord_1(r, a)) = a$

$\forall a\colon Atom \mid in_1(a, N[Int]) \Rightarrow i2a(a2i(a)) = a$

$\forall i\colon int \mid in_1(i2a(i), N[Int]) \wedge a2i(i2a(i)) = i$

Fig. 4. Translation rules for Alloy0 expressions. x_i and y_i represent Alloy0 expressions of arity i. The expression $e[e_1/e_2]$ substitutes e_1 for all occurrences of e_2 in e.

are potentially infinite, and thus cardinality is defined only for those that are explicitly known to be finite. For this purpose, we introduce a family of finiteness predicates $finite_n\colon Rel_n \to \texttt{Prop}$ that hold if the user marks a signature as finite, or if finiteness can be inferred[2]. Unlike the Alloy Analyzer that finitizes relations by user-provided, specific upper bounds, Kelloy considers *all* finite domains for those relations that are flagged as finite.

As shown in Fig. 4, we translate Alloy0's cardinality operator to a KFOL function $card_n : Rel_n \to int^{\geq 0}$ which yields the cardinality of an n-ary relation r if it is finite, and is unspecified otherwise. $card_n$ is computed using an ordering function $ord_n : Rel_n \times Atom^n \to int^{>0}$—a bijection from the elements of a finite

[2] Kelloy includes a set of axioms to infer finiteness. For example, the singleton $sin_1(a)$ is always finite, and the union of two finite relations is finite.

$F : dcl \cup formula \to \texttt{Frml}$ $\quad F[\texttt{one } x_n] = one_n(E[x_n])$

$F[x_n \texttt{ in } y_n] = subset_n(E[x_n], E[y_n])$ $\quad F[\texttt{lone } x_n] = lone_n(E[x_n])$

$F[x_n \texttt{ = } y_n] = (E[x_n] = E[y_n])$ $\quad F[\texttt{some } x_n] = some_n(E[x_n])$

$F[\texttt{all } a \texttt{ : set } x_n \mid g] = (\forall\ N[a]\colon Rel_n \mid subset_n(N[a], E[x_n]) \Rightarrow F[g])$

$F[\texttt{some } a \texttt{ : set } x_n \mid g] = (\exists\ N[a]\colon Rel_n \mid subset_n(N[a], E[x_n]) \wedge F[g])$

`KFOL Axioms:`

$\forall r, s\colon Rel_n \mid r = s \Leftrightarrow \forall a_{1:n}\colon Atom \mid in_n(a_{1:n}, r) \Leftrightarrow in_n(a_{1:n}, s)$

$\forall r\colon Rel_n \mid one_n(r) \Leftrightarrow some_n(r) \wedge lone_n(r)$

$\forall r\colon Rel_n \mid lone_n(r) \Leftrightarrow$
$\quad \forall a_{1:n}, b_{1:n}\colon Atom \mid in_n(a_{1:n}, r) \wedge in_n(b_{1:n}, r) \Rightarrow (a_1 = b_1) \wedge \ldots \wedge (a_n = b_n)$

$\forall r\colon Rel_n \mid some_n(r) \Leftrightarrow \exists a_{1:n}\colon Atom \mid in_n(a_{1:n}, r)$

Fig. 5. Translation rules for Alloy0 formulas. x_n denotes an Alloy0 expression of arity n.

relation r to the inclusive integer interval $[1, \ldots, card_n(r)]$. It is easy to show that if the axioms for ord_n, as shown in Fig. 4 hold, $card_n(r)$ gives the cardinality of r. We also define the function $ordInv_1 : Rel_1 \times int \to Atom$ to denote the inverse of $ord_1(r)$ for any unary relation r.

The Alloy0 signature `Int` is translated like other top-level signatures to a constant function symbol $N[Int] : Rel_1$. The Alloy0 cast operators `Int` and `int` are translated using the bijections $i2a : int \to Atom$ and $a2i : Atom \to int$ that give the atom corresponding to an integer value and vice versa. Since in Alloy0, the `int` operator is only applicable to scalar variables, the atom corresponding to `v` in the expression (`int v`) can be retrieved by $ordInv_1(E[v], 1)$.

The `sum` construct is translated using the cardinality function and KFOL's bounded sum operator. Note that, due to the underspecification of $card_1$ and $ordInv_1$, the result of the sum operation is unspecified if $E[S]$ is not finite.

