F-Alloy: an Alloy Based Model Transformation Language

Loïc Gammaitoni and Pierre Kelsen

University of Luxembourg

Abstract. Model transformations are one of the core artifacts of a model-driven engineering approach. The relational logic language Alloy has been used in the past to verify properties of model transformations. In this paper we introduce the concept of functional Alloy modules. In essence a functional Alloy module can be viewed as an Alloy module representing a model transformation. We describe a sublanguage of Alloy called F-Alloy that allows the specification of functional Alloy modules. Transformations expressed in F-Alloy are analysable using the powerful automatic analysis features of Alloy but can also be interpreted efficiently without the use of backtracking.

1 Introduction

Alloy [13] is a formal language based on a first-order relational logic with transitive closure. It is based on a small set of core concepts, the main one being that of a mathematical relation. It was developed to support agile modeling of software designs. It does this by allowing fully automatic analysis of software design models using SAT solving. By providing immediate feedback to users, the use of Alloy is meant to facilitate identifying design errors early.

In the context of model-driven development the Alloy language has been used to verify properties of models and model transformations. The approach for verifying model transformations typically involves translating the model transformation language to Alloy. On the basis of this translation one can exercise the transformation on a suitably constrained set of input models. One can apply the Alloy Analyzer tool to generate the specified set of models as well as the corresponding target models.

Thus, in a sense, one can execute a model transformation using the Alloy analyzer as an execution engine. This approach is impractical for two reasons:

– Despite many advances in the performance of SAT solvers the analysis can become quite time consuming when the model requires larger scopes to find a suitable instance.
– The problem of finding small upper bounds (scopes) for the number of entities of the different types is itself non-trivial (in fact it is undecidable). This is particularly problematic for complex models with many different entity types.

In this paper, we introduce the notion of functional Alloy modules as specifications of model transformation from a source to a target metamodel (represented by Alloy modules). We show that under certain conditions such a functional Alloy module can be efficiently interpreted instead of being analyzed via SAT solving. More precisely we define a sub-language of Alloy, named F-Alloy, that allows to express functional Alloy modules and that guarantees that these modules can be interpreted efficiently, that is, in polynomial time.

A central concept of F-Alloy are so-called *bridge mappings* which are essentially injective functions. We may thus view F-Alloy as a relational model transformation language. Compared to existing relational model transformation languages (of which QVT Relational [16] is a prominent representative) our approach offers two notable features:

– rather than defining a new model transformation language from scratch we restrict an existing formal language in order to express model transformations. An important consequence of this approach is the possibility to reuse the formal semantics of the Alloy language, thus permitting verification of model transformations using Alloy's automatic analysis capabilities.
– interpretation directly exploits the functional nature of model transformations. This allows efficient backtrack-free execution of model transformations. We demonstrate the effectiveness of this functional approach by applying it to a non-trivial example, namely, the CD to RDBMS model transformation that has been used as standard example for evaluating model transformation approaches.

The paper is structured as follows. In the next section we present the running example — namely a transformation from Class Diagrams to Relational Database Management Systems — that will be used to evaluate our approach. In section 3 we give a formal presentation of central concepts of Alloy. In section 4 we introduce the notion of functional Alloy module and illustrate its relation with model transformations. Sections 5 and 6 present the syntax and (translational) semantics of F-Alloy. In section 7 we explain how F-Alloy modules can be efficiently interpreted. We provide an evaluation of our approach in section 8 by comparing the performance of analysis and interpretation in the execution and verification of the CD2RDBMS transformation. We explain the context of our work and discuss related work in section 9. The final section presents concluding remarks and future work.

2 Running Example: The CD2RDBMS Transformation

To evaluate our approach, we shall use the standard Class Diagram to Relational Database Management System transformation case study [6] — which we will call CD2RDBMS. The source and target metamodels of this transformation, CD and RDBMS, are shown as UML class diagrams in fig.1; further constraints have been left out for succinctness. We now give an informal specification of this transformation:

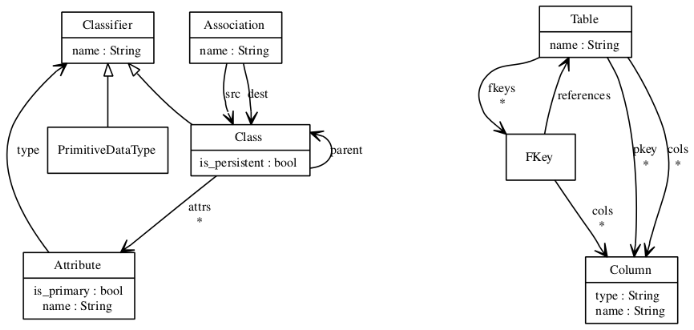

Fig. 1. CD and RDBMS metamodels

For each persistent class c without a parent, a table is created. This table is populated with columns (1) corresponding to the primitive attributes of c, (2) referring to the class of class-typed attributes of c, (3) referring to the destination of the associations having as source c, (4) corresponding to attributes declared in children of class c.

