**Noname manuscript No.**
(will be inserted by the editor)

# F-Alloy: A Relational Model Transformation Language Based on Alloy

**Loïc Gammaitoni · Pierre Kelsen**

the date of receipt and acceptance should be inserted later

**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 specifically designed to concisely specify functional Alloy modules. The restrictions on F-Alloy's syntax are meant to allow efficient execution of the specified transformation, without the use of backtracking, by an adapted interpretation algorithm. F-Alloy's semantics is given in this paper as a direct translation to Alloy, hence F-Alloy specifications are also analyzable using the powerful automatic analysis features of Alloy.

**Keywords** Model Transformation · F-Alloy · Alloy · Analysis · Formal method · endogenous · exogenous

## 1 Introduction

In recent works [12,10], we investigated the use of Alloy [17], a formal language based on a first-order relational logic with transitive closure, in the definition of Domain Specific Languages (DSLs) [31]. These investigations were led by the intuition that Alloy, which has been successfully used to validate and verify systems from a variety of domains [13,28], could also be used to validate and verify DSLs specifications. Ultimately, we defined a design cycle for DSL design in which Alloy analysis is applied at every step, hence allowing an agile validation of every aspect of the DSL defined: abstract syntax, concrete syntax, and operational semantics, all defined in Alloy modules. The Lightning[1] language workbench is an environment enabling the definition of DSLs following this design cycle.

Model transformations play a key role in the specifications of DSLs. Indeed, the concrete syntax takes the form of an exogenous transformation from the Abstract Syntax Model to a Visual Language Model and the operational semantics is defined by an endogenous inplace transformation from/to a semantic domain model defining valid execution states of the language.

While it is known that model transformations can be specified in Alloy [6,4,21], their execution relying on the Alloy analyzer (a SAT-based tool) is deemed impractical for two reasons:

- Despite many advances in the performance of SAT solvers [19], the analysis of a model can become quite time consuming when it requires larger scopes to find a suitable instance (as shown by measurements listed in our evaluation section).
- The problem of determining small scopes, which are needed in order to find relevant instances in a reasonable time, is itself non-trivial.In the context of modelling model transformations, scopes would be needed both for concepts in the source metamodel and the target metamodel (both represented in Alloy), assuming we want to generate pairs of input and output models.

In this work, we propose a solution to overcome this limitation. This solution takes the form of a new language, called F-Alloy, specifically designed to allow the concise specification of efficiently executable model

Loïc Gammaitoni (✉) · Pierre Kelsen
University of Luxembourg
E-mail: loic.gammaitoni@uni.lu
E-mail: pierre.kelsen@uni.lu

[1] http://lightning.gforge.uni.lu

transformation. This F-Alloy language is based on a subset of the syntax of Alloy but its associated semantics differs in the sense that every Alloy construct is interpreted in the context of model transformations. This semantics "alteration" can be translated into explicit Alloy constraints hence allowing to easily translate F-Alloy specification into Alloy.

It would of course have been possible in principle to reuse an existing model transformation language (such as ATL) for specifying the model transformations in Lightning. However, since we require DSL specifications to be analyzable in Lightning, a translation from that model transformation language to Alloy would also be needed. For the case of F-Alloy this translation is fairly simple since F-Alloy reuses the structural concepts of Alloy and only constraints need to added. Because this is not the case for other model transformation languages, a more complex translation procedure is likely to be needed for those languages. Furthermore the structural similarity between F-Alloy and Alloy should ease the learning of this language for potential users of Lightning since the use of Lightning requires some familiarity with Alloy.

The model transformations we consider in this paper are exogenous (out-place) model transformations (source and target are distinct Alloy modules) and endogenous in-place transformations (source and target are the same Alloy module and the transformation is expressed in terms of refinement operations) – see taxonomy of model transformations given in [22]. We ignore endogenous out-place transformations as we regard them as a special application of exogenous out-place transformations, where the target meta-model is a duplicate of the source metamodel, considered as different though declaring the exact same concepts. Throughout the paper, we therefore mean by endogenous transformation, if not stated otherwise, endogenous in-place transformation.

A central concept of F-Alloy are so-called *mappings* which are essentially injective functions. The F-Alloy language can thus be viewed as a relational model transformation language (since functions are special cases of relations). Compared to existing relational model transformation languages (of which QVTr [23] is a prominent representative) F-Alloy offers two notable features:

– F-Alloy specification can natively be translated into Alloy for validation's sake.
– execution (which we will refer to as interpretation) directly exploits the functional nature of model transformations , i.e., the fact that for each input model there is at most one output model. This allows efficient backtrack-free execution of model transformations.

We demonstrate the effectiveness of this functional approach by applying it to two examples, namely, the Class Diagrams to Relational Database Management Systems (`CD2RDBMS`) exogenous transformation and a Class Diagram refinement (`CDRefinement`) endogenous transformation (see Section 3).

The paper is structured as follows. In the next section, we provide an overview of the different notions introduced in this paper. In Section 3 we present two running examples that will be used to illustrate and evaluate our approach. In Section 4 we introduce the central concepts of Alloy. In Section 5 we introduce the notion of functional Alloy module and explain its relation with model transformations. Sections 6 and 7 present the syntax and (translational) semantics of F-Alloy. In Section 8 we illustrate how F-Alloy specifications can be efficiently interpreted. We provide an evaluation of our approach in Section 9 by comparing the performance of analysis and interpretation for both case studies. We explain the context of our work and discuss related work in Section 10. The final section presents concluding remarks and future work.