4.4 Formulas

Alloy0 formulas are translated using the auxiliary translation function F given in Fig. 5. Subset and equality formulas are translated using the $subset_i$ predicates and KFOL's built-in (polymorphic) equality. Negation, conjunction, disjunction, and implication operators in Alloy0 are translated to their counterparts in KFOL, and skipped in Fig. 5 in the interest of space. For an Alloy0 expression x of arity n, a multiplicity formula `(mult x)` is translated to a predicate $mult_n(E[x])$ in KFOL where *mult* stands for the multiplicities *some*, *lone*, and *one*. Further axioms give the semantics of these predicates. Universal and existential quantifiers in Alloy0 are translated to those in KFOL where the bounding expression is incorporated into the body of the quantifier.

4.5 The Ordering Module

The Alloy Analyzer provides some library modules that can be used in Alloy problems. Most library functions are inlined and treated like other expressions. The ordering module, however, triggers special optimizations in the Analyzer.

Since this module is widely used, we also treat it specially. The declaration `ord[S]` defines a total order[3] on a signature `S`, which is represented by Alloy0 relations `next:S->S`, `first:S`, and `last:S` which, respectively, denote the successor of an element, and the smallest and the largest elements of the order. These relations are translated to KFOL constants $nextS : Rel_2$, $firstS : Rel_1$, and $lastS : Rel_1$, respectively.

If $finite_1(N[S])$ holds, the previously defined ord_1 function induces an ordering. When $N[S]$ is not finite, $nextS$ relates each element to its immediate successor and thus makes $N[S]$ countable (i.e. isomorphic to the natural numbers). In this case, we extend the axioms for ord_1 to define a bijection from $N[S]$ to $int^{>0}$. The semantics of $nextS$ is then given by:

$$\forall a, b\colon Atom \mid (in_1(a, N[S]) \wedge in_1(b, N[S])) \\ \Rightarrow (in_2(a, b, nextS) \Leftrightarrow ord_1(N[S], b) = ord_1(N[S], a) + 1)$$

Ordered signatures in Alloy0 cannot be empty. This is encoded as $\neg(N[S] = none_1)$. The constant $firstS$ yields the element associated with 1, and $lastS$ yields the one associated with $card_1(N[S])$ if $N[S]$ is finite, and the empty set otherwise.

$$firstS = sin_1(ordInv_1(N[S], 1)) \qquad \neg finite_1(N[S]) \Rightarrow lastS = none_1$$
$$finite_1(N[S]) \Rightarrow (lastS = sin_1(ordInv_1(N[S], card_1(N[S]))))$$

Properties about the elements of a countable set are often proved using induction. KeY provides an induction scheme for its integer type which can be used for this purpose.

4.6 Theoretical Properties

This section discusses the correctness and completeness of our translation. In KFOL, the semantics of the built-in integers is set to $\mathbb{Z}$. KeY's calculus contains a set of inference rules to deal with arithmetic expressions. The calculus, however, cannot be complete because according to Gödel's incompleteness theorem, there is no sound and complete calculus for integer arithmetic [4, §2.7].

The first two theorems state the properties for Alloy0 problems without integers and the third one handles the integer case. The proof sketches for the theorems can be found elsewhere [21].

In the following, we use $sigs(P)$ to denote the set of all signatures (not including the signature Int) defined in an Alloy0 problem P.

Theorem 1 (Correctness). *Let P be an Alloy0 problem that does not contain any integer expression (neither of type `int` nor `Int`) and $fin \subseteq sigs(P)$ a set of signatures. Let $T[\langle P, fin \rangle] = \langle C, k_d, k_a \rangle$ be the translation of P. If $Ax \models_C k_d \Rightarrow k_a$, then P is valid in all structures that interpret the signatures in fin as finite.*

[3] The Alloy Analyzer treats signatures as finite, so the last element of the order does not have a next element.

This theorem implies that the Alloy Analyzer will not produce a counterexample for an Alloy0 problem (not containing integers) that has been proved by Kelloy. If $fin = \emptyset$, Thm. 1 implies that our translation is correct with respect to all structures, i.e. both finite and infinite ones. The Alloy Analyzer, on the other hand, interprets Alloy0 problems only with respect to finite structures.

Completeness, however, holds only for finite structures. In first-order logic, it is not possible to formalize that one type is the powerset of another type. Consequently, our axioms cannot guarantee that the KFOL type Rel_n represents the set of all n-ary relations. This limitation did not appear problematic in practice, but restricts the completeness theorem to the case of finite structures.