In case (1) the column is typed after the primitive type of the attribute, and named after the attribute. In case (2), we create a column for each primary attribute of the type class. Those columns compose a foreign key which refers to the table representing the type class. Case (3) is similar to case (2). We create columns referring to the association's destination's primary attributes. Those columns compose a foreign key that refers to the table corresponding to the destination.

In case (4), each subclass's attribute is created in conformance to point (1) and (2) in the table corresponding to the topmost superclass.

Note that in cases (2), (3) and (4) the naming of the column depends both on the nature (referred class, association, inheriting class respectively) and name of the attribute.

A complete solution for this case study based on F-Alloy can be found in [9].

3 Background

3.1 Alloy Modules and Instances

A metamodel can be expressed in one or several Alloy modules, each module being associated to a single file. Modules are composed of signature and field declarations, and of constraints. A module may *import* other modules, in which case the importing module can use features of the imported modules.

Definition 1 (Alloy Module, Signature, Field). *An Alloy module is a tuple* (S, F, φ) *with* S *and* F *being the sets of signatures and fields declared in the module or any of its (recursively) imported modules, respectively. Signatures may* *be defined as sub-signatures of other signatures (using the* extends *keyword). Fields have as type a sequence of signatures, the first one being the signature that contains it. φ is a first-order logic formula (plus transitive closure) representing the set of constraints expressed in the module.*

The RDBMS module is defined in Alloy as follows:

```
module RDBMS

abstract sig RDBMSElem{
  disj  label: seq String
}

sig Table extends RDBMSElem{
  disj cols : some Column,
  pkey : some Column,
  fkeys: set FKey
}{pkey in cols}

sig Column extends RDBMSElem{
  type : Type
}{this in Table.cols}

sig FKey{
  references: Table,
  disj columns: set Column
}{this in Table.fkeys}

abstract sig Type{}
one sig Number,Text extends
      Type{}
```

It can be written $m = (S, F, \varphi)$ with:

- $S = \{\texttt{Table}, \texttt{Column}, \texttt{FKey}, \texttt{RDBMSElem}, \texttt{Type}, \texttt{Number}, \texttt{Text}\}$
- $F = \{\texttt{cols} : \texttt{Table} \times 2^{\texttt{Column}}, \texttt{pkey} : \texttt{Table} \times 2^{\texttt{Column}}, \texttt{fkeys} : \texttt{Table} \times 2^{\texttt{FKey}}, \texttt{type} : \texttt{Column} \times \texttt{Type}, \texttt{references} : \texttt{Fkey} \times \texttt{Table}, \texttt{columns} : \texttt{FKey} \times 2^{\texttt{Column}}, \texttt{label} : \texttt{RDBMSElem} \times \texttt{Int} \times \texttt{String}\}$
- $\varphi = (\forall t : \texttt{Table}, \texttt{pkey}(t) \in \texttt{cols}(t)) \wedge (\forall c : \texttt{Column}, \exists t : \texttt{Table}, c \in \texttt{cols}(t)) \wedge (\forall f : \texttt{FKey}, \exists t : \texttt{Table}, f \in \texttt{fkeys}(f)))$

Considering now A, a set of indivisible entities called atoms, and T, a set of atom tuples, and a module $m = (S, F, \varphi)$, we call: *typed atoms* pairs (x, s) where $x \in A$ and $s \in S$. Typed atoms (x, s) are also denoted x^s (read "atom x of type s"); *typed tuples* pairs (t, f) where $t \in T$ and $f \in F$. Typed tuples (t, f) are also denoted t^f (read "tuple t of type f"). Note that for a typed field t^f the following needs to hold: if the type of the field is $X_1, \ldots, X_n$, then the i-th component of the tuple needs to have as type X_i or a subsignature of X_i.

We call x^s an *s-atom* and t^f an *f-tuple*, and extend the superscript notation such that sets of s-atoms B and of f-tuples T, are denoted B^s and T^f, respectively.

