Automatic Unbounded Verification of Alloy Specifications with Prover9

Alcino Cunha and Nuno Macedo

HASLab — High Assurance Software Laboratory
INESC TEC & Universidade do Minho, Braga, Portugal

May 2011

Abstract. Alloy is an increasingly popular lightweight specification language based on relational logic. Alloy models can be automatically verified within a bounded scope using off-the-shelf SAT solvers. Since false assertions can usually be disproved using small counter-examples, this approach suffices for most applications. Unfortunately, it can sometimes lead to a false sense of security, and in critical applications a more traditional unbounded proof may be required. The automatic theorem prover Prover9 has been shown to be particularly effective for proving theorems of relation algebras [7], a quantifier-free (or point-free) axiomatization of a fragment of relational logic. In this paper we propose a translation from Alloy specifications to fork algebras (an extension of relation algebras with the same expressive power as relational logic) which enables their unbounded verification in Prover9. This translation covers not only logic assertions, but also the structural aspects (namely type declarations), and was successfully implemented and applied to several examples.

1 Introduction

The Alloy specification language [8] was created by following a different approach than the so called "classic" formal methods. It was built to be *lightweight* [9] and, instead of focusing on theorem proving, the emphasis is on automatic analysis. Alloy's underlying logic is a kind of relational logic, making it easier to write and read specifications without the need to learn complicated concepts. On the other hand, it is also influenced by object modeling languages, from which it inherits the navigational style and type hierarchy. As for the verification of the model, the Alloy Analyzer tool is provided, which automatically verifies the specifications within a bounded scope using off-the-shelf SAT solvers.

However, sometimes bounded verification is not enough. In safety-critical systems, for instance, there is a need to make sure that the program is *always* correct, i.e., we must perform *unbounded* verification. The only way to do this is to mathematically prove that the specifications are correct. Since Alloy's basic elements are relations, relational logic provides a natural framework to reason about their specifications. Moreover, we believe that using a *point-free* (PF) notation, a style where there are no variables or quantifications, as opposed to

point-wise (PW), provides a simpler framework where proofs can be carried out by simple equational steps (see e.g. [15]). Fork algebras are the point-free counterpart to relational logic. They extend the more traditional relation algebras with products in order to regain the expressiveness of relational logic, and will be our logic of choice to reason about Alloy specifications.

Being as expressive as first-order logic, fork algebras are also *undecidable*: in principle this would restrict us to perform manual proofs, eventually assisted by interactive theorem provers. However, off-the-shelf automatic theorem provers (ATPs) are becoming increasingly efficient and are nowadays able to solve complex problems. Prover9 [14] is an ATP for first-order logic and equational logic. Studies have show that it is the most efficient off-the-shelf ATP to deal with relation algebras [2], and it has been used to prove several properties about them [7]. Building on these results, we propose a framework for automatic unbounded verification of Alloy specifications using Prover9. The key contribution of this framework is a translation from Alloy models into fork algebras, that covers not only logic assertions, but also other structural aspects of the model, such as type declarations.

The next section briefly presents Alloy with an example. Sect. 3 presents fork algebras (assuming previous knowledge of relation algebras) and describes how relations with arbitrary arity can be represented in this formalism. Sect. 4 and 5 present the translation from Alloy models to fork algebras: the former focuses on formulas and the latter on the structural aspects. Sect. 6 describes the implementation of the translation, and presents the result of translating our running example. Sect. 7 discusses some related work, and Sect. 8 concludes the paper with some reflexions and suggestions for future work.

2 Alloy

We will briefly present the Alloy language by following a very simple example, a model of a university with students and the courses they have completed, which is presented in Fig. 1. Roughly, an Alloy model is divided in two parts. The first, the *signature* declarations, defines the existing types and the relations between them. In our example, `Student` and `Professor` both extend the signature `Person`, inducing a type hierarchy. `Person` is also declared as abstract, meaning that there are no persons besides those contained in its sub-signatures. A particularity of Alloy relations is that they can have arbitrary arity. For instance, `course` is a ternary relation that, for a particular university, relates students with the courses they have completed. We can also attach *multiplicities* to signature and relation declarations. For example, the multiplicity `some` in relation `lecturer` forces that at least one professor is lecturing each course. The second part of the model consists of *facts*, that define constraints and properties of the model, and the *assertions* we want to verify.

Predicates can be defined to be reused in facts and assertions. In our example, we define one predicate to model the invariant of the university (students with completed courses must be enrolled in the university, and to have completed a

```
abstract sig Person {}
sig Student, Professor extends Person {}
sig Course {
  lecturer : some Professor,
  depends : set Course
}
sig University {
  enrolled : set Student,
  courses  : Student -> Course
}
pred inv[u : University] {
  (u.courses).Course in u.enrolled
  all s : Student | (s.(u.courses)).*depends in s.(u.courses)
}
pred enroll[u, u' : University, s : Student] {
  u'.enrolled = u.enrolled + s
  u'.courses = u.courses
}
assert {
  all u,u':University,s:Student | inv[u] and enroll[u,u',s] => inv[u']
}
```

Fig. 1. Alloy example

course a student must also complete the courses it depends on), and another to model an operation which enrolls a new student to the university. The assertion to be verified in this case is that the invariant is preserved by the operation.

In order to make translation of the Alloy models easier, we restrict the grammar to an essential core (see Appendix A), to which almost every other constructions can trivially be reduced.

3 Fork algebras

Relational logic (RL) is a characterization of first-order logic (FOL) with relational operators. Although it introduces different notations, RL is as expressive as FOL. If we remove all quantifiers (and variables) from RL, we obtain the *calculus of relations* (CR), a PF fragment of RL. In CR, formulas consist of boolean combinations of inequations, formulas of the form $R \subseteq S$. This calculus is axiomatized by *relation algebras* (RAs) (for a standard axiomatization see [12]), which however is not as expressive as RL (only to a fragment with 3 variables).

Fork algebras (FAs) [4] were created to overcome this expressiveness limitation. They extend CR by introducing pairing (products) to the base set of the algebra, and a new operator *fork*, denoted by ∇, and defined by $(x, y)\,(R \nabla S)\,z \equiv x\,R\,z \wedge y\,S\,z$. FAs are as expressive as RL and their axiomatization, an extension of RA, is the following.