**Reader's guide:** We propose different ways to read the paper matching the different expectations readers may have. All readers should first get an idea of the overall approach by reading Section 2. If readers are interested in:

– the practical use of F-Alloy, then they can read:
  – Section 3 providing case studies and their implementation in F-Alloy as well as in other recognized languages like TGG and Henshin.
  – Optionally Section 4.2 to get a short and informal introduction to Alloy's concepts
  – Section 6.2, 6.4 and 6.5 to get familiar with the syntax of F-Alloy
  – Optionally Section 8 to understand how F-Alloy specifications are interpreted, though the intuition given in Section 3 should be enough
– the formal definition of F-Alloy, then they can read:
  – Section 4.1 providing the formal notation used to reason about Alloy
  – Section 5 defining the notion of functional Alloy module, a formalization of Alloy modules denoting transformations
  – Section 6.1, 6.2, 6.3 and 6.5 introducing the syntax of F-Alloy in terms of the syntax of Alloy and containing some definition of terms and properties later used in the paper
  – Section 7 defining the translational semantics of F-Alloy
  – Optionally Section 8 showing that the interpretation of F-Alloy transformations follows the semantics defined in Section 7

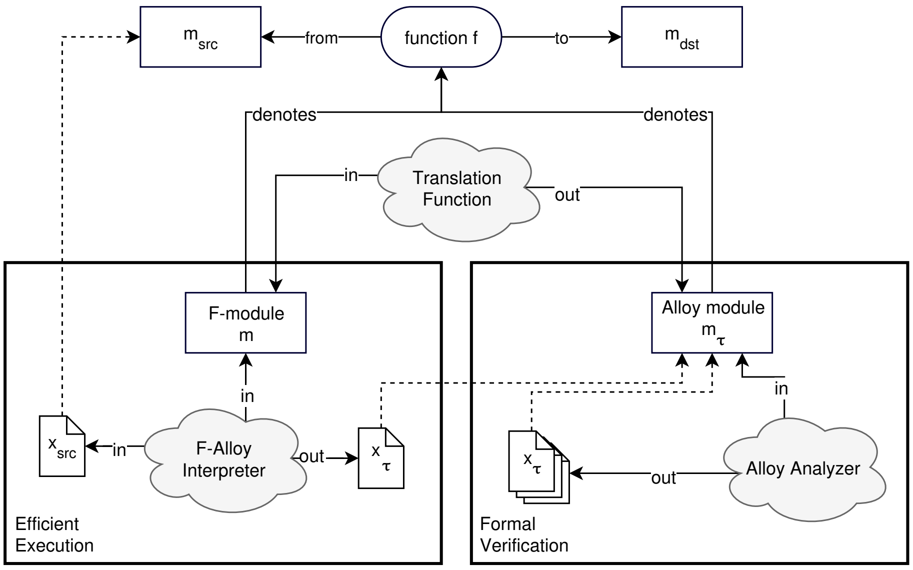


Fig. 1: An overview of the F-Alloy approach

– having a quick overview of the contributions, then they can read:
  – Section 5 defining the notion of functional Alloy module, a formalization of Alloy modules denoting transformations
  – Section 7.1 providing an overview of how the semantics of F-Alloy is defined
  – plots in Section 9 illustrating the performance of F-Alloy w.r.t. Alloy and existing model transformation languages
  – points of interest in Section 10 positioning our work w.r.t. related works

**Extension statement:** A previous version of this work has been published in the proceedings of the $8^{th}$ International Conference on Model Transformation ( ICMT 2015)[9]. Thanks to the valuable comments and suggestions from the anonymous reviewers of the ICMT 2015 conference and SoSyM journal, we hereby present an extended and improved version of our work. Notably, the main contributions with respect to the earlier publication are:

– a more general definition of the notion of functional Alloy modules,
– an extension of F-Alloy (both in terms of syntax and semantics) to endogenous transformations,
– a comparison of implementations of case studies in F-Alloy with those using existing model transformation languages, namely, Henshin and TGG,
– a more comprehensive evaluation by comparing the efficiency of F-Alloy based implementations with ATL- and Henshin-based implementations.

## 2 Approach Overview

Model transformations can be viewed as defining relations between models. We limit ourselves to the study of a subset of those relations we call functions. Functions have the property of being deterministic and of yielding, given a source model, at most one target model.

Following the line of work presented in [10], there is a need for the specification of such functions to be formally verifiable yet efficient computable. While seamless formal verification is natively supported by Alloy [17], making it a formalism of choice to express those specifications, the Alloy analyzer's performance, relying on SAT-solving, hinders the practical use of such specifications.

In this work we are interested in adapting Alloy to the specification of functions in such a way that these functions can be computed efficiently while the underlying specification can still be analyzed using Alloy's automatic analysis features. We hence define a new language, called F-Alloy[2], specifically designed to concisely express functions in a restricted Alloy syntax, that achieves the goals we just stated, i.e.:

– Functions expressed in F-Alloy can be efficiently computed (by a process called *interpretation*)
– Function specifications in F-Alloy (called *f-modules*) translate to an Alloy module denoting the same function (called *functional Alloy module*) thus allowing them to be subject to Alloy analysis.

[2] stands for Functional-Alloy as it is used to specify functions

To summarize, we provide a graphical overview of the approach in Fig. 1.

In this figure, we are interested in specifying a function $f$ from a source to a target metamodel. To do so, we produce an f-module $m$ "denoting" $f$. This f-module can be interpreted, given an instance $x_{\text{src}}$ conforming to $m_{\text{src}}$, to efficiently build the instance $x_{\mathcal{T}}$ corresponding to the execution of $f$ given $x_{\text{src}}$. The translational semantics of F-Alloy allows us to obtain the Alloy module $m_{\mathcal{T}}$, Alloy equivalent of $m$ in the sense that instances yielded by interpretation of $m$ conform to $m_{\mathcal{T}}$. Alloy analysis can then be performed on $m_{\mathcal{T}}$ (as shown in Fig.1) to obtain, for validation's purpose, a set of possible transformation executions (represented by the instance $x_{\mathcal{T}}$ in Fig. 1, bottom right).

## 3 Case Studies

We present in this section two model transformation scenarios to evaluate and illustrate the use of F-Alloy. The first one — referred to as `CD2RDBMS` — is an exogenous model transformation from Class Diagram to Relational Database Management System. The second one — referred to as `CDRefinement` — is an endogenous model transformation on Class Diagrams.

