HAL
open science

A First Step in the Translation of Alloy to Coq

Salwa Souaf, Frédéric Loulergue

To cite this version:

Salwa Souaf, Frédéric Loulergue. A First Step in the Translation of Alloy to Coq. 21st International Conference on Formal Engineering Methods (ICFEM), 2019, Shenzen, China. 10.1007/978-3-030-32409-4_28 . hal-02317118

HAL Id: hal-02317118

https://hal.science/hal-02317118

Submitted on 15 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A First Step in the Translation of Alloy to Coq

Salwa Souaf[1,2] and Frédéric Loulergue[2]

[1] School of Informatics Computing and Cyber Systems
Northern Arizona University, USA
firsname.lastname@nau.edu
[2] INSA Centre Val de Loire, LIFO EA 4022, Bourges, France
salwa.souaf@insa-cvl.fr

Abstract. Alloy is both a formal language and a tool for software modeling. The language is basically first order relational logic. The analyzer is based on instance finding: it tries to refute assertions and if it succeeds it reports a counterexample. It works by translating Alloy models and instance finding into SAT problems. If no instance is found it does not mean the assertion is satisfied. Alloy relies on the small scope hypothesis: examining all small cases is likely to produce interesting counterexamples. This is very valuable when developing a system. However, Alloy cannot show their absence. In this paper, we propose an approach where Alloy can be used as a first step, and then using a tool we develop, Alloy models can be translated to Coq code to be proved correct interactively.

Keywords: first order relational logic, calculus of inductive construction, translation.

1 Introduction

There are many different formal methods, ranging from completely automated tools, for e.g. static analyzers and sanitizers [24], to interactive theorem proving that requires a lot of human work.

Often, the users of such tools need to provide a specification of the analyzed system. Analyzing this specification can then be automatic or interactive. Alloy and the Alloy analyzer [10] fall into the first category. Alloy was and is used in many different domains. For example software engineering [7], and security [20]. More specific applications of it, as presented by Torlak et al. in [28], are modeling and analysis of software systems, bounded program verification and test-case generation. Multiple systems have been studied using Alloy: the flash file system [12,13], the Mondex electronics purse [21], a proton therapy machine [22], an information system library [6], *etc.*

When it comes to bounded program verification two related works were presented in detail in [28]. The Jalloy tool [11] checks a Java method against a specification of its behavior. It starts by translating the method to Alloy then invoking an early prototype of the Alloy Analyzer on the resulting constraints. The second work was built on the previous work and is called Forge [4]. It employed a new translation from procedural code to relational logic involving symbolic execution, using the KodKod API [27]. Alloy have also been exploited in

many tools for test-case generation, to mention: TestEra [16] and Whispec [23]. While TestEra [16] employs Alloy in a specification-based black-box framework for testing of Java programs, Whispec [23] is an approach for specification-based white-box testing using Kodkod. KodKod [5], that is at the heart of Alloy's engine is also used in Niptick [2] a counter-example finder for the proof assistant Isabelle.

Alloy is a lightweight formal method as it relies on the small scope hypothesis: examining all small cases is likely to produce interesting counterexamples. However, the Alloy analyzer cannot show the absence of errors. Other formal tools such as the interactive theorem provers Coq [26] and Isabelle [18] have been used to provide very strong guarantees on verified software, including a C compiler [15] and the kernel of an operating system [14].

We think it is very valuable to use lightweight formal methods. In practice, if one is to use a tool such as Alloy as a first step, then wants to use a more heavyweight tools such as Coq as a second step, the formalization done first is lost. To support the transition from Alloy to Coq, we propose a translator from Alloy models to Coq code.

The paper is organized as follows. In Section 2 we briefly present Alloy and Coq. The principles of the tool we propose are described in Section 3, including examples of translation. We compare our approach to related work in Section 4, discuss the current limitations of our tool in Section 5, and conclude in Section 6.

2 An Overview of Alloy and Coq

2.1 Alloy and the Alloy Analyzer

Alloy [9] is both a language and an analyzer for writing and checking formal models. This section provides the details of the properties and main components of this language.

Alloy Properties Alloy have been widely used for modeling systems in order to simulate them and verify their properties. It allows a simplified view of the systems by abstracting implementation details and focusing on their properties and constraints. The language has a simple syntax based on the Z language. It is a structural language: it allows to model complex structures with hierarchies and relations. Although it offers the possibility to define entities with properties and constraints to describe systems, it does not conduct treatments. Alloy is an analyzable language. The properties of an Alloy model can be checked and simulated using the Alloy Analyzer.

Atoms & Relations In Alloy, atoms are the basic elementary entities. It is an abstract concept that is used to model aspects of the real world. Alloy data types are universally based on relations. They represent a concept that serves to define correlations between atoms. Relations and atoms cooperate to represent different aspects of a system. Relations can have an n arity and can be declared as $f : A_1 \rightarrow \ldots \rightarrow A_n$.