Definition 1 (Closure Fork Algebras). *A* fork algebra *is an algebraic structure*

$$\langle U, \cup, \cap, ^-, \bot, \top, \cdot, id, ^\circ, \nabla \rangle$$

where $\langle U, \cup, \cap, ^-, \bot, \top, \cdot, id, ^\circ \rangle$ *is a relation algebra and for all* $R, S, T, Q \in U$*,*

$$\begin{aligned}
&R \nabla S = ((id \nabla \top) \cdot R) \cap ((\top \nabla id) \cdot S) \\
&(R \nabla S)^\circ \cdot (T \nabla Q) = (R^\circ \cdot T) \cap (S^\circ \cdot Q) \\
&(id \nabla \top)^\circ \nabla (\top \nabla id)^\circ \subseteq id \\
&R^* = id \cup R^* \cdot R \\
&\top \cdot S \cdot R^* \subseteq \top \cdot S \cup (\overline{\top \cdot S} \cap \top \cdot S \cdot R) \cdot R^*
\end{aligned}$$

The relations $(id \nabla \top)^\circ$ and $(\top \nabla id)^\circ$ select the first and second element from a pair, and will be denoted by π_1 and π_2, respectively. An operator $\times$ that applies two relations in parallel can be defined as $R \times S \triangleq (\pi_1 \cdot R) \nabla (\pi_2 \cdot S)$.

3.1 Handling relations of arbitrary arity

While in Alloy relations can have an arbitrary arity, FA is restricted to binary relations. As such, a mechanism must be devised to "binarize" arbitrary relations. Unary relations (sets) can be represented in FA using coreflexives, i.e., fragments of the identity relation that filter elements of the given set. More precisely, a unary relation $\mathtt{R}$ can represented by a coreflexive $\Phi_R \subseteq id$ such that $x \in \mathtt{R}$ iff $x \; \Phi_R \; x$. For n-ary relations with $n > 2$ a mechanism analogous to uncurrying can be used: an Alloy relation $\mathtt{R} : A_1 \to \cdots \to A_n$ can be represented by the binary relation $R : A_n \times \cdots \times A_2 \to A_1$, whose domain is the nested product (associated to the right) of the last $n-1$ columns of $\mathtt{R}$. Domains and ranges appear reversed in the binary version to allow a direct encoding of composition: binary relations $\mathtt{R} : A \to B$ and $\mathtt{S} : B \to C$ can be composed in Alloy as $\mathtt{R.S} : A \to C$ using the dot join; by reversing domains and ranges, this expression can be directly translated to the FA composition $R \cdot S$. Throughout the presentation, the binary representation of an n-ary Alloy relation $\mathtt{R}$ will be denoted as Φ_R if $n = 1$ or just R if $n > 1$. We will often abuse the notion of arity and classify the binary version of an n-ary relation also as n-ary. Given a relation $\mathtt{R}$ its arity will be denoted by $|R|$.

Notice that in Alloy composition is not limited to binary relations. For example, relations $\mathtt{R} : A_1 \to \cdots \to A_n$ and $\mathtt{S} : B_1 \to \cdots \to B_m$ can be composed as $\mathtt{R} \cdot \mathtt{S} : A_1 \to \cdots \to A_{n-1} \to B_2 \to \cdots \to B_m$, provided $n + m > 2$. In order to translate this directly it is convenient to have an n-ary composition operator in FA. Given two relations R and S, where R denotes a n-ary relation with $n > 1$, the composition of R after S will be denoted by $R \bullet^n S$, and defined as follows:

$$R \bullet^n S = \begin{cases} R \cdot S & \text{if } n = 2 \\ R \bullet^{n-1} (id \times S) & \text{if } n > 2 \end{cases}$$

It is trivial to show by induction that n-ary composition satisfies analogous properties to normal binary composition, in particular identity and associativity:

$$R \bullet^n id = R \quad \wedge \quad id \bullet^2 R = R$$
$$R \bullet^n (S \bullet^m T) = (R \bullet^n S) \bullet^{n+m-2} T$$

These properties enable us to calculate directly with this operator without the the need to expand its definition. Composition with unary relations can be performed with normal binary composition since these are represented as binary coreflexives.

In an n-ary relation the notion of range and domain is somehow arbitrary: for example, given a ternary relation $\mathtt{R} : A \to B \to C$ we may want to compose it via the middle column with a relation $\mathtt{S} : B \to D$. To allow this, we will define an operator to rotate a relation: given an n-ary relation $R : A_n \times \cdots \times A_2 \to A_1$, its right-rotation will be denoted by $\overrightarrow{R} : A_{n-1} \times \cdots \times A_1 \to A_n$ and can be defined as

$$\overrightarrow{R} = X_{n-1}^{n-1} \cdot (R \nabla X_1^{n-1} \nabla \ldots \nabla X_{n-2}^{n-1})^\circ$$

where $(R \nabla \ldots \nabla S)$ represents the right-nested fork $R \nabla (\ldots \nabla S))$, and X_i^n selects the ith component of a right-nested n-ary tuple, according to the following definition:

$$X_i^n = \begin{cases} id & \text{if } n = i = 1 \\ \pi_1 & \text{if } n > 1 \wedge i = 1 \\ X_{i-1}^{n-1} \cdot \pi_2 & \text{if } n > 1 \wedge i > i \end{cases}$$

Note that, when R is a binary relation, $\overrightarrow{R} = R^\circ$. We will denote by $\overset{k\rightarrow}{R}$ the application of the rotate k times. Likewise to n-ary composition, this operator satisfies some useful calculational properties, namely:

$$\overset{|R|\rightarrow}{R} = R \quad \wedge \quad R \bullet^{|R|} S \equiv \overrightarrow{S} \bullet^{|S|} \overset{(|R|-1)\rightarrow}{R}$$