Definition 2 (Alloy Instance). *An Alloy Instance of m is a triplet (X, Y, m) where $m = (S, F, \varphi)$, X is a set of atoms typed by signatures of m and Y is a set of tuples typed by fields of m. We write $x \vDash \varphi$ if an instance x of m satisfies φ and call valid instances [1] (of m) the subset of instances (of m) which satisfy φ. We denote the set of valid instances of m by $I(m)$. Formally:*

$$I(m) = \{(X, Y, m) | \forall x^v \in X, v \in S \wedge \forall y^w \in Y, w \in F \wedge (X, Y, m) \vDash \varphi\}$$

Instance (X, Y, m) is a *sub-instance* of (X', Y', m') if $X \subseteq X'$ and $Y \subseteq Y'$.

Note that the definition of the set of valid instances does not take into account bounds on the numbers of atoms typed by different signatures. These bounds are collectively known as the *scope* of a module (see [13]). Scopes need only to

[1] We relax here the Alloy terminology in which *instance* usually means *valid instance*

be taken into account when performing actual analyses with the Alloy Analyser, which is deferred until section 8.

The projection of an instance x on a module m' is meant to extract an m'-instance out of atoms and tuples present in x. This operation will be used extensively later in the paper.

Definition 3 (Instance Projection). *A projection of an instance $x : (X, Y, m)$ on a module $m' : (S', F', \varphi')$ is the m'-instance composed of the atoms and tuples present in x and typed by signatures and fields of m', respectively. We denote projections using the evaluation symbol $\Downarrow$: $x \Downarrow m'$ reads "the projection of x on m'". Formally : $x \Downarrow m' = (X', Y', m')$ with $X' = \{a^s | a \in X \wedge s \in S'\}$ and $T' = \{t^f | t \in T \wedge f \in F'\}$.*

4 Functional Alloy Modules

Suppose an Alloy module m imports two modules m_1 and m_2. An instance of m will then contain an m_1- and m_2-sub-instance. Furthermore, module m induces a binary relation over $I(m_1) \times I(m_2)$ defined as follows:

$$\forall x_1 \in I(m_1), x_2 \in I(m_2) : R(x_1, x_2) \Leftrightarrow \exists x \in I(m) : x \Downarrow m_1 = x_1 \wedge x \Downarrow m_2 = x_2$$

In this paper we restrict ourselves to one-to-one model transformations, that is, one input model is mapped to exactly one output model. In other words the previously defined relation should be a mathematical function.

This motivates the following definition:

Definition 4 (Functional Alloy Module). *An Alloy module m importing two modules m_1 and m_2 is called a functional Alloy module from m_1 to m_2 if for any valid instances x and x' of m, if x and x' have the same projection on m_1, then they also have the same projection on m_2.Formally:*

$$\forall x, x' \in I(m), (x \Downarrow m_1 = x' \Downarrow m_1) \implies (x \Downarrow m_2 = x' \Downarrow m_2)$$

Caveat: This definition only makes sense if m_1 and m_2 are distinct, that is, for the case of exogenous transformations. Indeed if $m_1 = m_2$ the condition stated in the definition trivially holds. In the following we thus restrict our attention to *exogenous transformations*.

To illustrate this definition, consider the CD2RDBMS transformation. A hypothetical Alloy module defining this transformation would import the modules defining the class diagram and the RDBMS metamodels and would define a set of "rules" that specify the transformation. Such a module would be a functional Alloy module if and only if any two valid instances of it having the same projection on the class diagram module would have the same projection on the RDBMS module. Of course we still have not explained how to write such a functional Alloy module. This will be explained in section 5 when we define a sub-language of Alloy for expressing functional Alloy modules.

5 Syntax of F-Alloy

In this section we formally introduce the syntax of F-Alloy, a new language meant to ease the specification of functional Alloy modules.

We call *f-module* m from m_1 to m_2, a module m, written in F-Alloy, importing module m_1 and m_2. An `<F-Module>` is composed of :

- A `<Bridge>` signature (of multiplicity one[2]) allowing to define and keep track of functions from m_1 to m_2. Those functions are called bridge `<Mapping>`.
- `<Guard>` predicates, each associated to one bridge mapping. Their role is to define via the use of an Alloy Formula (`<Formula>`) under which condition an element of m_1 is part of the associated mapping.
- `<Value>` predicates also associated to a bridge mapping. Their role is to provide additional details on how the output instance is constructed. It contains interpretable Alloy formulae called `<Rules>`.