Theorem 2 (Relative completeness). *Let P be an Alloy0 problem that does not contain any integer expression (neither of type* `int` *nor* `Int`*). Let $T[\langle P, sigs(P)\rangle] = \langle C, k_d, k_a\rangle$ be the translation of P with all the signatures finitized. If P has no counterexample which interprets all signatures as finite, then $Ax \models_C k_d \Rightarrow k_a$.*

The next theorem considers Alloy0 problems that contain integer expressions. The Alloy Analyzer—due to its methodology—finitizes all domains. Hence, it cannot check problems for validity with respect to $\mathbb{Z}$, but only with respect to a fixed bitwidth. In contrast, integers in KFOL are never interpreted bounded. Therefore, we cannot establish a relationship between arbitrary Alloy0 and KFOL counterexamples. For example, an Alloy0 formula that specifies that there is a maximum integer value is not valid with respect to $\mathbb{Z}$ and thus cannot be proved by Kelloy. However, the Alloy Analyzer will not produce a counterexample either since the formula is valid in all structures with a finite integer domain. Consequently, we generalize the correctness and completeness results by fixing the semantics of the Alloy signature *Int* to $\mathbb{Z}$:

Theorem 3 (Correctness and Completeness modulo integers). *Let P be an Alloy0 problem (which may contain integer expressions) and $fin \subseteq sigs(P)$ a set of signatures. Let $T[\langle P, fin\rangle] = \langle C, k_d, k_a\rangle$ be the translation of P.*

If $Ax \models_C k_d \Rightarrow k_a$, then P is valid in all structures that interpret the signatures in fin as finite and interpret the signature Int as $\mathbb{Z}$.

If $fin = sigs(P)$ and P has no counterexample which interprets all signatures in $sig(P)$ as finite and the signature Int as $\mathbb{Z}$, then $Ax \models_C k_d \Rightarrow k_a$.

5 Reasoning Strategy

The KeY system uses a sequent calculus [15] for proving. A proof-tree is constructed by applying the calculus rules to a proof sequent. This can either be done manually through the GUI, or automatically by KeY's proof search strategy that assigns priorities to all applicable rules and instantiates quantifiers heuristically. We extend the existing strategy by incorporating new (Alloy-specific) calculus rules.

All axioms from the previous sections are implemented as rules for the calculus. For example, axioms that follow the form $\forall x{:}T \mid P(x) \Rightarrow (F(x) \Leftrightarrow G(x))$

become *conditional rewrite rules* that replace $F(x)$ with $G(x)$ when the guard $P(x)$ is known to hold. Since all axioms become rules, the set of Axioms Ax is no longer included in the proof obligation.

The axiom rules rewrite all invocations of the predicates $subset_n$, $disj_n$, $lone_n$, one_n, $some_n$, and relational equality to their quantified definitions. We consider this undesirable for two reasons: (1) formulas grow considerably in size and are thus hard to understand, and (2) when quantifiers cannot be eliminated by skolemization, they require the strategy to provide a suitable instantiation which is a heuristic task.

Our strategy addresses this by only expanding predicates to their definitions when skolemization is applicable. Otherwise, *lemma rules* are used to exploit the semantics of the predicates without rewriting them. For example, a lemma lets us conclude $in_n(a_{1:n}, s_n)$ from $in_n(a_{1:n}, r_n)$ and $subset_n(r_n, s_n)$. This maintains the structural correspondence between formulas and the Alloy0 specification, and allows reasoning on the abstraction level of relations. Overall, the strategy features around 500 lemma rules that have been proved using KeY to follow from the axioms.

The recursive nature of the definition of transitive closure (tc_2) poses a special challenge during proving. In order to simplify proofs and to increase the automation level, we use additional lemmas to capture several simple properties about tc_2, such as its transitivity. Such lemmas are useful for proving some assertions that involve transitive closure. For some cases, however, an induction scheme is required. We can use induction on the integers in the definition of $itrJoin_2$. However, formulas generated this way get cumbersome quickly. We therefore define special induction schemes that are more intuitive, and thus easier to use. Further information can be found elsewhere [21].

6 Evaluation

In this section, we summarize our experimental results of proving Alloy assertions with Kelloy. We proved a total of 22 assertions in 10 Alloy problems of varying sizes and complexity[4]. The chosen collection of Alloy problems is included in the Alloy Analyzer 4.1 distribution and involves all relevant aspects of the language, including transitive closure, integer arithmetic, and the ordering module. In the following, we first elaborate on the automatically proved assertions and show the impact of our reasoning strategy. We then report on the interactive proofs.