4 Translating Alloy formulas to FA

While in RL variables can only be used in relation application, i.e membership testing, in Alloy they can be used as normal unary relations in relational formulas. For example, we can have the composition `R.x.S`, where `x` denotes a variable, and `R` and `S` arbitrary relational expressions. This feature makes the direct translation of Alloy formulas to FA quite difficult, since standard heuristics for variable elimination cannot be used. To overcome this problem, our approach is to first expand Alloy formulas to standard RL, where variables only appear applied to relations, and then use a RL to FA translation to remove the variables. As will be seen in Sect. 4.3, this allows us to use powerful heuristics for variable elimination and output much simpler FA formulas.

$$[\![\texttt{!R}]\!] \equiv \neg[\![\texttt{R}]\!]$$
$$[\![\texttt{R \&\& S}]\!] \equiv [\![\texttt{R}]\!] \wedge [\![\texttt{S}]\!]$$
$$[\![\texttt{all x : X | S}]\!] \equiv \langle \forall x : [\![\texttt{x in X}]\!] : [\![\texttt{S}]\!] \rangle$$
$$[\![\texttt{some X}]\!] \equiv \langle \exists x_1, \ldots, x_{|\texttt{X}|} :: [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X}]\!] \rangle$$
$$[\![\texttt{X in Y}]\!] \equiv \langle \forall x_1, \ldots, x_{|\texttt{X}|} : [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X}]\!] : [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{Y}]\!] \rangle$$

$$[\![x \in \texttt{v}]\!] \equiv x \, id \, v$$
$$[\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{R}]\!] \equiv x_1 \, R \, (x_{|2|}, \ldots, x_{|\texttt{X}|})$$
$$[\![x \in \texttt{S}]\!] \equiv x \, \Phi_S \, x$$
$$[\![x \in \texttt{univ}]\!] \equiv \mathit{true}$$
$$[\![x \in \texttt{none}]\!] \equiv \mathit{false}$$
$$[\![(x_1, x_2) \in \texttt{iden}]\!] \equiv x_1 \, id \, x_2$$
$$[\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X + Y}]\!] \equiv [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X}]\!] \vee [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{Y}]\!]$$
$$[\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X \& Y}]\!] \equiv [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X}]\!] \wedge [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{Y}]\!]$$
$$[\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X - Y}]\!] \equiv [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X}]\!] \wedge \neg[\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{Y}]\!]$$
$$[\![(x_1, x_2) \in \texttt{\~{} X}]\!] \equiv [\![(x_2, x_1) \in \texttt{X}]\!]$$
$$[\![(x_1, \ldots, x_{|\texttt{X}|+|\texttt{Y}|-2}) \in \texttt{X . Y}]\!] \equiv \langle \exists k :: [\![(x_1, \ldots, x_{|\texttt{X}|-1}, k) \in \texttt{X}]\!] \wedge [\![(k, x_{|\texttt{X}|}, \ldots, x_{|\texttt{X}|+|\texttt{Y}|-2}) \in \texttt{Y}]\!] \rangle$$
$$[\![(x_1, \ldots, x_{|\texttt{X}|+|\texttt{Y}|}) \in \texttt{X -> Y}]\!] \equiv [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X}]\!] \wedge [\![(x_{|\texttt{X}|+1}, \ldots, x_{|\texttt{X}|+|\texttt{Y}|}) \in \texttt{Y}]\!]$$
$$[\![(x_1, \ldots, x_{|\texttt{Y}|}) \in \texttt{X <: Y}]\!] \equiv [\![x_1 \in \texttt{X}]\!] \wedge [\![(x_1, \ldots, x_{|\texttt{Y}|}) \in \texttt{Y}]\!]$$
$$[\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X :> Y}]\!] \equiv [\![x_{|\texttt{X}|} \in \texttt{Y}]\!] \wedge [\![(x_1, \ldots, x_{|\texttt{X}|}) \in \texttt{X}]\!]$$

Fig. 2. Translation from Alloy formulas to RL

4.1 From Alloy formulas to RL

Fig. 2 presents the translation of Alloy formulas to RL. Alloy formulas are boolean combinations of atomic formulas of shape `some X` or `x in X`, where `X` and `Y` denote Alloy relational formulas. The first set of rules in Fig. 2 introduces variables in order to convert these atomic formulas into relation applications of shape $x \in \texttt{X}$, where x denotes a tuple of variables and `X` is still an Alloy relational formula. The second set of rules expands these into boolean combinations of standard relation applications by computing the expected semantics of relational operators. Moreover, variable occurrences are translated to equality tests (using the identity relation), and constant unary relations (denoting signatures) to the corresponding coreflexives that filter values of that type.

4.2 From RL to FA

The translation from RL to FA will be defined as a *strategic rewriting* process [10]: basic rewrite rules are combined using strategic combinators in order

to build powerful rewrite systems. We will use the following combinators: *sequence*, a binary combinator denoted by $\triangleright$, which applies the second rule to the result of the first, if successful; *choice*, a binary combinator denoted by $\oslash$, which applies the first rule or, if it fails, applies the second; *many*, which applies a rule repetitively until it fails; and *once*, which applies a rule once somewhere inside a term.

For example, we can easily define a strategy to eliminate universal quantifiers and implications as follows:

$$\phi \Rightarrow \psi \rightsquigarrow \neg\phi \vee \psi \tag{1}$$

$$\langle \forall z :: \phi \rangle \rightsquigarrow \neg\langle \exists z :: \neg\phi \rangle \tag{2}$$

$$\langle \forall z : \phi : \psi \rangle \rightsquigarrow \langle \forall z :: \phi \Rightarrow \psi \rangle \tag{3}$$

$$\mathit{normalize} \triangleq \mathit{many}\,(\mathit{once}\,(1) \oslash \mathit{once}\,(2) \oslash \mathit{once}\,(3))$$

To distinguish arbitrary RL formulas from purely relational expressions, variables ϕ, ψ, ..., will be used to denote the former, and variables R, S, ..., to denote the latter.