Signatures represent the entities of a system. A signature is the only element to represent the types and atoms in an Alloy model. Although it is a non-object-oriented language, Alloy allows inheritance between signatures. A signature can have attributes as explained below.

Facts in Alloy are used to describe different constraints about the system being modeled that remain always true. In Alloy, all facts are defined using the keyword *fact*.

Predicates are an abstraction of logical formulas for reuse purposes. A predicate can be defined with parameters used in the logical formula of its body. Predicates are often used in assertions that we want to verify on the model.

Functions return typed values for reuse and model clarity sake.

Assertions are used to specify properties about the model that we expect to hold or that we want to check if they hold. Once an assertion is stated we can check if it holds in a specific scope, using the keyword **check** and feeding the model to the Alloy Analyzer. The analyzer looks for a counterexample to the assertion within the specified scope.

The scope is the cardinality, specified by the user, of the top level signatures in a model. Although working within limited scopes ensures that the model-finding problem is decidable, it limits the generality of the results produced by the Alloy Analyzer. Jackson explains this design decision through the *small scope hypothesis*: most bugs can be found by testing programs for all test inputs within a small scope. For more details refer to [10, section 5].

We discuss more specific Alloy syntax and semantics on the example of Figure 1 that is basically the example of [10, page 16]. We will use this example as running example throughout the paper. The interested reader can refer to [10] for a longer discussion of this example.

Name and Addr are two *signatures* in Alloy terminology. They are sets. Book is also a signature containing an attribute, addr. While addr is given type Name →Addr, the fact that it is an attribute of Book means it is actually a ternary relation between Book, Name and Addr. In lines 3 and 4 of Figure 1 we can see the definition of the predicates add and del both defining two different states of book, the first by adding a new entry (i.e. addr) and the second by deleting an existing one. In this code, + means union, − set difference, and . is the relational join of Alloy. One specificity of the join operation in Alloy is that in an expression r1.r2, the right-most column of relation r1 and the left-most column of relation r2 are not in the join result. The function lookup returns the Addr associated to the Name n in the book b, n and b given as arguments of the function.

We can see how assertions are defined for this example in lines 11–21. The assertion delUndoesAdd is stating that by adding an entry to a book then deleting it we go back to the initial state of the book (taking into consideration that these are the only two operations done on the book). In order to check if this assertion holds, we execute the **check** stated in line 22 using the Alloy Analyzer (the scope in this case is 5 atoms, if the scope is not specified it is set to 3).

```
sig Name, Addr { }
sig Book { addr: Name → Addr }
pred add [b, b': Book, n: Name, a: Addr] { b'.addr = b.addr + n→a }
pred del [b, b': Book, n: Name] { b'.addr = b.addr − n→Addr }
fun lookup [b: Book, n: Name] : set Addr { n.(b.addr) }
assert delUndoesAdd {
  all b, b', b'': Book, n: Name, a: Addr |
    no n.(b.addr) and add [b, b', n, a] and del [b', b'', n]
    implies b.addr = b''.addr
}
assert addIdempotent {
  all b, b', b'': Book, n: Name, a: Addr |
    add [b, b', n, a] and add [b, b'', n, a]
    implies b'.addr = b''.addr
}
assert addLocal {
 all b, b': Book, n, n': Name, a: Addr |
   add [b, b', n] and n != n'
   implies
   lookup [b, n'] = lookup [b', n']
}
check delUndoesAdd for 5
```

Fig. 1. Alloy Example

2.2 The Coq Proof Assistant

The Coq proof assistant is based on the calculus of inductive constructions [19], a higher-order typed λ-calculus. Coq and the calculus of inductive constructions are based on the Curry-Howard correspondence: a type corresponds to the statement of a theorem, and a program to the proof of a theorem.

The core of Coq is very small. For example there is no pre-defined data type. All definitions are typed in Coq. Therefore a user-defined type has a type, named a sort. There are three sorts in Coq: **Set** is the sort of types that correspond to types found is usual programming language. It is the sort of the "computational" types. **Prop** is the sort of "logical" types. Both **Set** and **Prop** are typed: their type is **Type**. Most of the time when using Coq, the type of **Type** will be displayed as **Type**. Actually there is a countable infinity of sorts **Type**.

In Gallina, the language of Coq, a definition contains three components: a name, a type, and a term. For example the polymorphic identity function can be defined as shown in lines 1–2 of Figure 2.