### 3.1 CD2RDBMS: A Class Diagram to Relational Database Management System Model Transformation

The `CD2RDBMS` model transformation is the defacto standard case study when it comes to benchmarking a model transformation language [7]. We first provide a summary of the specifications provided in [7], before showing how they can be implemented using *Triple Graph Grammars* (TGG). The TGG implementation is then used to give a first informal introduction to F-Alloy, by simply providing the F-Alloy implementation of `CD2RDBMS` and comparing it to the TGG implementation.

We note that we have chosen TGG for its popularity and ability in providing a concise graphical overview of the transformation.

#### *3.1.1 Informal Specification*

The source and target metamodels of this model transformation (`CD` and `RDBMS`, respectively) are shown as UML class diagrams in Fig. 2 and 3, respectively. Well-formedness constraints of those metamodels can be found in their respective Alloy representations, given in Listings 1 and 2.

We now give an informal specification of this transformation.

For each class $c$ without a parent, a table is created. This table is populated with columns (1) representing the attributes of $c$ or of its inheriting classes, (2) issued from associations having $c$ as their source.

In case (1), the column is typed and named after the represented attribute. If the class declaring the attribute has a parent, names of all the parents of the declaring class have to appear in the name of the representing column.

In case (2), a column is created for each primary key of the table representing the class at the destination of the association, and is named after the association and the attribute it represents. In this example, we flatten associations so that their sources and targets are topmost classes. This enables us to simplify our implementations (in TGG and F-Alloy) , without decreasing the complexity of the transformation (Recursion is still necessary when naming columns).

To better grasp the expected behavior of the transformation, we provide a visualization of a `CD2RDBMS` application in Fig. 4. We invite the reader to pay particular attention to the traceability links (dashed arrows). In this figure we see that two tables are created from the two topmost classes `A` and `B`. The columns `a` and `C_c` of table `A` and the column `b1` of table `B` are obtained as per rule (1). The column `x_b1` composes the foreign key of table `A`, representing association `x` and referring to table `B`, and is obtained as per rule (2).

#### *3.1.2 TGG Implementation*

TGG (Triple Graph Grammar)[24] is a well-known formalism allowing the declarative specification of model transformations. TGG specifications are composed of rules, each of them being expressed using three graphs. The source and target graph represents those subgraphs whose match will trigger the rule and those who will be generated when the rule is triggered, respectively. A third graph called correspondence graph keeps track of relations between source and target elements for future reference. In this work, we reuse the partial[3] implementation of the `CD2RDBMS` provided in [16]. We preferred this solution amongst others for the concise overview of the `CD2RDBMS` transformation it provides.

Figure 5 provides an overview of the solution and of the kind of traces (nodes of the correspondence graph) used to relate elements of `CD` (source graph) to elements of `RDBMS` (target graph).

The rules composing the transformation are given in Fig. 6. We note that those rules are bi-directional, *i.e.*,

[3] the notion of persistence is abstracted away and the naming of column make abstractions of inheritance

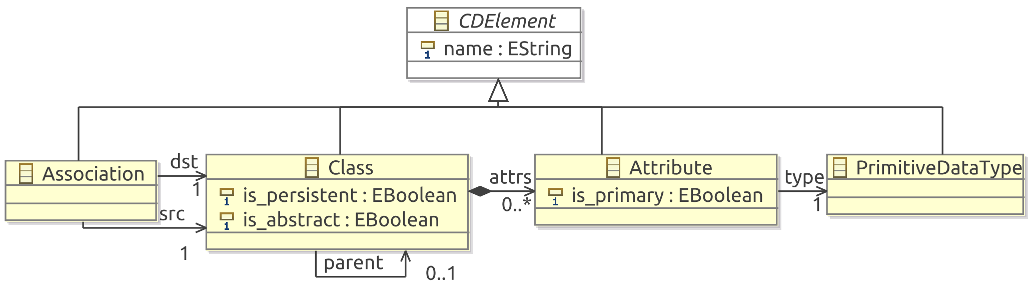


Fig. 2: CD metamodel (adapted from [7])

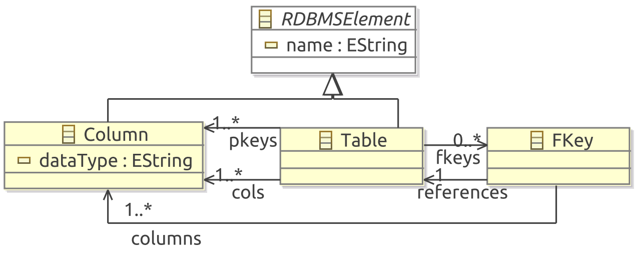


Fig. 3: RDBMS metamodel (adapted from [7])

they can be read either from left to right or right to left. We are only interested in going from `CD` to `RDBMS` (left to right) and thus provide a reading in that direction solely:

– `C2T` enforces, via the `CT` correspondence node, the creation of a table for every class so that table and related class share the same name.
– `SC2T` enforces inheriting classes to be related to the same table as their parents (via `CT` as well).
– `PA2C` enforces the creation of a primary column for each primary attribute of a class. The column is named and typed after the attribute it has been created from and is contained in the table related to the class declaring the attribute.
– `A2FK` enforces the creation of an FKey for each association and of a column for each primary column of tables related to associations' destination classes. The created columns are typed after those primary key columns' type and are named after those columns and the association that led to their creation. Those created columns are also composing the foreign key created from the association and referring to the table representing its destination class.

### *3.1.3 F-Alloy implementation*

The source and target metamodels of an F-Alloy model transformation are represented as Alloy modules. To implement the `CD2RDBMS` transformation in F-Alloy, it is thus a prerequisite step to express the `CD` and the `RDBMS` metamodels in Alloy. The `CD` and `RDBMS` Alloy modules are given in Listing 1 and 2, respectively. In these modules, we see that each concept is declared as a signature (`sig`) preceding a block containing declarations of the references and attributes related to that concept. These modules also contain several constraints explained in comments.