Intuitively, the translation of a normalized RL formula to an equivalent FA formula will proceed as follows:

1. The formula will first be universally quantified over a fresh pair of special variables, denoted $\mathbf{x}$ and $\mathbf{y}$, that will represent arbitrary output and input values, respectively. The following rule performs this transformation.

$$\mathit{insertVars} \triangleq \phi \rightsquigarrow \langle \forall \mathbf{x}, \mathbf{y} :: \phi \rangle$$

2. Together with quantifier elimination, all relation applications will iteratively be uniformed to operate on the same variables, so that boolean combinations of applications can be combined into a single application using the following simple strategy:

$$u\,R\,v \wedge u\,S\,v \rightsquigarrow u\,(R \cap S)\,v \tag{4}$$

$$u\,R\,v \vee u\,S\,v \rightsquigarrow u\,(R \cup S)\,v \tag{5}$$

$$\neg(u\,R\,v) \rightsquigarrow u\,\overline{R}\,v \tag{6}$$

$$\mathit{aggregate} \triangleq \mathit{once}\,(4) \oslash \mathit{once}\,(5) \oslash \mathit{once}\,(6)$$

   Here, u and v denote arbitrary variables or variable tuples.
3. In the end we will obtain a single relational expression applied to the quantified special input and output variables. Those can then be eliminated by the following rule:

$$\mathit{dropVars} \triangleq \langle \forall u, v :: u\,R\,v \rangle \rightsquigarrow R = \top$$

We will now detail the strategies for uniforming applications and existential quantifier elimination.

Uniforming Applications. To simplify the presentation we will represent variables by indices (similar to de Bruijn indices): existential quantifiers do not explicitly mention variable names and a quantified variable is referred to by the natural number that indexes the respective quantifier. For example, the formula $\langle \exists :: \langle \exists :: 2\, R\, 1 \rangle \rangle$ is equivalent to $\langle \exists x_1 :: \langle \exists x_2 :: x_2\, R\, x_1 \rangle \rangle$. The depth of an application is the number of existential quantifiers that enclose it. The depth of a formula is the maximum depth of all applications. In the following presentation, n denotes the depth of the application that matches with the left hand side of a rewrite rule.

To uniform the input of applications we generalize it to a tuple containing all existentially quantified variables at that depth, and use the selection operator to choose the desired variable:

$$i\, R\, j = i\, (R \cdot X_j^n)\, (1, \ldots, n)$$

In order to keep the tuple of quantified variables only on the input side, a slightly different strategy is used to uniform the output, motivated by the following calculation:

$$\begin{aligned} i\, (R \cdot X_j^n)\, (1, \ldots, n) &= \langle \exists z :: z\, X_i^n\, (1, \ldots, n) \wedge z\, (R \cdot X_j^n)\, (1, \ldots, n) \rangle \\ &= \langle \exists z :: z\, (X_i^n \cap (R \cdot X_j^n))\, (1, \ldots, n) \rangle \\ &= \langle \exists z :: \mathbf{x}\, \top\, z \wedge z\, (X_i^n \cap (R \cdot X_j^n))\, (1, \ldots, n) \rangle \\ &= \mathbf{x}\, (\top \cdot (X_i^n \cap (R \cdot X_j^n)))\, (1, \ldots, n) \end{aligned}$$

The quantified (temporary) variable z is introduced to bind the output with the appropriate variable of the input tuple. The quantification is then eliminated by introducing an intersection, composing with $\top$, and applying the resulting expression to the special variable $\mathbf{x}$. A strategy to uniform applications at any given depth can thus be implemented as follows:

$$\mathit{uniform} \triangleq \mathit{once}\, (i\, R\, j \rightsquigarrow \mathbf{x}\, (\top \cdot (X_i^n \cap (R \cdot X_j^n)))\, (1, \ldots, n))$$

Although this strategy only covers applications to a single variable, it is rather straightforward to generalize it to applications to tuples of variables using forks of projections, as described in [11].

Removing Existential Quantifiers. The strategy to remove existential quantifiers is similar to the one used above to remove the temporary quantifier. While $n > 1$, we compose the terms with the relation $\langle id, \top \rangle$ on the right side to cut the rightmost existential quantified variable from the input variable tuple:

$$\langle \exists :: \mathbf{x}\, R\, (1, \ldots, n) \rangle \rightsquigarrow \mathbf{x}\, (R \cdot \langle id, \top \rangle)\, (1, \ldots, n-1) \tag{7}$$

If $n = 1$, the term is composed with $\top$ and applied to the special variable $\mathbf{y}$:

$$\langle \exists :: \mathbf{x}\, R\, 1 \rangle \rightsquigarrow \mathbf{x}\, (R \cdot \top)\, \mathbf{y} \tag{8}$$

$$dropExists \triangleq once\,(7) \oslash once\,(8)$$

Uniforming applications enables the application of strategy *aggregate*, reducing all applications at the deepest level of the formula to a single application. This in turn allows strategy *dropExists* to remove one level of quantifiers, thus reducing the depth of the formula. These steps can be combined in the following strategy:

$$shorten \triangleq uniform \triangleright aggregate \triangleright dropExists$$

Finally, the translation from RL to FA iteratively shortens the depth of the formula until the external universally quantified variables can be dropped.

$$translate \triangleq normalize \triangleright insertVars \triangleright many\,(dropVars \oslash shorten)$$

4.3 Heuristic simplification

An heuristic rewrite strategy can be defined to further simplify the resulting FA expressions. This simplification is based on two key ideas: the relaxation of the form of the resulting formula, which we now allow to be an inequation of the form $R \subseteq S$; and the use of additional relational operators in order to remove quantifiers. We will only present a simplified version of this strategy: the full set of rules included in the implementation can be found in [11].

The heuristic simplification strategy is defined as

$$simplify \triangleq many\,(FOLrules \oslash definitions \oslash FArules)$$

where

- *FOLrules* performs several simplifications related to first-order logic. Essentially, it is the *choice* of rules like those presented in Table 1, applied *once*. Note that we present simplified versions of the rules. The implemented version considers the commutativity and associativity of logical operators. Since we allow arbitrary inequations as result, this strategy also tries to push some expressions into the range of universal quantifiers, thus distributing the complexity of the formula between both sides of the inequation.
- *definitions* applies definitions of relational operators in order to remove quantifiers. For example, it applies the definition of composition and converse in order to remove some existential quantifiers, according to the following rules:

$$\langle \exists w :: u\,R\,w \wedge w\,S\,v \rangle \rightsquigarrow u\,(R \cdot S)\,v$$
$$\langle \exists w :: u\,R\,w \wedge v\,S\,w \rangle \rightsquigarrow u\,(R \cdot S^{\circ})\,v$$
$$\langle \exists w :: w\,R\,u \wedge w\,S\,v \rangle \rightsquigarrow u\,(R^{\circ} \cdot S)\,v$$