As the core does not contain predefined types (but the sorts **Set**, **Prop** and **Type**), Coq provides a mechanism to define new inductive types. This is done by giving a list of *constructors* for values of the defined type. For example, Peano natural numbers are defined in lines 4–6 of Figure 2. There are two constructors for values of type nat: O and S the latter taking a nat as argument.

```
1  Definition id: ∀ (A:Type), A → A :=
2    fun A x ⇒ x.
3
4  Inductive nat : Set :=
5  | O : nat
6  | S : nat → nat.
7
8  Fixpoint add (n1 n2:nat) : nat :=
9    match n1 with
10   | O ⇒ n2
11   | S n1 ⇒ S(add n1 n2)
12   end.
13
14 Lemma add_n_O: ∀ n,
15   add n O = n.
16 Proof.
17   induction n as [ | n IH ].
18   − trivial.
19   − simpl. rewrite IH. trivial.
20 Qed.
```

Fig. 2. Coq Example

Functions are most often written using pattern matching as in lines 8–12 of Figure 2. For each possible way of constructing a value of the type of the matched expression (in this case n1 of type nat), the pattern matching construct returns (after ⇒) a specific result. The patterns (on the left-hand side of ⇒) may contain variables: in case the matching succeeds, the free variables are bound to the matched values in the right-hand side of ⇒. Note that add is a recursive function (**Fixpoint** keyword). Only terminating functions are allowed in Coq: in this case the system checks the termination by checking that the recursive call is done on a strict syntactic subterm of n1.

Coq is a proof assistant: it is possible to define theorems and prove them. As mentioned at the beginning of this section, a Coq definition contains three elements: a name, a type and a term. In the case of a theorem (or lemma, proposition, *etc.*), the term (i.e. the proof) is usually not written as a program (even though the Curry-Howard correspondence states a program and a proof are the same thing): the proof script language of Coq is used instead. In the code of Figure 2, add_n_O is the name of the lemma, ∀ n, add n O = n is its type, and the proof script between **Proof** and **Qed** builds a term that is the proof of the lemma.

One important feature of Coq is that computational terms can be embedded into types. For example the library Vector of Coq standard library contains the following inductive type definition:

```
Inductive t (A : Type) : nat → Type :=
| nil : t A 0    | cons : ∀ (h:A) (n:nat), t A n → t A (S n).
```

The size of a value of this type contains the length of the vector. For example, a value of type Vector.t nat 10 is a vector containing ten nat values. Vector.t is called a *dependent type*.

This feature can also be used to define predicates as inductive types. For example the < predicates on Peano natural numbers is defined in the Coq standard library as:

```
Inductive le (n : nat) : nat → Prop :=
| le_n : le n n   | le_S : ∀ m : nat, le n m → le n (S m).
```

More generally, Coq functions can take both computational values and types as arguments, and also return them as results. As values of some types (like add_n_O) are proofs, Coq functions can also take proofs as arguments and return proofs as results. We use these features in the Coq code generated from Alloy models.

It is also possible to *declare* values in Coq: in this case we have only a name and a type. In the case of a value that needs a proof, it means an axiom is introduced in Coq's logic. Note that when such declarations can be written inside a section, in such a way that at the closing of the section, all the elements that depend on these hypotheses are added additional arguments corresponding to these hypotheses.

3 The Transformation

3.1 Basic Principles

Logical Quantifiers and Connectors Logical elements present in the Alloy language, are also present in Gallina, either as primitives (universal quantification) or defined in the standard library (existential quantification, negation, conjuction, disjunction). The design choices thus appear when translating the relational parts of Alloy.

Sets, Relations and Elements In the Coq standard library, sets and binary relations are formalized using predicates. Given a type A, a subset of A is formalized as a predicate on A, i.e. a value of type A→**Prop**, and a binary relation on types A and B as a value of type A→B→**Prop**. We could use directly such a formalization, and consider higher arities: the simple example of Figure 1 indeed contains a relation of arity 3. Some other translation tools from Alloy to provers (discussed in Section 4) have explicit different translations for sets, binary relations, ternary translations, *etc.* Some of them are limited to a given arity.

However, in addition to a "raw" translation from Alloy to Coq, we wanted our tool to provide some support to ease the proof in Coq of the assertions of an Alloy model. Such a support includes general lemmas about the properties of the set and relational operations of Alloy. While of course possible in Coq, we chose to avoid such a solution as it would mean we would have to generate as many versions of the operations as there are combination of the arities, and as many supporting lemmas as there are combinations of these operations. Also in Alloy, relational operations can be applied to elements that are seen as singleton sets.

Therefore we chose to generalize the approach present in the Coq standard library: considering a type U (the universe of Alloy), a relation of arity n (with $0 < n$) is formalized as a value of type U→...→U→**Prop** that contains n U.