Given those CD and RDBMS modules, a possible F-Alloy implementation of the `CD2RDBMS` transformation is given in Listing 3. We explain it, basing ourselves on the TGG implementation given earlier, as follows:

The `CD2RDBMS` module (declared on l.1) expressing a model transformation from `CD` (imported on l.2) to `RDBMS` (imported on l.3) is composed of four `CREATE` mappings (l.6-9). CREATE mappings are used to "create" elements in the output of the transformation. A mapping declaration consists of a sequence of arrow-separated signatures, the last one being the type of elements the mapping can produce. Each mapping is associated to a guard and a value predicate declared using the keyword `pred` and identified by the name of the mapping they are associated with prefixed by `guard_` and `value_`, respectively. Guard predicates define the condition under which mappings are triggered. A guard predicate can thus be seen as a mapping precondition and serves thus the same purpose as TGG's

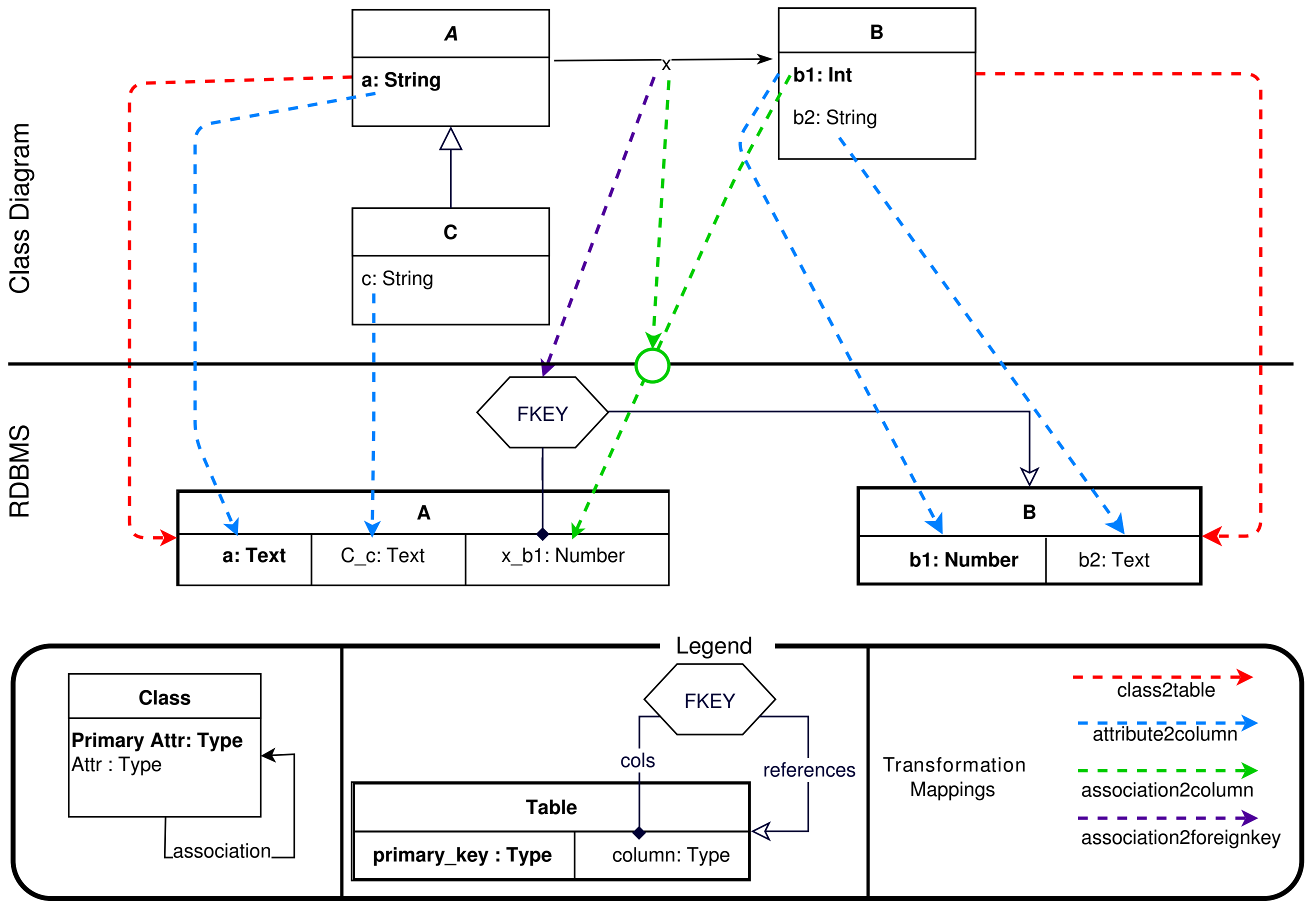


Fig. 4: Sample input and corresponding output of the CD2RDBMS transformation (adapted from [11])

left hand-side graph. Value predicates define the value of elements created by mappings. A value predicate can thus be seen as a mapping postcondition and serves thus the same purpose as TGG's left hand-side graph.