  In particular, it introduces the n-ary composition operator introduced in Sect. 3.1 whenever possible, by generalizing the rules above. Among others, we have the following rule:

$$\langle \exists x_1 :: u\,R\,(x_1, \ldots, x_n) \wedge x_n\,S\,v \rangle \rightsquigarrow u\,(R \bullet^n S)\,(x_1, \ldots, x_{n-1}, v)$$

Conjunction	Disjunction	Implication
$\psi \wedge \mathit{true} \rightsquigarrow \psi$	$\psi \vee \mathit{false} \rightsquigarrow \psi$	$\neg\psi \vee \phi \rightsquigarrow \psi \Rightarrow \phi$
$\psi \wedge \mathit{false} \rightsquigarrow \mathit{false}$	$\psi \vee \mathit{true} \rightsquigarrow \mathit{true}$	$\mathit{false} \Rightarrow \psi \rightsquigarrow \mathit{true}$
$\psi \wedge \psi \rightsquigarrow \psi$	$\psi \vee \psi \rightsquigarrow \psi$	$\mathit{true} \Rightarrow \psi \rightsquigarrow \psi$
$\neg(\psi \wedge \phi) \rightsquigarrow \neg\psi \vee \neg\phi$	$\neg(\psi \vee \phi) \rightsquigarrow \neg\psi \wedge \neg\phi$	$\psi \Rightarrow (\phi \Rightarrow \varphi) \rightsquigarrow (\psi \wedge \phi) \Rightarrow \varphi$
Negation	Quantifiers	
$\neg\neg\psi \rightsquigarrow \psi$	$\langle \forall u : \psi : \phi \Rightarrow \varphi \rangle \rightsquigarrow \langle \forall u : \psi \wedge \phi : \varphi \rangle$	
$\neg \mathit{true} \rightsquigarrow \mathit{false}$	$\langle \exists u : \psi : \phi \rangle \rightsquigarrow \langle \exists u :: \psi \wedge \phi \rangle$	
$\neg \mathit{false} \rightsquigarrow \mathit{true}$	$\langle \forall u : \psi : \langle \forall v : \phi : \varphi \rangle \rangle \rightsquigarrow \langle \forall u, v : \psi \wedge \phi : \varphi \rangle$	
	$\langle \exists u : \psi : \langle \exists v : \phi : \varphi \rangle \rangle \rightsquigarrow \langle \exists u, v : \psi \wedge \phi : \varphi \rangle$	

Table 1. FOL rules.

Meet rules	Join rules	Complement rules
$A \cap A \rightsquigarrow A$	$A \cup A \rightsquigarrow A$	$\overline{R \cdot S} \rightsquigarrow R^\circ \backslash \overline{S}$
$A \cap \top \rightsquigarrow A$	$A \cup \top \rightsquigarrow \top$	$\overline{R \backslash S} \rightsquigarrow R^\circ \cdot \overline{S}$
$A \cap \bot \rightsquigarrow \bot$	$A \cup \bot \rightsquigarrow A$	$\overline{S} \subseteq \overline{R} \rightsquigarrow R \subseteq S$
$(A \cap B)^\circ \rightsquigarrow A^\circ \cap B^\circ$	$(A \cup B)^\circ \rightsquigarrow A^\circ \cup B^\circ$	$\overline{\overline{R}} \rightsquigarrow R$
$\overline{A \cup B} \rightsquigarrow \overline{A} \cap \overline{B}$	$\overline{A \cap B} \rightsquigarrow \overline{A} \cup \overline{B}$	$\overline{R^\circ} \rightsquigarrow R^\circ$
Converse rules	Division rules	Product rules
$R^{\circ\circ} \rightsquigarrow R$	$R \subseteq S \backslash T \rightsquigarrow S \cdot R \subseteq T$	$\pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \rightsquigarrow (R \nabla S)\rangle$
$(R \cdot S)^\circ \rightsquigarrow S^\circ \cdot R^\circ$	$\bot \backslash R \rightsquigarrow \mathit{true}$	$(R \nabla S)^\circ \cdot (A \nabla B) \rightsquigarrow R^\circ \cdot A \cap S^\circ \cdot B$
$\top \subseteq R^\circ \rightsquigarrow \top \subseteq R$	$R \backslash \top \rightsquigarrow \mathit{true}$	$(\mathit{id} \nabla \top) \cdot R \rightsquigarrow (R \nabla \top \cdot R)$
$R^\circ \subseteq \bot \rightsquigarrow R \subseteq \bot$	$\mathit{id} \backslash R \rightsquigarrow R$	$(R \cdot \pi_1 \nabla S \cdot \pi_2) \rightsquigarrow R \times S$

Table 2. FA rules.

Expressions similar to this one, but with x_1 on a different position of the tuple are dealt with other rules, which resort to the rotate operator. Besides composition, we also introduce forks, complement, and other derived operators characterized by powerful algebraic laws, such as the relational division operators. These are particularly interesting because they allow us to remove universal quantifiers, according to the following rules:

$$\langle \forall w : w\, R\, u : w\, S\, v \rangle \rightsquigarrow u\, R \backslash S\, v$$
$$\langle \forall w : u\, R\, w : v\, S\, w \rangle \rightsquigarrow u\, R / S\, v$$

– *FArules* performs several simplifications related to FA operators. Among many others, it is the *choice* of all rules presented in Table 2, applied *once*. These rules correspond to equations of the axiomatization of FA presented in Sect. 3 or can easily be derived from them.