To be able to define operations on arbitrary relations, we first need to express the arity of a relation. This is done by the following definition:

```
Fixpoint arity (n : nat): Type :=
  match n with
    | 0 ⇒ Prop
    | S n' ⇒ U → arity n'
  end.
```

Therefore arity 1 simplifies to U→**Prop**, arity 2 to U→U→**Prop**, *etc.* With this definition we are able to translate any Alloy signature into a set of declarations of Coq values whose types are declared using arity.

To model an element as a singleton set, we define a Singleton predicate:

```
Fixpoint Singleton n (R: arity (S n)) : Prop :=
  match n with
  | 0 ⇒ ∃! (x:U), R x
  | S n' ⇒ ∃! (x:U), Singleton n' (R x)
  end.
```

Basically what this predicate does is that for a relation R of arity n greater than 1, it indicates there exists a unique element x of U such that the partial application R x is also a singleton relation. For a relation of arity 1, it just states that there exists a unique x such that R x.

Unfortunately the code above is not accepted by Coq. The problem is that Coq cannot determine without additional information that R x can be considered as a value of type arity n. To help the system we need "cast" functions (Figure 3). Note that both these functions are defined using the proof script language of Coq. However, these cast functions are not enough: we need to provide them a proof as their last argument. This proof is simple, that is actually a proof by reflexivity, and we can use what Chlipala calls the "convoy pattern" [3, page 172] to get these proofs in the right hand sides of the pattern matching construction.

```
Definition cast n1 (R1 : arity n1) (H: n1 = 0) : Prop.
  subst. simpl in ∗. trivial.
Defined.
Definition cast' n1 n1' (R1 : arity n1) (H: n1 = S n1') : arity (S n1').
  subst. simpl in ∗. trivial.
Defined.
Fixpoint Singleton n (R: arity (S n)) : Prop:=
  match n as m return n = m → Prop with
  | 0 ⇒ fun H ⇒ ∃! x, cast _ (R x) H
  | S n' ⇒ fun H ⇒ ∃! y, Singleton n' (cast' _ _ (R y) H)
  end eq_refl.
```

Fig. 3. Actual Definition of Singleton

This small example shows that while having generic arity relations is indeed very generic, it makes the formalization more technically challenging. However, by providing general theorems on the Coq formalization of Alloy operations, we

think the user of our tool will not have to deal with such technicalities most of the time.

Operations All the basic relational operations have the same shape as Singleton. For example the inclusion operator **in** of Alloy is translated as (the cast and convoy pattern are omitted):

```
Fixpoint IN n (R1: arity n)(R2: arity n): Prop :=
  match n with
   | 0 ⇒ R1 → R2
   | S n' ⇒ ∀ (x:U), IN n' (R1 x) (R2 x)
  end.
```

Basically it means that for all n-tuple t, if R1 t then R2 t. The Alloy equality is not translated as the default syntactic equality (up to reduction) of Coq, but as:

```
Definition EQUAL n (R1: arity n)(R2: arity n): Prop :=
  (IN R1 R2) ∧  (IN R2 R1).
```

Note that all the first nat arguments of these definitions are made implicit. It is therefore not necessary to give them explicitly when using these definitions: Coq infers them. Also instead of writing EQUAL a b, we use Coq's notations a == b.

Slightly more challenging operations are the join and the product. Again omitting the casts and the convoy pattern, the Alloy join operation is defined as shown in Figure 4.

```
Fixpoint JOIN_R n2 (R1: arity 1)(R2: arity (S n2)) : arity n2 :=
  match n2 with
  | 0 ⇒ ∃ x:U, (R1 x) ∧  (R2 x)
  | S n2' ⇒ fun (y:U) ⇒ JOIN_R n2' R1 (fun (x:U) ⇒ R2 x y)
  end.
Fixpoint JOIN n1 n2 (R1: arity (S n1)) (R2: arity(S n2)) : arity(n1+n2) :=
  match n1 with
  | 0 ⇒ JOIN_R n2 R1 R2
  | S n1' ⇒ fun (y:U) ⇒ JOIN n1' n2 (R1 y) R2
  end.
```

Fig. 4. Definition of Join (Details Omitted)

Operation Properties As mentioned before, in addition to translate the definitions, operations, formulas of Alloy, we also provide properties of Alloy operations. The first set of properties concerns the Alloy equality ==: we proved it is an equivalence relation and also that it is compatible with the Alloy operations, i.e. for an operation f, if for all a, b such that a == b, then f a == f b. This allows to use the rewriting tactics of Coq while writing proofs. These are very important as most of the other properties are stated as equalities using ==.

The second set of properties are mostly algebraic properties. For example we have:

```
Lemma UNION_idem: ∀ n (R: arity n), UNION R R == R.
```

We developed a tactic that is able to prove most of these properties, the proof script in this case is **Proof**. solve_alloy. **Qed**.