We split the BNF definition of F-Alloy in two parts in order to ease its understanding. While the first part reveals the structure of F-modules, the second part focuses on those interpretable Alloy formulae called rules.

```
<F-Module>::= module <qualName> <import> <Bridge><Guard>*<Value>*
<import>::= import <qualName> import <qualName>
<Bridge> ::= one sig Bridge {<Mapping>*}
<Mapping> ::= <name>  :  <qualName> (-><qualName>)+,
<Guard> ::= pred guard_<name> ( <paraDecl>* ){ <Formula> }
<Value> ::= pred value_<name>  ( <paraDecl>* ){ <Rule>* }
<paraDecl>::= (<name>:<qualName>,)*<name>:<qualName>
<qualName>::= [this/] (<name>/)* <name>

-----------------------------------------------------------------------------
<Rule>::= <Formula> implies <Rule>|<Strict>|<Loose>|<Loop>
<Strict> ::= <name>.<field> = [<Formula> implies <value> else]? <value>
<Loose> ::= <name> in Bridge.<field>.<field>
<Loop> ::= all <name>:<Expr>|<Expr> implies <Rule>
<value>::= [<Expr>|Bridge.<field>]
<field> ::= <field> [[<Expr>]]?
```

Listing 1.1. F-Alloy BNF

Additional static semantics constraints for the syntax are: (1) There is exactly one guard and one value predicate per bridge mapping, and the association is done by name; (2) the qualified names in the `<Mapping>` except the last one correspond to signatures in m_1, while the last one refers to a signature of m_2; (3) there is one parameter in the guard predicate for each m_1-signature in the `<Mapping>` (4) the same holds for the value predicate, with an additional parameter for the m_2-signature;

Here is an excerpt of the CD2RDBMS transformation expressed in F-Alloy:

[2] valid instances of the f-Module will contain exactly one Bridge atom

```
 1 module UML2RDBMS
 2
 3 open CD/AbstractSyntax/CD
 4 open RDBMS/AbstractSyntax/RDBMS
 5
 6
 7 one sig Bridge{
 8   class2table: Class  ->  Table,
 9   primAttr2column: Attribute  ->  Column,
10   classAttr2column: Attribute  ->  Attribute
        ->   Column,
11   classAttr2Fkey: Attribute  ->  FKey,
12   association2column: Association  ->
         Attribute  ->  Column,
13   association2FKey: Association  ->  FKey,
14  }
15
16 pred guard_class2table(c:Class){
17   c.is_persistent=True
18   c.parent=none
19 }
20
21 pred value_class2table(c:Class ,  t:Table){
22   t.label[0]=c.name
23 }
```

This f-module UML2RDBMS (declared on l.1) from CD (imported on l.3) to RDBMS (imported on l.4) contains 6 bridge mappings (l.8-13) and the guard and value predicates of only one mapping (the others are omitted for lack of space). The bridge mapping `class2table` defines a partial function from Class to Table. The guard predicate of `class2table` defines that the domain of this function consists only of persistent (l.17) topmost (l.18) super classes. The value predicate of `class2table`, containing a single strict rule, states that the image of a class through mapping `class2table` should be labelledafter the name of the class (l.22).

We note that, from a syntactic point of view, any f-module is also an Alloy module since it is essentially composed of a signature and a collection of predicates. In that sense F-Alloy is a sub-language of Alloy. The intended meaning of an f-module is however different from its Alloy semantics, as explained in the next section. Indeed additional constraints need to be added to ensure that the module is a functional Alloy module, i.e., it specifies a transformation.

6 Translational Semantics of F-Alloy

In this section we define the semantics of F-Alloy using the semantics of Alloy. For the purpose of this paper we define the meaning of an Alloy module to be its set of valid instances. We map an f-module $m : (S, F, \varphi)$ expressed in F-Alloy to an Alloy module m_A - called *augmented module* - that is obtained by adding constraints to m. The meaning of f-module m is then equal to the meaning of the augmented Alloy module (defined above). Later we will show that the augmented module is in fact a functional Alloy module.

Five different types of constraints are added to m. We illustrate those using excerpts of our CD2RDBMS case study.

Map Disjunction. Bridge mappings of an f-module define partial functions which have disjoint ranges.

E.g., columns representing primitive and class attributes should be disjoint.

```
primAttr2column[Attribute] & classAttr2column[Attribute] = none
```

Map Injectiveness Functions defined by Bridge mappings are injective.

E.g., a given Column shouldn't be mapped to two different primitive attributes through the same mapping.

```
forall disj a1,a2 : Attribute| primAttr2column[a1] ≠
    primAttr2column[a2]
```

Predicate association. Guard and value predicates of an f-module associated with a bridge mapping condition its valuation and the valuation of its output elements' field, respectively.