With this strategy we can redefine the translation from Alloy to FA as follows:

$$\mathit{translate} \triangleq \mathit{simplify} \triangleright (\mathit{dropVars} \oslash (\mathit{normalize} \triangleright \mathit{insertVars} \triangleright \mathit{many}\,(\mathit{dropVars} \oslash (\mathit{simplify} \triangleright \mathit{shorten}))))$$

By applying the heuristic strategy we highly simplify the formula before the automatic translation kicks in. At this point, the expression might even be in

a shape where variables can be dropped. Since it keeps the range in universal quantifiers, the *dropVars* strategy must be redefined as follows:

$$dropVars \triangleq (\langle \forall u, v :: u\, R\, v \rangle \rightsquigarrow R = \top) \oslash (\langle \forall u, v : u\, R\, v : u\, S\, v \rangle \rightsquigarrow R \subseteq S)$$

4.4 Translating the closures

Translating the reflexive-transitive and transitive closures requires a slightly different approach. In this section only the translation of the reflexive-transitive closure will be presented, but a similar translation can be applied to the reflexive closure. Due to the closure type restrictions, our goal is to obtain an expression of the form:

$$[\![(x, y) \in *\mathtt{R}]\!] \equiv (a_1, \ldots, a_k, x)\, (translate*[\![(x, y) \in \mathtt{R}]\!])^*\, (a_1, \ldots, a_k, y) \quad (9)$$

Where $(a_1, \ldots, a_k)$ are the k free variables occurring in $\mathtt{R}$, with type $A_1 \times \cdots \times A_k$. This operation "lifts" the variable applications from inside the closure, resulting in the following type transformation:

$$\mathtt{R}\, (a_1, \ldots, a_k) : A \leftarrow A \rightsquigarrow translate*[\![(x, y) \in \mathtt{R}]\!] : A_1 \ldots A_k \times A \leftarrow A_1 \ldots A_k \times A$$

The resulting expression can then be processed by the standard PF transformation already defined, where the variables will be dropped.

The expansion of the inside expression by the $[\![\,]\!]$ operation results only in the introduction of boolean operators and existential quantifiers, which have to be removed in order to obtain the desired PF expression. The existential quantifications may be removed by the *definitions* rule, which applies the composition or n-ary composition, associated with the reverse or rotate operators, to drop those variables. In order to obtain the wanted shape of input and output, we apply a rule similar to *uniform*, but with a tuple on each side instead:

$$uniform* \triangleq many\, (once\, (a_i\, R\, a_j \rightsquigarrow (a_1, \ldots, a_n, x)\, (X_i^{n\circ} \cdot R \cdot X_j^n)\, (a_1, \ldots, a_n, y)))$$

Once again, if a_i or a_j are tuples, this transformation is generalized to forks of X projections. The rest of the boolean operators may then be removed by the *aggregate* rule already defined, by applying intersections, reunions and complements. The translation is thus defined by:

$$translate* \triangleq many(aggregate \triangleright definitions \triangleright uniform*) \quad (10)$$

5 Translating Alloy declarations to FA

Besides translating Alloy formulas to FA we also need to express in this formalism other facts about an Alloy model, namely all the information concerning signature, relation and multiplicity declaration.

Alloy signatures denote sets of atoms and thus will be represented by constant coreflexives. A sub-signature declaration induces an inclusion between its

coreflexive and the super-signature coreflexive. Sub-signatures must also be pairwise disjoint. If a signature is abstract, then its coreflexive is exactly the union of all the coreflexives of its sub-signatures. Moreover, according to Alloy semantics, the identity relation is equal to the union of all the top-level signature coreflexives.

A binary relation of type $R :: A \to B$ induces the fact $R \subseteq \Phi_B \cdot \top \cdot \Phi_A$. In case of n-ary relations, since we transform the domain into a product, the property can be generalized to $R \subseteq \Phi_{A_1} \cdot \top \cdot \Phi_{A_2} \times \ldots \Phi_{A_n}$, for a relation $R :: A_1 \to \cdots \to A_n$. Notice that type reasoning can be easily performed using this kind of declaration. For instance, since the inclusion is monotone, for two relations $R \subseteq \Phi_A \cdot \top \cdot \Phi_B$ and $S \subseteq \Phi_C \cdot \top \cdot \Phi_D$ we have $R \cdot S \subseteq \Phi_A \cdot \top \cdot \Phi_B \cdot \Phi_C \cdot \top \cdot \Phi_D$. Since for any coreflexives Φ and Ψ we have $\Phi \cdot \Psi = \Phi \cap \Psi$, we see that the values that are composed are precisely the ones belonging to $\Phi_B \cap \Phi_C$. In particular, if the signatures are disjoint we have $R \cdot S \subseteq \bot$ as expected.

Signature and relation multiplicities can be used to introduce many useful constraints in an Alloy model, without the need for additional facts. They can be directly expressed as FA properties, thus avoiding the need to be formulated in RL and processed by the default transformation. For example, a signature A constrained by multiplicity `some` induces the property $\top \subseteq \top \cdot \Phi_A \cdot \top$ involving its coreflexive. The following calculation makes evident why this formula expresses the desired meaning:

$$\begin{aligned}
\top \subseteq \top \cdot \Phi_A \cdot \top &\equiv \langle \forall x, y :: x \top y \Rightarrow x\,(\top \cdot \Phi_A \cdot \top)\,y \rangle \\
&\equiv \langle \forall x, y :: \mathit{true} \Rightarrow \langle \exists z, w :: x \top z \wedge z\,\Phi_A\,w \wedge w \top y \rangle \rangle \\
&\equiv \langle \forall x, y :: \langle \exists z, w :: \mathit{true} \wedge z\,\Phi_A\,w \wedge \mathit{true} \rangle \rangle \\
&\equiv \langle \exists z, w :: z\,\Phi_A\,w \rangle
\end{aligned}$$

A similar calculation allows us to deduce that a signature A constrained by multiplicity `lone` can be represented the property $\Phi_A \cdot \top \cdot \Phi_A \subseteq id$. If the multiplicity is `one` we just generate both facts.