We detail here the purpose of each mapping and make a parallel with the previously listed TGG rules:

1. `class2table`: creates a `Table` for each top-most `Class` (l.13) so that the `Table` is named after the name of the `Class` (l.16). This mapping covers the `C2T` rule defined in TGG.
2. `attribute2Column`: creates a `Column` for each `Attribute` (as guard l.19 is empty). The `Column` is typed (l.21) and named (l.22) after the `Attribute`. The name of the `Column` also contains the name of the `Classes` in the inheritance hierarchy of the `Class` declaring the `Attribute` if this latter has a parent (l.24-26). Note that this requirement of including the name of inherited classes in the column's name is lacking in the provided TGG implementation. The `Column` has to be in the set of columns of the `Table` representing the top-most `Class` affiliated to the `Attribute` (l.29), and should also be in the set of primary keys of the said `Table` if the `Attribute` is primary (l.27-28). This mapping covers the `PA2C` rule defined in TGG.
3. `association2column`: creates a `Column` for each primary `Attribute` of a given `Association`'s target `Class` (l.33). The `Column` is typed after the `Attribute` (l.36) and named after both the `Association` (l.37) and the `Attribute` (l.38). The `Column` has to be in the set of columns of the `Table` representing the `Class` at the source of the `Association` (l.39) and in the set of columns composing the foreign key (`FKey`) obtained via the `association2FKey` mapping from the given `Association` (l.40). The `association2column` mapping covers the `Column` creations of the `A2FK` rule.
4. `association2FKey`: creates an `FKey` for each `Association` (l.43). The `FKey` references the `Table` representing the `Association`'s target `Class` (l.45), and is in the set of foreign keys of the `Table` associated via the `class2table` mapping to the class at the source of the `Association` (l.46).

```
module CD
open util/boolean

abstract sig CDElement{
//no 2 distinct CDElement share the same name
  name: disj  String
}
sig Class extends CDElement{
  attrs: some Attribute,
  parent: lone Class,
  is_abstract: Bool
}{
//no 2 distinct class share the same attributes
// & in Alloy is the set intersection operator
  no c:Class | c!=this and c.@attrs & attrs !=none
//class cannot be its own ancestor
  this not in this.^@parent
}
sig Attribute extends CDElement{
  is_primary: Bool,
  type:PrimitiveDataType
}{
//no orphan attributes
  this in Class.attrs
}
sig Association  extends CDElement{
  src: Class,
  dest: Class
}{
//association between top-classes only
  (src+dest).parent=none
}
sig PrimitiveDataType extends CDElement{}{
//primitiveDataType is either named String or int
  PrimitiveDataType.@name= "String"+"int"
}
```

Listing 1: CD Alloy module

```
module RDBMS

abstract sig RDBMSElement{
  name: disj seq String
}
sig Table extends RDBMSElement{
  cols: disj some Column,
  pkeys: some Column,
  fkeys: set FKey
}{
//pkeys is a subset of cols
  pkeys in cols
}
sig Column extends RDBMSElement{
  dataType: String
}{
//no orphan columns
  this in Table.cols
//dataType is either text or number
  dataType in "TEXT"+"NUMBER"
}
sig FKey{
  references: Table,
  columns: set Column
}{
//no orphan FKeys
  this in Table.fkeys
//fkey columns are in owning table
  columns in this.~fkeys.cols
}
```

Listing 2: RDBMS Alloy module

```
module  CD2RDBMS
open  CD
open  RDBMS

one sig CREATE{
  class2table: Class -> Table,
  attribute2column: Attribute -> Column,
  association2column: Association -> Attribute -> Column,
  association2FKey: Association -> FKey,
}

pred guard_class2table(c:Class){
  c.parent= none
}
pred value_class2table(c:Class ,  t:Table){
  t.name[0]= c.name
}

pred guard_attribute2column(a:Attribute){}
pred value_attribute2column(a:Attribute,  c:Column){
  c.dataType= (a.type.name="String" implies "TEXT" else "NUMBER")
  c.name[0]= a.name
  c.name[1]=
    ((a.~attrs.parent)!=none implies a.~attrs.name else none)
  all i:Int| i>=1 and i< #(a.~attrs.^parent) implies
    c.name[add[i,1]]= c.name[i].~name.parent.name
  a.is_primary= True implies
    c in CREATE.class2table[a.~attrs.*parent].pkeys
  c in CREATE.class2table[a.~attrs.*parent].cols
}

pred guard_association2column(ass:Association, att:Attribute){
  att.is_primary= True and att in ass.dest.attrs
}
pred value_association2column(ass:Association, att:Attribute, c:Column){
  c.dataType= (att.type.name="String" implies "TEXT" else "NUMBER")
  c.name[0]= ass.name
  c.name[1]= att.name
  c in CREATE.class2table[ass.src].cols
  c in CREATE.association2FKey[ass].columns
}

pred guard_association2FKey(a:Association){}
pred value_association2FKey(a:Association, f:FKey){
  f.references= CREATE.class2table[a.dest]
  f in CREATE.class2table[a.src].fkeys
}
```

Listing 3: F-Alloy specification of the `CD2RDBMS` transformation

The `association2FKey` mapping covers the `FKey` creations of the `A2FK` rule.

We note that `SC2T` is not represented as a mapping in F-Alloy as it is not used to create any element in the output. The use of `SC2T` is to relate inheriting classes to the table representing their topmost ancestors. In F-Alloy we use the expression $c$`.*parent` (where $c$ is an expression of type class) to return the set of all parents of $c$. The expression `Create.class2table[`$c$`.* parent]` then returns the table associated to the ancestors of $c$, just as defined by `SC2T`.

Figure 4 shows traces corresponding to each of the aforementioned mappings. In this figure, we can see amongst other details that tables `A` and `B` are created