E.g., a column y is associated to an attribute x if and only if the guard predicate is satisfied for x. In that case, the value predicate has to hold for x and y as well.

```
  all x : Attribute    |
  (guard_primAttr2column[x] and #primAttr2column[x]=1 and
      value_primAttr2column[x , primAttr2column[x] ]) or
  (not guard_primAttr2column[x] and primAttr2column[x]=none)
```

Minimum Output. In a valid instance of an f-module m from m_1 to m_2, atoms typed by a signature of m_2 are limited to the ones that are part of a bridge mapping of m.

E.g., RDBMS elements are limited to co-domains of declared mappings.

```
RDBMSElem = class2table[Class] + primAttr2column[Attribute] +
    classAttr2column[Attribute,Attribute] +
    association2column[Association,Attribute]
```

Minimal Assignment. Rules of an f-module follow the principle of minimal assignment. In other words, the valuation of a field is limited to the values explicitly assigned through the rules.

E.g., the label of a column being a sequence, its size is bounded by the number of elements explicitly assigned through rules (see the last of the following constraints).

```
c.label[0]= a2.name
c.label[1]= a1.name
c.label[2]= ((a1.~attrs.parent)≠none implies a1.~attrs.name
            else none)
  all i:Int| i≥1 and i≤ #(a1.~attrs.*parent) implies c.label[add
      [i,1]]= c.label[i].~name.parent.name//5
 #c.label.elems=add[#(a1.~attrs.*parent),1]
```

6.1 Rule semantics

In order to prove in the next sub-section that the augmented module is a functional Alloy module, we need two properties of rules that are expressed in the two lemmas below.

The first lemma claims that each rule in a value predicate can be rewritten in the form:

$$F_r \texttt{ in } g$$

where g denotes a field in m_2, F_r is a set-valued expression typed by g and `in` denotes set inclusion (in Alloy). Since F_r depends in general on the instance x_A of m_A and on the parameters $\vec{x}$ and y of the valued predicate containing r, we write F_r as $F_r(x_A, \vec{x}, y)$. We use the vector notation for $\vec{x}$ since it represents a sequence of parameters typed by signatures of m_1.

Lemma 1 (Rules as functions). *Any rule r of m_A can be written in the form $F_r(x_A, \vec{x}, y)$ `in` g for some field g in m_2.*

Proof sketch. We only consider the case of loose rules. A loose rule of the form `y in Bridge.f[expr1].g` can be rewritten using the equivalent Alloy constraint: `(Bridge.f[expr1] -> expr2 -> y) in g`. If b and e denote the value of `Bridge.f[expr1]` and `expr2` for a given instance x_A and arguments $\vec{x}$, then we can define $F_r(x_A, \vec{x}, y) = \{(b, e, y)^g\}$. F_r can be defined similarly for the other types of rules. □

The second lemma (whose proof is omitted) states that function $F_r(x_A, \vec{x}, y)$ only depends on the projection of x_A on m_1.

Lemma 2 (F_r is independent of m_2). *For any rule r of an f-module m, considering the function F_r associated to r (see lemma 1), we have :*

$$\forall \vec{x}, y\ \forall x_A, x'_A \in I(m_A), F_r(x_A, \vec{x}, y) = F_r(x'_A, \vec{x}, y)\ \textit{if}\ x_A \downarrow m_1 = x'_A \downarrow m_1$$

6.2 Augmented Modules and Functional Alloy Modules

Theorem 1 (m_A is a functional Alloy module). *For any f-module m from m_1 to m_2 the corresponding augmented module m_A is a functional Alloy module from m_1 to m_2.*

Proof sketch. By the minimum output constraints of m_A the atoms in the projection of a valid instance x_A of m_A on m_2 are exactly those in the ranges of bridge mappings. The set of these atoms depends only on the projection of x_A on m_1 (up to atom renaming).

From lemma 1 and lemma 2 we know that each rule of each value predicate contributes to an instance x_A of m_A a set of tuples typed by a field of m_2 that only depends on the projection on m_1. By taking the union of these sets of tuples over all bridge mappings and rules, the resulting set of tuples still only depends on the projection of x_A on m_1. The construction rules for the augmented module guarantee that only those tuples explicitly added by rules will be in the projection of x_A on m_2. It follows that m_A is a functional Alloy module. □

7 F-Alloy Interpretation

The following pseudocode shows how interpretation of an f-module works. Note that the output is an instance of the augmented module. If one is interested only in the m_2-subinstance, it can be obtained by projecting the m_A-instance on m_2.