Concerning relations, each multiplicity in a declaration will be treated separately and will induce a different FA property. A `lone` in the last column of a relation declaration (for example, $R : A \to \texttt{lone}\,B$) can be represented by the property $R^\circ \cdot R \subseteq id$, as justified by the following calculation:

$$\begin{aligned}
R^\circ \cdot R \subseteq id &\equiv \langle \forall x, y :: x\,(R^\circ \cdot R)\,y \Rightarrow x\,id\,y \rangle \\
&\equiv \langle \forall x, y :: \langle \exists z :: x\,R^\circ\,z \wedge z\,R\,y \rangle \Rightarrow x = y \rangle \\
&\equiv \langle \forall x, y :: \langle \exists z :: z\,R\,x \wedge z\,R\,y \rangle \Rightarrow x = y \rangle
\end{aligned}$$

If the `lone` multiplicity appears in a column other than the last, we can just apply the rotate operator in combination with the above property. In general, given a `lone` in the ith column we have the following equivalent property:

$$\overset{(|R|-i)\to^\circ}{R} \cdot \overset{(|R|-i)\to}{R} \subseteq id$$

A similar reasoning can be applied to the `some` multiplicity. If the ith field is marked with this multiplicity we generate the following property:

$$id \subseteq \overset{(|R|-i)\rightarrow}{R} \cdot \overset{(|R|-i)\rightarrow^\circ}{R}$$

Likewise to signatures, if the multiplicity is `one` we generate both facts.

6 Implementation and example

The translations defined in Sects. 4 and 5 were implemented in a tool that transforms Alloy models into FA inequations ready to be proved by Prover9. The tool is divided in three parts: first Alloy models are parsed and translated to RL; then the RL models are transformed to FA; lastly the FA model is embedded into Prover9.

All the translations were implemented in the functional language Haskell. Parsing of the Alloy model is performed by resorting to the parser generator *Happy* [13]. The model is then transformed to fit the restricted grammar defined in Appendix A. Finally, by applying the rules defined in Fig. 2 and the properties induced by the signatures defined in Sect. 5 we obtain the RL model.

In order to implement the translation from RL to FA, defined in Sect. 4, a generic RL strategic rewriter was implemented, allowing an effective way to manipulate formulas by defining rules and strategies. As such, once the needed RL rules were defined, we were able to define not only the translation from PW to PF, with several variants, but also its reverse, from PF to PW. The facts and the properties induced by the signature declarations are passed on as the environment of the other translations. Finally, although this tool was implemented with Alloy in mind, a mode for directly translating RL formulas to FA is also provided.

The output of the translation is a step-by-step RL to FA translation along with the final FA model, pretty-printed in LaTeX. In Fig. 3 the translation of the example Alloy model in Fig. 1 is presented. For space reasons, in the assertion we abbreviate signature and relation names to the first letter and denote by R_i the term $(X_i^{3\circ} \cdot R)$. This FA model can now be passed to Prover9. To prove theorems in Prover9, it is necessary to define useful valid axioms and lemmas for the particular context. In order to efficiently verify Alloy models, useful relational and FA properties were defined. In particular, the assertion in our example model was automatically verified by Prover9.

7 Related work

Little work has been done concerning unbounded verification of Alloy specifications. Arkoudas et al. [1] have developed a tool, `Prioni`, which provides the axiomatization of the RL of Alloy to Athena [17], a type-ω denotational proof language. However, this translation is made to a first-order logic, and thus the

$$id = \Phi_{Person} \cup \Phi_{Course} \cup \Phi_{University}$$
$$\Phi_{Student} \cup \Phi_{Professor} \subseteq \Phi_{Person} \wedge \Phi_{Student} \cap \Phi_{Professor} = \bot$$
$$lecturer \subseteq \Phi_{Course} \cdot \top \cdot \Phi_{Professor}$$
$$enrolled \subseteq \Phi_{University} \cdot \top \cdot \Phi_{Student}$$
$$courses \subseteq \Phi_{University} \cdot \top \cdot \Phi_{Student} \times \Phi_{Course}$$
$$depends \subseteq \Phi_{Course} \cdot \top \cdot \Phi_{Course}$$
$$id \subseteq lecturer \cdot lecturer^{\circ}$$

$$\begin{gathered}(\Phi_U \times \Phi_U \times \Phi_S) \cap c_1/(e_1 \cdot \pi_1) \cap (c_1 \cdot (\Phi_S \times d^{*\circ}))/c_1 \\ \cap \\ c_1/c_2 \cap c_2/c_1 \cap e_1/e_2 \cap e_2 \cdot \pi_2 \cdot \pi_2 \cap e_2/(e_1 \cup id_3) \\ \subseteq \\ c_2/(e_2 \cdot \pi_1) \cap (c_2 \cdot (\Phi_S \times d^{*\circ}))/c_2\end{gathered}$$

Fig. 3. Example model translated to FA

relational flavor of Alloy is lost. Frias et al. defined a translation of Alloy formulas to FA [5], from which the initial inspiration for our translation was drawn. They then extended the PVS [16] language to support FA, and used it to verify Alloy specifications [6]. Later, the same team extended the FA so that it supports first-order quantifiers, making it more friendly for Alloy users [3]. They also presented a tool, `DYNAMITE`, which provides interaction between PVS and Alloy Analyzer. A significant methodological difference to our work is that, in both these works, the focus was not on automatic verification but on manual proofs assisted by a theorem prover.

Although initially inspired by [5], our translation results in much simpler expressions and ends up having very few similarities with it. We also convert the input of the relations to a tuple containing all quantified variables, from which the desired variable is then selected. However, the way this goal is achieved differs significantly from the original. In [5] all variable occurrences were directly translated to a coreflexive which forced the input to take the value of the desired variable. Like has been presented, we expand the Alloy terms to RL, in order to be able to completely remove the variables by applying some simple FA rules. Other restrictions that highly increase the complexity of the initial translation were lifted in our translation, like imposing every formula to be of the shape $\top = R$ and enforcing that shape on the inner terms. For comparison purposes, consider the following Alloy formula:

```
all a : A | some b : B | a in R.b && a in S.b
```