6.1 Automation

Out of the 22 assertions, 12 have been proved completely automatically by Kelloy. The remaining 10 assertions required manual guidance as discussed in the next section. Table 1 shows runtime measured on an Intel Core2Quad, 2.8GHz, 8GB memory, and the number of proof steps (i.e. the number of single rule applications) required for each automatic proof.

[4] All Alloy problems and proofs can be found at `http://asa.iti.kit.edu/306.php`

		Kelloy Strategy		Basic Strategy	
Problem	Assertion	Time (Steps)	Result	Time (Steps)	Result
address book	delUndoesAdd	9.3 (2476)	proved	27.1 (5475)	proved
	addIdempotent	0.1 (113)	proved	5.0 (1176)	proved
abstract memory	writeRead	0.8 (567)	proved	1.0 (597)	proved
	writeIdempotent	14.0 (4482)	proved	6.5 (1009)	proved
media assets	hidePreservesInv	0.0 (39)	proved	0.1 (70)	proved
	pasteNotAffectsHidden	15.9 (2619)	proved	time-out (–)	failed
mark sweep	soundness1	3.0 (1195)	proved	time-out (–)	failed
grandpa	noSelfFather	0.0 (77)	proved	0.0 (77)	proved
	noSelfGrandpa	26.5 (3144)	proved	39.8 (3276)	proved
filesystem	FileInDir	0.5 (160)	proved	time-out (–)	failed
	SomeDir	0.2 (205)	proved	time-out (–)	failed
birthday	addWorks	0.1 (129)	proved	1.2 (506)	proved

Table 1. Automatically-proved assertions (time in seconds, time-out after 2h)

Kelloy's strategy uses numerous lemmas to maintain the structure of formulas and to allow reasoning on the abstraction level of relations. To evaluate the impact of these lemmas on the automation level, we compared Kelloy's strategy to a basic strategy that applies all the axiom rules, but none of the lemmas. Table 1 also shows these numbers.

Out of the 12 assertions automatically proved by Kelloy, 4 could not be proved automatically by the basic strategy. Furthermore, although for the remaining assertions, the basic strategy suffices, Kelloy performs most of the proofs faster and requires fewer proof steps. Exceptions include *writeIdempotent* for which the basic strategy is superior, and *noSelfFather* and *writeRead* for which the two strategies perform equally well. These assertions involve simple formulas for which rewriting function symbols as their quantified definitions and using the default quantifier instantiation is sufficient.

6.2 Interactive Proofs

10 of the verified Alloy assertions required user interaction to guide the prover. During interactive proving, the user manually applies rules to the sequent (KeY's GUI makes this quite convenient). The automatic proof search strategy can be invoked anytime on the subgoals of a proof. The strategy then either proves the subgoal, or stops when the maximum number of steps (set by the user) is reached. It is a common practice to frequently invoke the strategy and only focus on those cases that the prover cannot solve on its own.

The manual rule applications can be categorized into three groups of descending complexity: (1) Hypothesis introduction: for example as an induction hypothesis or for a case distinction. (2) Prover guidance: rule applications that allow the strategy to solve a subgoal (more quickly). These include, for example, quantifier instantiations and case distinctions on formulas from the sequent. (3) Non-essential steps: simple steps that the strategy would find automatically but the user prefers to do manually to keep track of the proof.

The complexity of the proofs for the 10 interactively-proved assertions differ considerably. 7 assertions required only very few (max. 10) interactive steps. One example of such a proof is the *completeness* assertion for the mark and sweep Alloy problem: only one, yet quite a complex step to handle transitive closure

had to be done interactively. The remaining 3 assertions required between 36 and 291 interactive steps. The assertion stating the correctness of Dijkstra's deadlock prevention algorithm had the most complex proof: we introduced seven intermediate hypotheses that were proved using induction. Overall, the proof took 18875 steps out of which 7/219/65 were manual steps of the categories 1/2/3. A proof of this complexity can be conducted by an experienced user in roughly one work day.

In order to evaluate the impact of the user's expertise on the interactive proof process, we asked an Alloy user with no previous experience in KeY to prove a soundness assertion for the mark and sweep Alloy problem. Out of 1389 steps, 207 (2/57/148) have been performed manually. The proof, including a proof-sketch on paper, was conducted within two work days.

In comparison to that, an experienced user in both Alloy and KeY, proved the same assertion in 4 hours with only 10 (5/1/4) interactive steps out of a total of 9372 steps. The higher number of total steps, but drastically smaller number of interactive steps show that it requires some experience to effectively leverage the automation strategy. However, the experiment indicates that the proof process is intuitive enough for a user with no prior experience in Kelloy.