For an instance $x = (X, Y, m)$, a set of atoms A and a set of tuples T, we use the notation $x \cup A$ and $x \cup T$ to denote the instances $(X \cup A, Y, m)$ and $(X, Y \cup T, m)$, respectively. We use the vector notation $\vec{X}$ to denote the sequence of m_1-signatures in the definition of a bridge mapping.

```
 1 Input:  -f-module $m$ from $m_1 : (S_1, F_1, \varphi_1)$ to $m_2 : (S_2, F_2, \varphi_2)$
 2         -Instance $x_1$ of $m_1$
 3 Output: -Instance $x_A = (X_A, Y_A, m_A)$ s.t. $x_A \downarrow m_1 = x_1$
 4
 5 BEGIN
 6   $x_A := x_1 \cup \{b^{\texttt{Bridge}}\}$
 7   FOR EACH mapping $f : \vec{X} \to Y$ IN $m$ DO:
 8     LET $\vec{X}_f$ denote the set of $\vec{X}$ tuples (of atoms present in $x_1$) that
           satisfy the guard of mapping $f$
 9     LET $Y_f$ be a set of $Y$-atoms s.t. $|Y_f| = |\vec{X}_f|$ and $Y_f \cap x_A = \emptyset$
10     LET $T_f \subseteq \vec{X}_f \times Y_f$ be a set of tuples $(\vec{x}, y)$ that maps $\vec{X}_f$ bijectively to
           $Y_f$
11     $x_A := x_A \cup Y_f \cup T_f$
12   DONE
13   FOR EACH mapping $f : \vec{X} \to Y$ IN $m$ DO:
14     FOR EACH rule $r$ IN pred value_$f$ DO:
15       FOR EACH  tuple $(\vec{x}, y)$ IN $T_f$ DO: // $T_f$ defined on line 10
16         $x_A := x_A \cup F_r(x_A, \vec{x}, y)$ //F_r defined in lemma 1
17       DONE
18     DONE
19   DONE
20   IF $x_A \models \varphi_1 \wedge \varphi_2$ THEN
21     RETURN $x_A$
22    ELSE
23     invalid transformation
24 END
```

Listing 1.2. F-Alloy Interpretation pseudo code

Let us analyse the time complexity of interpretation. Let n denote the number of atoms in x_1. Both in the first and the second loop we need to evaluate an Alloy constraint or expression on a number of tuples that is at most polynomial in n. If we assume that the evaluation of Alloy expressions and constraints can be done in time polynomial in n - which can be shown by structural induction - then the overall time will be at most polynomial in n. Thus we expect interpretation to be efficient. That this is true in practice will be supported by our experimental results in the next section.

The following theorem states that the interpretation of f-modules implemented by the pseudo code of listing 1.2 conforms to the translational semantics given to F-Alloy.

Theorem 2. *Given an f-module m from m_1 to m_2 and a valid instance x_1 of m_1, the instance x_A returned by interpretation (in line 21) on inputs m and x_1*

is a valid instance of m_A. Moreover interpretation returns no instance only when there is no valid instance for m_A whose projection on m_1 is x_1.

Proof sketch. From lines 9 and 10 we see that map disjunction and map injectiveness constraints are satisfied. From lines 13 — 19 it follows that the predicate association constraints are satisfied in x_A. From lines 7—12 it follows that the atoms in the projection of x_A on m_2 are exactly those in the ranges of bridge mappings, implying that the minimum output constraints are satisfied. Finally the minimal assignment constraints follow from the fact that only those tuples are added on lines 13 — 19 which are explicitly required by the rules.

In the case the interpretation of an f-module m fails to produce an instance satisfying constraints of m_1 and m_2, then so will analysis. Indeed because of the constraints of m_A, any valid instance of m_A will have the same atoms and the same tuples in the projection of x_A on m_2 (up to atom renaming) than the interpreted instance since those tuples are exactly the tuples explicitly required by the rules. □

8 Evaluation

In this section we evaluate the benefits of using F-Alloy to specify model transformations. This evaluation is based on comparing the performance of traditional analysis and of F-Alloy interpretation in two cases :

1. The computation of a transformation (for a given input instance)
2. The verification of a transformation (no input given)

The manipulation needed to obtain the results presented in this section were performed in our Lightning tool[1] on models of the CD2RDBMS case study.

8.1 Transformation Computation

We start by comparing the performance of analysis and interpretation in the computation of the CD2RDBMS transformation.