Other properties are more specific to Alloy operations. For example we provide a lemma that states that if the join of a binary relation with itself contains the relation, then this relation is transitive:

```
Lemma JOIN_IN_transitive : ∀ R: arity 2,
  IN (JOIN R R) R ↔ (∀ x y z, R x y → R y z → R x z).
```

3.2 Alloy Models Translation

Now that we have translated the basic elements of the Alloy language, we use them to translate Alloy models. Here we present how each of the Alloy models components is translated into Coq syntax and the reasoning behind it. We continue using the example given in Figure 1.

Signatures As we presented so far, everything that is going to be in our Coq translation of the Alloy models should be of type arity n. In order to follow this reasoning and to be able to manipulate Alloy signatures, we have decided to represent them in the format of Coq Variables (declarations) by specifying first their arity. Top-level signatures like Name, Addr and Book are sets and thus unary (i.e. arity 1) relations. Signature attributes are declared as relations (arity greater than 1) then a Hypothesis is added to the Coq code for their types, lines 2 and 3 in the following Coq translation shows the example of attribute addr:

```
1 Variable Name Addr Book: arity 1.
2 Variable addr: arity 3.
3 Hypothesis addr_sig: IN addr (PRODUCT Book (PRODUCT Name Addr)).
```

Facts A way to declaring facts about a system in Coq is by stating Hypothesis. Thus, Alloy model facts are translated in our tool to Hypothesis and the syntax is as follows:

```
Hypothesis Model_fact: translated_fact_formula.
```

Functions and Predicates Both are transformed in the same way to Coq syntax. For reasons of re-usability and ease of application we have decided to transform them into Coq inductive type definitions. The following examples are transformation of the del predicate and lookup function presented in Figure 1. When writing the constructor for the inductive type, we start by modeling the "types" of the arguments as inclusions, possibly with additional expressions for modeling the cardinality. In the example of del, the argument b has type Book thus In b Book, but also b is an element, thus ONE b. We formalize functions as predicates, but with an additional argument that models the result returned by the function. In the case of lookup, the result is the value r_lookup:

```
Inductive del: arity 1 → arity 1 → arity 1 →  Prop:=
| del_def: ∀ (b: arity  1) (b': arity  1)  (n: arity  1),
    IN b Book ∧  (ONE b) →
    IN b' Book ∧  (ONE b') →
    IN n Name ∧  (ONE n) →
    JOIN b' addr == DIFFERENCE (JOIN b addr) (PRODUCT n Addr) →
    del b b' n.

Inductive lookup: arity 1 → arity 1 → arity 1 → Prop:=
| lookup_def: ∀ (r_lookup: arity 1) (b: arity  1)  (n: arity  1),
    IN r_lookup Addr →
    IN b Book ∧  (ONE b) →
    IN n Name ∧  (ONE n) →
    r_lookup == JOIN n (JOIN b addr) →
    lookup b n  r_lookup .
```

Assertions are defined in Coq syntax and then stated as Lemmas when called in an Alloy **check** block. Thus, the assertion delUndoesAdd is transformed as follows:

```
Definition delUndoesAdd:=
  ∀ (b: arity  1) (b': arity  1) (b'': arity  1) (n: arity  1)(a: arity  1),
  ( NO (JOIN n (JOIN b addr)) ∧  add b b' n a ∧  del b' b'' n ) →
  JOIN b addr == JOIN b'' addr.

Lemma delUndoesAdd_Lemma: delUndoesAdd.
```

3.3 The Address Book Example

```
Definition addIdempotent:=
  ∀ (b b' b'' a n: arity  1),
  (add_ b b' n a  ∧  add_ b' b'' n a ) →
  JOIN b' addr == JOIN b'' addr.
```

```
Definition addLocal:=
  ∀ (b b b' a n n': arity  1) r_1 r_2,
  lookup b n' r_1 →
  lookup b' n' r_2 →
  (add_ b b' n a  ∧  not(n == n')) →
  r_1 == r_2 .
```

Fig. 5. Translation of the Assertions addIdempotent and addLocal

In the previous subsections, we presented most of the translation of the Alloy example of Figure 1. Figures 5–7 present the automatic translation using our tool of the two other assertions addIdempotent and addLocal, as well as the proof scripts we wrote to prove two of the corresponding lemmas.