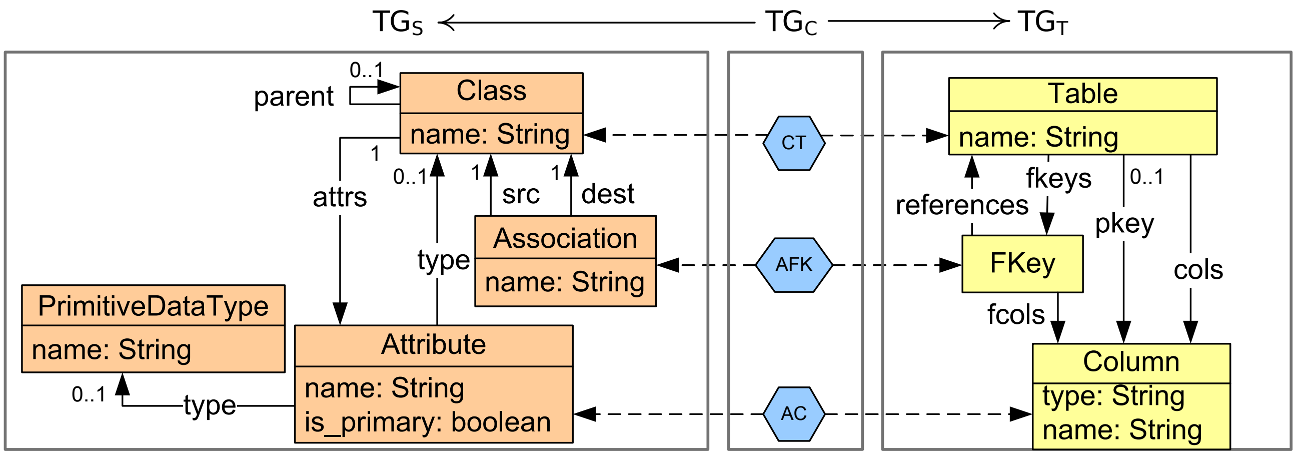


Fig. 5: A simplified overview of the CD2RDBMS TGG transformation as given in [16]

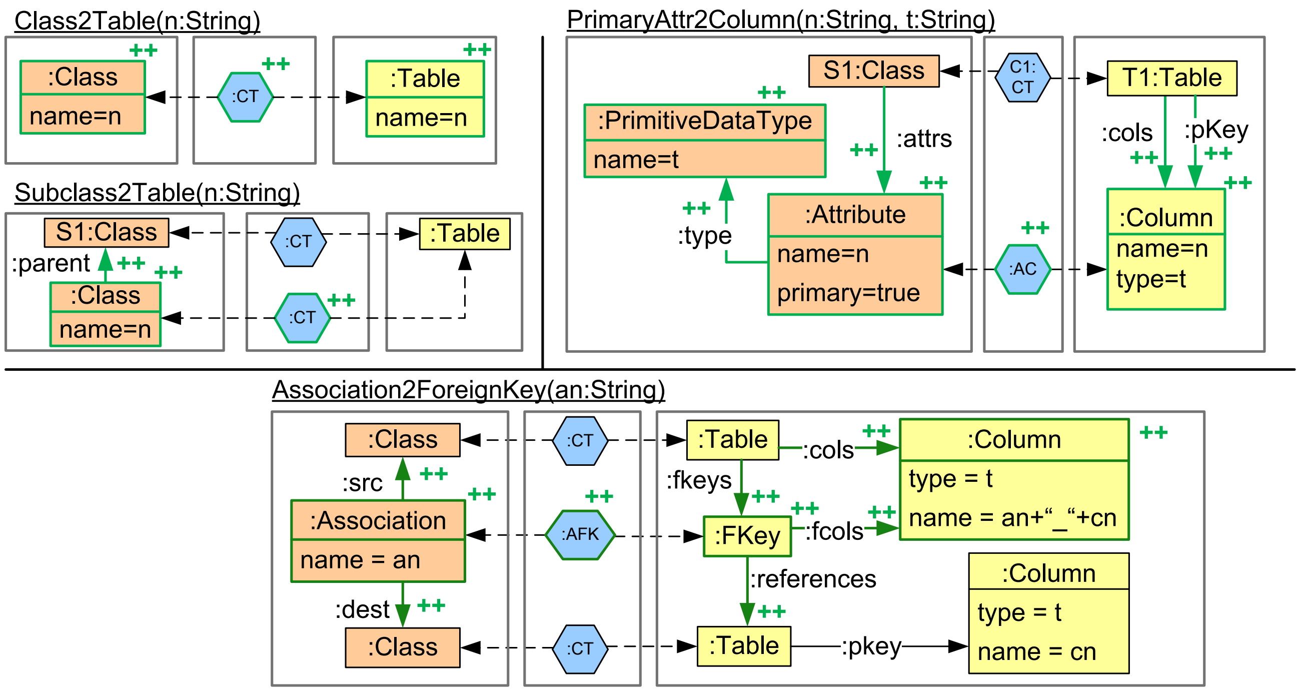


Fig. 6: TGG Rules composing the CD2RDBMS transformation proposed in [16])