7 Conclusion

We presented Kelloy, a tool for full verification of Alloy problems. We formally defined a translation of the Alloy language to the first-order logic of the theorem prover KeY and discussed its correctness and completeness. To our knowledge, Kelloy is the only system that provides proof capability for the whole Alloy language (including integers, cardinality, and the ordering module).

We used Kelloy to prove some challenging Alloy assertions semi-automatically. Our experiments showed that usually only structurally complex systems or systems that involve inductive properties require user interaction. Moreover, considerable parts of a proof can be automated while the user only performs central steps interactively. In many cases, however, conducting a proof using Kelloy requires in-depth knowledge of the analyzed Alloy problem; the required time and effort depend on the user's experience in the tool. Kelloy is thus intended to be used in conjunction with automatic approaches as described in [7].

The presentation of proof obligations in Kelloy resembles the original Alloy structure such that even a non-expert in KeY can conduct interactive proofs. The effort of interaction might be further lowered in the future, for example by pretty-printing expressions in the Alloy syntax or integration of the Alloy Analyzer for counterexample generation and visualization.

Several program analysis tools (see [7] for some examples) use Alloy as their specification languages. Incorporating Alloy into KeY raises the opportunity of full verification of programs that contain Alloy specifications, leveraging both the expressiveness of Alloy and the dynamic logic of KeY. Pursuing this idea is left for future work.

Acknowledgement

We thank Peter H. Schmitt and the anonymous reviewers for their helpful comments. This work was funded in part by the MWK-BW grant 655.042/taghdiri/1.

References

1. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae (2007)
2. Arkoudas, K., Khurshid, S., Marinov, D., Rinard, M.: Integrating model checking and theorem proving for relational reasoning. In: (RMICS) (2003)
3. Athena. `http://people.csail.mit.edu/kostas/dpls/athena/`
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. Springer-Verlag (2007)
5. Buss, S.R.: First-order proof theory of arithmetic. In: Handbook of Proof Theory, pp. 79–147. Elsevier (1998)
6. van Eijck, J.: Defining (reflexive) transitive closure on finite models. `http://homepages.cwi.nl/~jve/papers/08/pdfs/FinTransClosRev.pdf`
7. El Ghazi, A.A., Geilmann, U., Ulbrich, M., Taghdiri, M.: A Dual-Engine for Early Analysis of Critical Systems. In: (DSCI) (2011)
8. El Ghazi, A.A., Taghdiri, M.: Analyzing Alloy Constraints using an SMT Solver: A Case Study. In: (AFM) (2010)
9. El Ghazi, A.A., Taghdiri, M.: Relational Reasoning via SMT Solving. In: (FM) (2011)
10. Fortune, S., Leivant, D., O'Donnell, M.: The expressiveness of simple and second-order type structures. (J. ACM) (1983)
11. Frias, M., Lopez Pombo, C.: Interpretability of first-order linear temporal logics in fork algebras. In: Journal of logic and algebraic programming (2006)
12. Frias, M., Lopez Pombo, C., Aguirre, N.: An equational calculus for Alloy. In: (ICFEM) (2004)
13. Frias, M., Lopez Pombo, C., Baum, G., Aguirre, N., Maibaum, T.: Taking Alloy to the movies. In: (FME) (2003)
14. Frias, M., Lopez Pombo, C., Moscato, M.: Alloy Analyzer+PVS in the analysis and verification of Alloy specifications. In: (TACAS) (2007)
15. Gentzen, G.: Untersuchungen über das logische Schließen. Mathematische Zeitschrift (1935)
16. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press (2006)
17. Jackson, D., Wing, J.: Lightweight formal methods. In: IEEE Computer (1996)
18. Köker, C.: Discharging Event-B proof obligations. Studienarbeit, Universität Karlsruhe (TH) (2008)
19. Lev-Ami, T., Immerman, N., Reps, T.W., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating reachability using first-order logic with applications to verification of linked data structures. Logical Methods in Computer Science 5(2) (2009)
20. Shankar, N., Owre, S., Rushby, J., Stringer-Calvert, D.: PVS Prover Guide. Computer Science Laboratory, SRI International (1999)
21. Ulbrich, M., Geilmann, U., Ghazi, A.A.E., Taghdiri, M.: On proving alloy specifications using KeY. Tech. Rep. 2011-37, Karlsruhe Institute of Technology (2011)