A recommended style in Coq, is to avoid using explicitly automatically generated names by tactics. Our destruct_and tactics, that basically systematically replaces hypotheses of the form A∧ B by two hypotheses A and B, automatically generates names for these new hypotheses. The inversion tactic also automatically generates names. To explicitly give names to the hypotheses we want to

```
Lemma delUndoesAdd_Lemma : delUndoesAdd.
Proof.
  unfold delUndoesAdd.
  intros b b' b'' n a H. destruct_and.
  assert(Hadd: add_ b b' n a) by trivial.
  assert(Hdel: del b' b'' n) by trivial.
  inversion Hadd; inversion Hdel; subst.
  destruct_and.
  (* We are ready to prove: JOIN b addr == JOIN b'' addr *)
  assert(Hr1: JOIN b'' addr ==  DIFFERENCE (JOIN b' addr) (PRODUCT n Addr)) by trivial.
  assert(Hr2: JOIN b' addr == UNION (JOIN b addr) (PRODUCT n a)) by trivial.
  rewrite Hr1, Hr2.
  rewrite UNION_DIFFERENCE_distr_l with (R1:=JOIN b addr).
  rewrite UNION_NO_l by
      (apply DIFFERENCE_IN_NO;
       apply PRODUCT_IN_compat with (R1:=n);
       auto using IN_refl).
  rewrite DIFFERENCE_NO_INTERSECT by
      (assert(HH: NO (JOIN n (JOIN b addr))) by trivial;
       castsimpl; intros;
       specialize(HH x);
       contradict HH;
       intuition eauto).
  reflexivity.
Qed.
```

Fig. 6. Proof of Lemma delUndoesAdd

```
Lemma addIdempotent_Lemma: addIdempotent.
Proof.
  unfold addIdempotent.
  intros b b' b'' n a H. destruct_and.
  assert(Hadd1: add_ b b' n a) by trivial.
  assert(Hadd2: add_ b' b'' n a) by trivial.
  inversion Hadd1; inversion Hadd2; subst.
  assert(Hr1: JOIN b'' addr == UNION (JOIN b' addr)(PRODUCT n a)) by trivial.
  assert(Hr2: JOIN b' addr == UNION (JOIN b addr)(PRODUCT n a)) by trivial.
  rewrite Hr1, Hr2.
  rewrite ← UNION_assoc, UNION_idem.
  reflexivity.
Qed.
```

Fig. 7. Proof of Lemma addIdempotent

manipulate explicitly, we use the assert tactic of Coq that is used to prove an intermediate result. In our case, we just state and give an explicit name for already existing hypotheses, hence the use of the trivial tactic to prove the assertion (for e.g. lines 10–11 of Figure 6). To get the formulas corresponding to the definition of an Alloy predicate, or an Alloy function, the inversion tactic of Coq is needed (e.g. line 7 of Figure 6 and line 7 of Figure 7). Using the assert tactic, we give explicit names to the hypotheses generated by inversion (for e.g., lines 8–9 of Figure 7).

The two other main characteristics of these proof scripts are:

- The use of the rewrite tactic, that relies on the proofs of == is an equivalence relation, and the Alloy operations are compatible with this equivalence relation (e.g. line 13 of Figure 6 and line 10 of Figure 7).
- The systematic use of properties proved on Alloy operations: for example the distributivity of the union over the difference (line 15 of Figure 6) and the associativity and idempotence of the union (line 11 of Figure 7).

Most of the proof scripts are based on the element described above. The exception are lines 20–23 of Figure 6. The proof of the condition of the lemma DIFFERENCE_NO_INTERSECT is in a way more "low-level" than the other parts of the proof scripts as it directly makes use of the definitions of some Alloy operations. One non standard Coq tactic is castsimpl: it is a tactic we provide, and that simplifies the application of the Alloy operations and also removes all the casts in the hypotheses and the goal. In the example the goal before calling castsimpl is:

```
NO (INTERSECT (JOIN b addr) (PRODUCT n Addr))
```

meaning we have to prove that the intersection of JOIN b addr and PRODUCT n Addr is empty, while after it is:

```
∀ y x : U, ∼ ((∃ x0 : U, b x0 ∧  addr x0 y x) ∧  n y ∧  Addr  x)
```

As castsimpl simplifies the hypothesis HH in a similar way, it is quite easy to finish the proof.

These two proof scripts show that while most of the time the user can rely on proofs by rewrite and application of operation properties, when it is not possible, the proof writing remains accessible. With these two proofs, we guarantee that the Alloy assertions hold for arbitrary sets and relations Book, Name, Addr and addr.

The Tool The tool is written in Java and relies on ANTLR for parsing. There are about 2 KLoC of non-generated Java code, and the Coq supporting library Alloy is about 600 LoC. The tool and the complete examples are available at:

`https://alloy2coq.github.io`.

4 Related Work

Although theorem provers have proved their effectiveness in proving detailed properties of multiple complex system specifications, they are still considered

to be too expensive to use frequently during software development. Lightweight formal methods, on the other hand, are frequently used for checking software during design and implementation stages. Alloy, is a popular language and tool used for checking software systems against their requirements. On one hand, one of Alloy's strong suits is the counterexample returned in case of unfulfilled requirements. On the other hand, lack of counterexample, generally, does not give a correctness proof. Thus, for critical systems, a second round of analysis might be crucial. Several previous works have addressed the verification of Alloy specifications.