This manipulation consists, given a CD-instance x_1 and the CD2RDBMS transformation expressed as an f-module m from m_1 (CD) to m_2 (RDBMS):

- **In the case of analysis:**
  - In deriving the augmented module m_A from m
  - In "over-constraining" m_1 such that $\forall x_A \in I(m_A), x_A \downarrow m_1 = x_1$
  - in computing appropriate scopes (which will depend on the size of x_1) for the signatures in the augmented module
  - in launching the actual analysis based on these scopes
- **In the case of interpretation:** In interpreting the f-module m given instance x_1.

The result of those manipulations for CD-instances of three different sizes are given in table 1.

The complexity of analysis grows very quickly with the size of the input instance while interpretation exhibits a nearly linear behavior. This can be viewed as a first confirmation of the theoretical complexity analysis done in section 7.

number of UMLElem atoms	CD2RDBMS analysis (ms)	CD2RDBMS interpretation (ms)
10	2324	71
20	8052	162
25	20006	188

Table 1. Transformation Computation : Time performance comparison table

8.2 Transformation Verification

We now compare the performance of analysis and interpretation in the verification of a transformation. While different types of verification may be done, we consider here only the generation of examples of the transformation, which would help in establishing consistency and also point to abnormal behavior.

The manipulation consists:

- **In the case of analysis:** in analysing the augmented module m_A for the given exact scope associated with the UMLElem signature.
- **in the case of interpretation:** In analysing m_1 for the given scope and for each m_1-instance x_1 thus obtained, in interpreting the f-module m. Note that from theorems 1 and 2 it follows that the set of instances thus produced is equivalent — i.e., its instances have the same projections on m_2 — to the set of instances obtained by analysis.

The result of those manipulations are given in table 2.

UMLElem scope (number of atoms)	Analysis	Interpretation		
	CD2RDBMS analysis (ms)	CD analysis (ms)	CD2RDBMS interpretation (ms)	Total Time (ms)
10	5448	448	68	516
20	83759	974	159	1133
25	∞	1256	192	1448

Table 2. Transformation Verification : Time performance comparison table

The *Total Time* column gives the average amount of time needed in the case of verification with interpretation to obtain the first instance.The other instances are obtained seamlessly when browsing the instances.

We notice from those results that the complexity of analysing the transformation module can be reduced, with the use of interpretation, to the complexity of analysing its input module.

9 Discussion and Related Work

Context The motivation to search for a model transformation language based on Alloy stems from investigating the use of Alloy for designing a language workbench [1, 10]. In an earlier publication [8] we already showed that the concrete syntax of a language can be defined as a transformation using Alloy. The current work opens up the possibility to integrate the specification of general model transformations (e.g., for specifying operational semantics of languages) into the Alloy based language workbench.

F-Alloy vs. Alloy. Analyzing Alloy models is generally an undecidable problem. That is why actual analyses with the Alloy analyser are always done for a finite scope using SAT-solving, itself an NP-complete problem. In practice Alloy's analysis, although having a high worst case complexity, works surprisingly well, as documented in numerous publications. No guarantees can be given, though, on the time needed for analysing Alloy modules. Contrary to this we have shown in this paper that F-Alloy identifies a subset of Alloy modules for which analysis via interpretation can be done in polynomial time (see section 7). Furthermore interpretation of modules written in F-Alloy relieves the analyst of having to determine proper scopes for the signatures, itself a non-trivial problem.

Related work on model transformation languages. We can consider the F-Alloy language as a simple relational model transformation language. Relational model transformation languages (such as those given in [2], [16] and [11]) are those where the main concept is that of a mathematical relation [7]. Note that in F-Alloy the mathematical relations, represented by the bridge mappings, are in fact injective functions. In their pure form (e.g., [2]) relational specifications are not executable. In other cases (e.g., [16]) they are executable in principle but still lack proper tool support. In the case of QVT there are some tools that execute QVT specifications but none of them take into account all the features of the QVT language. This is an indication that providing execution semantics for a relational language is a non-trivial task, especially if some semantic inconsistencies exist as is the case for QVT ([15]). In this paper we have shown that F-Alloy specifications are efficiently executable.

One distinguishing feature of F-Alloy is that it inherits a formal semantics from the host language Alloy. Not all model transformation languages are formal. For instance a popular model transformation language called ATL [14] was defined semi-formally. A formal semantics in terms of rewriting logics was later given by [19]. Even if a formal semantics is given there is in general no guarantee that the implementation does indeed conform to the semantics. A good illustration of this is the case of the triple graph grammar approach [17, 18], for which the authors of [12] describe an approach to show conformance of an existing implementation to the formal semantics.