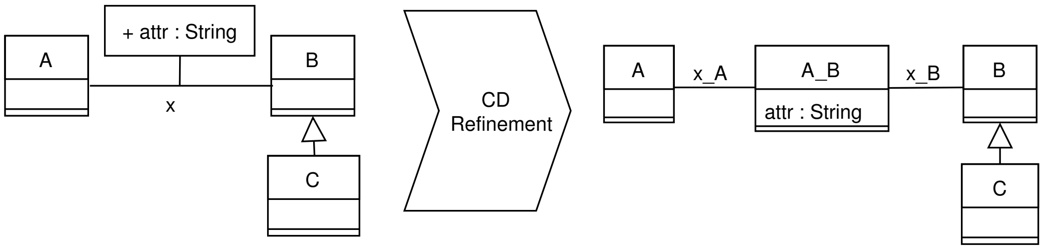


Fig. 7: Sample input and corresponding output of the CDRefinement transformation

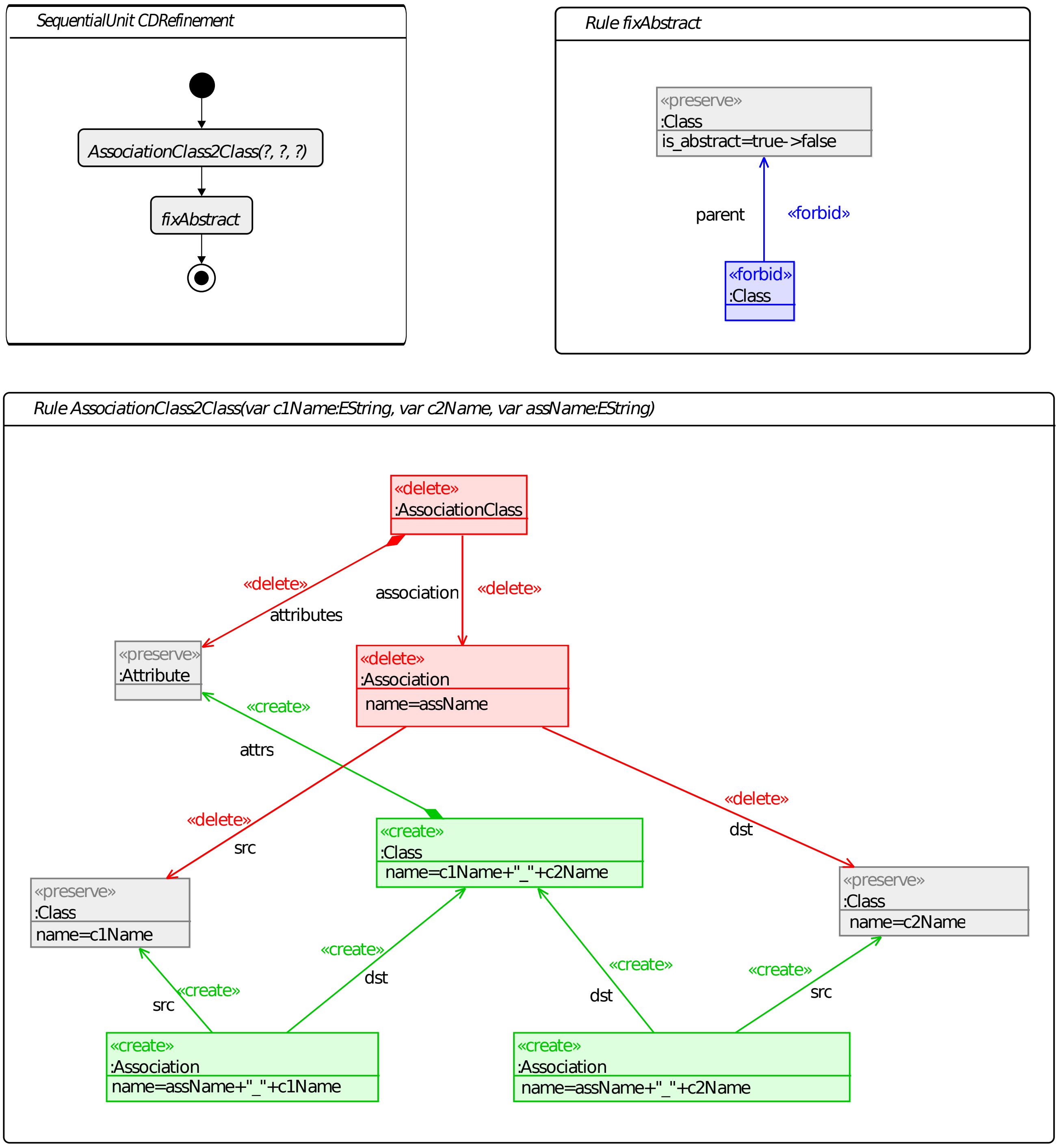


Fig. 8: CDRefinement transformation defined using Henshin

from classes `A` and `B` via the `class2table` mapping, respectively. No table `C` is created from class `C`, as it inherits class `A`. Column `a` , `b1` , `b2`, and `C_c` are all created from the `attribute2column` mapping. Column `x_b1` composing the foreign key created by the `association2FKey` mapping is created by the `association2column` mapping.

## 3.2 CDRefinement: A Class Diagram Refinement Scenario

### *3.2.1 Informal Specification*

The `CDRefinement` endogenous model transformation is the specification of a refinement transformation ap-

```
sig AssociationClass {
  disj association: Association,
  associationAttrs: set Attribute,
}
```

plied to class diagrams with association classes. This refinement transformation consists in:

1. replacing all association classes by regular classes
2. turning non-inherited abstract classes into non-abstract classes.

The purpose of (1) is to adapt the source model to languages in which association classes cannot be represented (*e.g.* Ecore), while (2) adapts the source model to languages where non-inherited abstract classes are not supported as such (*e.g.* in Alloy, non-inherited abstract classes are considered non-abstract).

The manipulations to achieve (1) consist (as described in [20]) in removing each association class and its associated association, and in creating a new class containing the same attributes than the removed association class as well as two new associations, both connecting the originally associated classes to the newly created class. To achieve (2) it is sufficient to update the `abstract` modifier of concerned classes. We choose this case study because it exercises the three elemental operations commonly used in endogenous model transformation, i.e., creation, deletion and update of elements (see [22]).

The source (and target) metamodel of this model transformation is the class diagram model given in Listing 1 extended with the concept of `AssociationClass` (an entity linked to an association and composed of attributes) declared in Alloy as follows :

We note that `AssociationClass` does not inherit from `CDElement` as it has no name attribute. The application of the `CDRefinement` transformation is illustrated in Fig. 7. In this figure, we see that the abstract class `A` becomes non-abstract due to the absence of inheriting classes while the abstract class `B` stays abstract as class `C` is inheriting `B`. We also see that the association class linked to association `x` is replaced by a class named `A_B` after the source and target of `x`, and that this class is associated through two new associations to the source and target classes of `x`.

### *3.2.2 Henshin Implementation*

We now provide an Henshin [5] implementation of the `CDRefinement` model transformation. We have chosen the graph based model transformation language Henshin over other languages for its popularity and intuitive graphical syntax. This graphical syntax, to the contrary of TGG's one, enables the concise depiction of in-place operations, notably through the use of colors and labels.

A possible Henshin implementation of our case study is given in Fig. 8.