In [1], Arkoudas et al. present a tool, Prioni, that integrates model checking and theorem proving for relational reasoning. Prioni takes as input formulas written in Alloy. It first uses the Alloy Analyzer to check their validity for a given scope. Once no counterexample is found, Prioni translates these Alloy formulas into Athena, a denotational proof language, proof obligations and uses the Athena tool for proof discovery and checking. Unlike Prioni, that only analyzes finite domains due to the fact that Athena cannot handle infinite sets, our proposed solution handles infinite domains. Another solution that works on infinite domains is presented in [29]. Kelloy [29] is a tool for verifying Alloy specifications with respect to potentially infinite domains.

Kelloy is an engine for verifying Alloy specifications aiming to bridge the gap between lightweight formal methods and theorem provers. It provides: a fully automatic translation of Alloy language to KFOL (the first-order logic of KeY, the deductive theorem prover used in Kelloy), an Alloy-specific extension to KeY's calculus and a reasoning strategy that improves KeY's capability in finding proofs and generates intermediate proof obligations that are easy to understand.

Unlike Prioni and the transformation tool we are presenting, Kelloy was developed in a way that only takes into consideration translation of Alloy relations up to ternary relations (i.e. arity 3). Such an approach requires to define the Alloy operations for all the different combinations of the arities in KFOL.

Mariano et al. [17] followed an approach closer to ours. They present an extension of PVS (Prototype Verification System), called Dynamite, that embeds Alloy calculus. It automatically adds and analyzes new hypotheses with the aid of the Alloy Analyzer. The generated PVS sequents get cluttered with some unnecessary formulas, thus, Alloy unsat-core extraction feature is used in order to refine proof sequents. Although both our work and that presented in [17] relies on users conducting proof manually, we provide a library with predefined lemmas to provide assistance in the proof process.

5 Discussion

The tool presented in this article shows the potential of translating and proving the correctness of critical Alloy models, but it still has some limitations in its current state.

The first limitation is the subset of the Alloy language that is supported. There is one aspect that the current translation does not handle: the cardinality of sets and relations. The design choice we made is not incompatible with dealing with cardinalities. It however requires additional hypotheses. First the universe U should be countable: this is actually in line with what is considered in Alloy, but it is not set as an hypothesis in our current Coq modeling. Then to compute the cardinality (the # operator in Alloy), the argument should be a finite relation: we also plan to add this hypothesis each time the operator is used. Other Alloy features that we have yet to integrate into our tool are: integer support, Coq can handle integer definition and thus, adding this to our solution will only require some formalization efforts. The other feature that we need to improve farther is the arrow operation. For now, our arrow operation is by default a many to many arrow operation, while Alloy's arrow operation handles different multiplicities.

The second limitation is not related to the translation itself, but rather to the support provided to the user in the translated Coq code. Although we do provide a few Coq tactics to ease the work to prove what are assertions in Alloy, currently the proofs are written mostly manually by the users. We plan to enrich the Coq Alloy library with more powerful tactics.

In translating one formal language to another one, the question of the correctness of the translation arises. One possibility would be to have a Coq representation of Alloy's abstract syntax, and then give a Coq semantics to this syntax: this would be a formalization of Alloy in Coq. Then we could implement in Coq what is currently the back-end of our translation in Java: the generation of Coq code from Alloy's syntax. Proving the correctness of the translation would then mean check that the semantics and the translation are equivalent. However, it is very likely that the semantics could be given using the same basic constructs we use for our translation: they would be essentially no difference between the Coq *semantics* of Alloy, and the Coq *translation* of Alloy. Another possibility would be to have a deep embedding of both Alloy and Coq in Coq, and check that the translation (from syntax to syntax) preserves the semantics. However, our current formalization of Alloy in Coq uses features that formalizations of Coq in Coq (for e.g. [8]) do not currently handle.

6 Conclusion and Future Work

In this paper we presented a tool for translating Alloy models into Coq code. Alloy main objects are relations: sets are unary relations, elements are considered as singleton sets. We chose to keep this view in Coq and to consider, as in the module Relation_Definitions of Coq's standard library, that a relation is a function to **Prop**. This module however, only considers binary relations, therefore they have type U→U→**Prop** where U is the type of the universe.

We decided to generalize this approach. This choice required us to use dependent types everywhere in the Coq library that provides the primitive relational operations of Alloy and supports the translation. We use our tool on examples

and prove with Coq the lemmas generated by the translation: this choice of Coq formalization seems appropriate.

One of the motivations for this tool is our project around a broker for the Cloud that takes into account user security requirements that can be expressed as first order relational logic formulas and that we checked using Alloy/Kodkod [25]. In order to increase the trust in this broker, we aim at formalizing all the hypothesis made on the system, and make sure that if the formal requirement given by the user contains no error and are added to the system, then conclusions about the security of the new state of the system can be drawn. This case study will require a significantly larger translation and Coq proofs than the examples we considered so far.