Related work on verifying model transformation languages. As mentioned in the introduction Alloy has been used in the past to verify model transformations. Anastasakis et al. [4] use Alloy to analyze the correctness of model transformations. They resort to their tool UML2Alloy [3] to transform the source and target metamodels into Alloy and translate the transformation rules into mapping relations and predicates at the Alloy level. The goal of their work is to check that the target instances are conforming to the target metamodel of the transformation. This is done by checking an Alloy assertion using the Alloy analyzer. In a similar line of work Baresi et al. [5] use Alloy to represent graph transformations represented in the AGG formalism. They use the Alloy analyzer to verify the correctness of the transformation by generating possible traces. We can apply these results to F-Alloy since f-modules denote Alloy modules that can be verified with these approaches. Furthermore, as we show in the evaluation section, in certain cases we can speed up the analysis using interpretation.

10 Conclusion and Future Work

In this paper we have introduced the notion of functional Alloy module which corresponds to an Alloy module representing a transformation. We have defined a sub-language of Alloy, named F-Alloy, which can be used to express functional Alloy modules and allows efficient interpretation of these modules. We have given first evidence of this for the CD2RDBMS model transformation. A more thorough evaluation will be needed for further confirmation.

F-Alloy inherits the formal semantics of Alloy, thus making the transformations analyzable. This contrasts with other approaches where a separate formal semantics has to be defined. It also implies that existing verification techniques based on Alloy can be applied to F-Alloy, with the added possibility of speeding up the actual analysis via interpretation.

Our current approach has one important restriction: as pointed out in section 4 the notion of functional Alloy modules, in its present form, only applies to exogenous transformation. Further work will investigate how to extend the approach to endogenous model transformations.

Another area of investigation concerns bidirectional transformations. These are transformations that allow forward and backward transformations to be generated from a unique transformation specification. Bidirectional transformations are useful in the context of synchronisation between models. Future work will examine whether we can make our approach bidirectional. This has already been achieved by existing relational model transformation languages such as QVT but also graph based approaches such as triple graph grammars.

References

1. Lightning tool website, http://lightning.gforge.uni.lu.
2. David H. Akehurst, Stuart Kent, and Octavian Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Software and System Modeling*, 2(4):215–239, 2003.
3. Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy: A challenging model transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. Springer, 2007.
4. Kyriakos Anastasakis, Behzad Bordbar, and Jochen M Küster. Analysis of model transformations via alloy. In *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.
5. Luciano Baresi and Paola Spoletini. On the use of alloy to analyze graph transformation systems. In *Graph Transformations*, pages 306–320. Springer, 2006.
6. Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Model transformations in practice workshop. In *Satellite Events at the MoDELS 2005 Conference*, pages 120–127. Springer, 2006.
7. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17, 2003.
8. Loïc Gammaitoni and Pierre Kelsen. Domain-specific visualization of alloy instances. In *ABZ 2014*, page to appear. Springer, 2014.
9. Loïc Gammaitoni and Pierre Kelsen. An f-alloy specification for the cd2rdbms case study, http://lightning.gforge.uni.lu/doc/TR-LASSY-15-01.pdf.
10. Loïc Gammaitoni, Pierre Kelsen, and Fabien Mathey. Verifying modelling languages using lightning: a case study. In *Proceedings of the 11th Workshop on Model-Driven Engineering, Verification and Validation co-located with 17th International Conference on Model Driven Engineering Languages and Systems, MoDeVVa@MODELS 2014, Valencia, Spain, September 30, 2014.*, pages 19–28, 2014.
11. Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of mda. In *Graph Transformation*, pages 90–105. Springer, 2002.
12. Holger Giese, Stephan Hildebrandt, and Leen Lambers. Toward bridging the gap between formal semantics and implementation of triple graph grammars. In *MoDeVVa 2010*, pages 19–24. IEEE, 2010.
13. Daniel Jackson. *Software abstractions*. MIT press Cambridge, 2012.
14. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.
15. Nuno Macedo and Alcino Cunha. Implementing qvt-r bidirectional model transformations using alloy. In *FASE 2013*, pages 297–311. Springer.
16. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011.
17. Andy Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995.
18. Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *Graph Transformations*, pages 411–425. Springer, 2008.
19. Javier Troya and Antonio Vallecillo. A rewriting logic semantics for atl. *Journal of Object Technology*, 10(5):1–29, 2011.