Applying the translation defined in [5] results in the following formula:

$$\overline{\overline{\overline{\top \cdot \rho(\langle id, \pi_2 \cdot \phi \rangle)} \cdot \langle \pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1, \pi_2, \top \rangle} \cdot \langle \pi_1, \pi_2, \top \rangle} \cdot \langle id, \top \rangle \cdot \top = \top$$

where

$$\phi = id \times \top \cap \overline{\delta(\pi_2 \cdot \pi_1 \cap \pi_2) \cap \overline{\rho(id \times R \cdot \delta(\pi_1 \cdot \pi_1 \cap \pi_2))} \cap id \times \top} \cap \overline{\delta(\pi_2 \cdot \pi_1 \cap \pi_2) \cap \overline{\rho(id \times S \cdot \delta(\pi_1 \cdot \pi_1 \cap \pi_2))} \cap id \times \top}$$

and ρ and δ compute the domain and range of a relation, respectively (as a coreflexive). On the other hand, our translation (without the heuristic simplification) results in

$$\top \subseteq \overline{\overline{(\top \cdot (\pi_1 \cap R \cdot \pi_2) \cap \top \cdot (\pi_1 \cap S \cdot \pi_2)) \cdot \langle id, \top \rangle} \cdot \top}$$

and with the heuristics just in $id \subseteq R^\circ \cdot S$. In fact, the resulting expressions were so complex that the approach proposed in [5] was abandoned by the authors in [3]. Another improvement from their work is that we consider the whole Alloy model, including all type information from declarations, while they consider only the translation of the formulas. A feature of the approach presented in [6] from which we drew inspiration is the definition of the n-ary composition operator.

Using Prover9 to verify relational models was already proposed in [7]. However, since they use only RA, their expressiveness power is limited to a 3 variable fragment of RL. A significant difference occurs in the way the types are represented. While they use *vectors* to represent sets, and thus types, we use coreflexives. This change is motivated by our belief that coreflexives are more amenable to calculation. They also did not propose a translation from Alloy to RA and assume the model is already defined in this formalism.

8 Conclusions

We have presented a framework for unbounded verification of Alloy specifications with the automatic theorem prover Prover9. The key ingredient of this framework is a translation from Alloy models into FA, a point-free characterization of relational logic. This translation greatly improves a previously existent one, by proposing a different translation strategy and using heuristics to further simplifies the results. It also covers the structural aspects of the model, namely type declarations. These improvements made proof automation with Prover9 viable.

The translation was fully implemented and tested on several examples with complexity similar to our running example. On all these Prover9 was able to automatically verify the assertions. However, in order to scale this approach to bigger case-studies some additional work is still needed to find the appropriate set of axioms to be included in Prover9. The main problem concerns the ubiquitous use in Alloy models of relations with arity bigger than 2, and thus we are currently searching for more useful laws characterizing the operators introduced in Sect. 3.1.

Although Prover9 was used to discharge the proofs, the translation presented is generic and not bound to it. For instance, since our translation provides much simpler results than the one from [5], it could be interesting to use it as a replacement in order to increase the performance of the underlying proof system.

References

1. Arkoudas, K., Khurshid, S., Marinov, D., Rinard, M.: Integrating model checking and theorem proving for relational reasoning. In: RelMiCS 7: 7th International Seminar on Relational Methods in Computer Science. pp. 21–33 (2003)
2. Dang, H.H., Höfner, P.: First-order theorem prover evaluation w.r.t. relation- and Kleene algebra. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS10/AKA5 - PhD Programme. pp. 48–52. Universität Augsburg (2008), Technical Report
3. Frias, M., López Pombo, C., Moscato, M.: Dynamite: Alloy Analyzer+PVS in the analysis and verification of Alloy specifications (2009)
4. Frias, M.F.: Fork Algebras in Algebra, Logic and Computer Science, Advances in Logic, vol. 2. World Scientific Publishing Co., Inc. (2002)
5. Frias, M.F., Pombo, C.L., Aguirre, N.: An equational calculus for Alloy. In: 6th International Conference on Formal Engineering Methods. LNCS, vol. 3308, pp. 162–175 (2004)
6. Frias, M.F., Pombo, C.L., Moscato, M.M.: Alloy Analyzer + PVS in the analysis and verification of Alloy specifications. In: TACAS. LNCS, vol. 4424, pp. 587–601. Springer-Verlag (2007)
7. Höfner, P., Struth, G.: On automating the calculus of relations. In: Proceedings of Automated Reasoning, 4th International Joint Conference, IJCAR 2008. LNCS, vol. 5195, pp. 50–66. Springer-Verlag (2008)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
9. Jackson, D., Wing, J.: Lightweight formal methods. IEEE Computer 29(4), 21–22 (1996)
10. Lämmel, R., Visser, J.: Typed combinators for generic traversal. In: Proceedings of the Practical Aspects of Declarative Programming PADL 2002. LNCS, vol. 2257, p. 137154. Springer-Verlag (2002)
11. Macedo, N.: Translating Alloy specifications to the point-free style. Master's thesis, Universidade do Minho (2010)
12. Maddux, R.D.: Relation Algebras, Studies in Logic and the Foundations of Mathematics, vol. 150. Elsevier (2006)
13. Marlow, S., Gill, A.: Happy User Guide (2001)
14. McCune, W.: Prover9 and Mace4 (November 2009), http://www.cs.unm.edu/mccune/prover9
15. Oliveira, J.N.: Extended static checking by calculation using the point-free transform. In: LerNet ALFA Summer School 2008. LNCS, vol. 5520, pp. 195–251. Springer-Verlag (2009)
16. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: 11th International Conference on Automated Deduction. Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer-Verlag (1992)
17. Pombo, C.L., Owre, S., Shankar, N.: Type-ω DPLs. Tech. Rep. AIM-2001-027, MIT (2001)

A Grammar for a fragment of Alloy

```
Model     := Paragraph*
Paragraph := Sig | Fact | Assert
```

```
Sig       := [Mult] [abstract] sig Id [extends Id] {Rel*}
Fact      := fact {Form}
Assert    := assert {Form}
Rel       := [Mult] Id | [Mult] Id -> Rel
Mult      := some | one | lone | set
Form      := Exp in Exp
           | Form && Form
           | ! Form
           | all Id : Exp | Form
           | some Exp | lone Exp
Exp       := Id                -- relation identifier
           | iden              -- identity relation
           | none              -- empty relation
           | univ              -- universe
           | ~ Exp             -- transposition
           | Exp . Exp         -- composition
           | Exp & Exp         -- intersection
           | Exp + Exp         -- union
           | Exp - Exp         -- difference
           | Exp -> Exp        -- cartesian product
           | Exp <: Exp        -- domain restriction
           | Exp :> Exp        -- range restriction
           | * Exp             -- reflexive-transitive closure
```