References

1. Arkoudas, K., Khurshid, S., Marinov, D., Rinard, M.: Integrating model checking and theorem proving for relational reasoning. In: Berghammer, R., Möller, B., Struth, G. (eds.) Relational and Kleene-Algebraic Methods in Computer Science (RAMICS). pp. 21–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
2. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Interactive Theorem Proving (ITP). LNCS, vol. 6172, pp. 131–146. Springer (2010)
3. Chlipala, A.: Certified Programming with Dependent Types. MIT Press (2014)
4. Dennis, G.D.: A relational framework for bounded program verification. Ph.D. thesis, Massachusetts Institute of Technology (2009)
5. Emina, T., Daniel, J.: Kodkod: A Relational Model Finder. In: Orna, G., Michael, H. (eds.) Tools and Algorithms for the Construction and Analysis of Systems: (TACAS). pp. 632–647. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
6. Frappier, M., Fraikin, B., Chossart, R., Chane-Yack-Fa, R., Ouenzar, M.: Comparison of model checking tools for information systems. In: Dong, J.S., Zhu, H. (eds.) Formal Methods and Software Engineering. pp. 581–596. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
7. Geri, G., Indrakshi, R., Kyriakos, A., Behzad, B., Manachai, T., Siv, H.H.: An aspect-oriented methodology for designing secure applications. Information and Software Technology 51(5), 846 – 864 (2009)
8. Glondu, S.: Towards certification of the extraction of Coq. Theses, Université Paris Diderot (Jun 2012)
9. Jackson, D.: Automating First-order Relational Logic. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). pp. 130–139. ACM, New York, NY, USA (2000)
10. Jackson, D.: Software Abstractions. MIT Press, revised edn. (2012)
11. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 14–25. ISSTA '00, ACM, New York, NY, USA (2000)
12. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in alloy. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) Abstract State Machines, B and Z. pp. 294–308. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
13. Kang, E., Jackson, D.O.: Designing and analyzing a flash file system with alloy. Int. J. Software and Informatics 3, 129–148 (2009)

14. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Commun. ACM 53(6), 107–115 (2010)
15. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
16. Marinov, D., Khurshid, S.: Testera: a novel framework for automated testing of java programs. In: Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001). pp. 22–31 (Nov 2001)
17. Moscato, M.M., Pombo, C.L., Frias, M.F.: Dynamite: A tool for the verification of alloy models based on pvs. ACM Trans. Softw. Eng. Methodol. 23, 20:1–20:37 (2014)
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer (2002)
19. Paulin-Mohring, C.: Introduction to the calculus of inductive constructions. In: Woltzenlogel Paleo, B., Delahaye, D. (eds.) All about Proofs, Proofs for All. Studies in Logic (Mathematical logic and foundations), vol. 55. College Publications (2015)
20. Power, D., Slaymaker, M., Simpson, A.: Automatic conformance checking of role-based access control policies via Alloy. In: Engineering Secure Software and Systems (ESSOS), pp. 15–28. Springer (2011)
21. Ramananandro, T.: Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method. Formal Aspects of Computing 20(1), 21–39 (Jan 2008)
22. Seater, R., Jackson, D., Gheyi, R.: Requirement progression in problem frames: deriving specifications from requirements. Requirements Engineering 12(2), 77–102 (Apr 2007)
23. Shao, D., Khurshid, S., Perry, D.E.: Whispec: White-box testing of libraries using declarative specifications. In: Proceedings of the 2007 Symposium on Library-Centric Software Design. pp. 11–20. LCSD '07, ACM, New York, NY, USA (2007)
24. Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., Franz, M.: Sok: Sanitizing for security. CoRR abs/1806.04355 (2018), `http://arxiv.org/abs/1806.04355`
25. Souaf, S., Berthomé, P., Loulergue, F.: A Cloud Brokerage Solution: Formal Methods Meet Security in Cloud Federations. In: International Conference on High Performance Computing Simulation (HPCS). IEEE (2018)
26. The Coq Development Team: The Coq Proof Assistant. `http://coq.inria.fr`
27. Torlak, E.: A constraint solver for software engineering : finding models and cores of large relational specifications. Ph.D. thesis, Massachusetts Institute of Technology (2009)
28. Torlak, E., Taghdiri, M., Dennis, G., Near, J.: Applications and extensions of Alloy: Past, present, and future. Mathematical Structures in Computer Science 23, 915–933 (2013)
29. Ulbrich, M., Geilmann, U., El Ghazi, A.A., Taghdiri, M.: A proof assistant for Alloy specifications. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 422–436. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)