Two rules and a *sequentialUnit* defining in which order the rules are to be executed are depicted in Fig. 8. The first rule, namely `AssociationClass2Class`, states that every `AssociationClass` and connected association are to be deleted. It also enforces that for each such deletion, a new class is created, containing those attributes declared in the deleted `AssociationClass`, and named after the two previously associated classes. Two associations are created as well, each connecting one of the previously associated classes to the newly created class. Those associations are named after the deleted association and the classes they are associating to the new class.

The second rule, namely `fixAbstract`, simply changes the value of `is_abstract` from true to false, for each class that has no children.

### *3.2.3 F-Alloy implementation*

A possible F-Alloy implementation of the `CDRefinement` transformation is given in Listing 4.

We explain it, basing ourselves on the Henshin implementation given earlier, as follows:

The `CD2Refinement` module (l.1) describes an endogenous transformation as it imports only one module (`CD`)(l.2). It is composed of the three signatures `CREATE`, `UPDATE` and `DELETE`. Mappings declared in the `CREATE` and `DELETE` signatures are meant to add and remove new elements to/from the source model, respectively. Mappings declared in the `UPDATE` signature are meant to modify how existing elements relate to each other.

The `CREATE` signature contains two mappings. The first, `associationClass2Class`, enforces the creation of a new `Class` for any `AssociationClass` present in the source model(l.9). This newly created `Class` is named (l.11) after the `AssociationClass` it represents and contains the same attributes (l.12).

The second one, `newAssociations`, ensures the creation of an `Association` for each combination of `Class` and `Association` $(c, a)$ present in the source model with $a$ linked to $c$ and adorned by an `AssociationClass` [4] (l.16). The created `Association` is named "$a\_c$"(l.19) and then linked to either $c$ or the `Class` replacing the `AssociationClass` previously adorning $a$

[4] `~association` is the inverse relation of `association` mapping an `AssociationClass` to an `Association`. The expression `a.~association` hence returns the `AssociationClass` adorning association `a`.

```
1  module CDRefinement
2  open CD
3
4  one sig CREATE{
5    associationClass2Class: AssociationClass -> Class,
6    newAssociations: Class -> Association -> Association
7  }
8
9  pred guard_associationClass2Class(a:AssociationClass){}
10 pred value_associationClass2Class(a:AssociationClass, y:Class){
11   y.name= a.association.name
12   y.attrs= a.attributes
13 }
14
15 pred guard_newAssociations(c:Class, a:Association){
16   c in a.(src+dest) and a.~association!=none
17 }
18 pred value_newAssociations(c:Class,a:Association,y:Association)
       {
19   y.name= a.name+c.name
20   y.src= (c= a.src implies c else CREATE.associationClass2Class
       [a.~association])
21   y.dest= (c= a.dest implies c else CREATE.
       associationClass2Class[a.~association])
22 }
23
24 one sig UPDATE{
25   fixAbstract: Class -> Class
26 }
27
28 pred guard_fixAbstract(c:Class){
29   c.is_abstract= True and c.~parent= none
30 }
31 pred value_fixAbstract(c:Class, y:Class){
32   y.is_abstract= False
33 }
34
35 one sig DELETE{
36   associationWithClass: Association,
37   associationClass: AssociationClass
38 }
39
40 pred guard_associationWithClass(a:Association){
41   a.~association!= none
42 }
43 pred guard_associationClass(a:AssociationClass){}
```

Listing 4: F-Alloy specification of the `CDRefinement` transformation

depending on whether the source association was pointing to or was coming from an `AssociationClass` (l.20-21).

The previously adorned associations as well as all the association classes are removed from the source model as specified by the two `DELETE` mappings (l.36-37).

All those aforementioned mappings express the Henshin rule `AssociationClass2Class`.

The Henshin rule `fixAbstract` is represented by the `UPDATE` mapping of the same name.

The mapping `fixAbstract` simply enforces that each abstract class without children (l.29) should have their `is_abstract` field set to `False` (l.32). We note that other fields of `Class` do not appear in the body of `value_fixAbstract` as they are meant to remain unchanged.

## 4 Background on Alloy

We provide in this section a formal definition of the basic concepts of Alloy used throughout the paper following by a comparison of those concepts with their Ecore counterparts. While the first part allows us to lay down the formal background needed to reason later on about Alloy, the second part is provided as a way to put the given definitions in the MDE perspective, specifically using Ecore as reference.

### 4.1 An Alloy Formal Introduction

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 $(S, F, \varphi)$ 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 subsignatures of other signatures (using the `extends` keyword). Fields of $F$ have as type a sequence of signatures in $S$, the first one being the signature that contains it. $\varphi$ is a first-order logic formula (possibly containing the transitive closure operators[5] ˆ and *) representing the set of constraints, called facts, expressed in the module.

Alloy modules representing the `CD` and `RDBMS` metamodels of the case study presented in Section 3 are given in Listings 1 and 2, respectively. We note that the field `name` in the `RDBMSElement` signature is declared as a sequence of string to enable string concatenation, string operations not being supported by Alloy.

The `RDBMS` module presented in Listing 2 could thus be rewritten following definition 1 as $m = (S, F, \varphi)$ with[6]:

$S = \{$ `Table` , `Column` , `FKey` , `RDBMSElement` $\}$
$F = \{$ `cols`: (`Table` , `Column`)
`pkeys`: (`Table` , `Column`) ,
`fkeys`: (`Table` , `FKey`) ,
`type`: (`Column` , `String`) ,
`references`: (`Fkey` , `Table`) ,

[5] $\hat{}R$ returns the smallest relation $R'$ containing $R$ and being transitive while $^*R$ returns the smallest relation $R'$ containing $R$ and being both transitive and reflexive

[6] We have omitted the constraints that express multiplicities and disjointedness in $\varphi$ – e.g. that each table should have at least one column, or that no two `RDBMSElement` should have the same name.

$\quad$ `columns`: (`FKey` , `Column`) ,
$\quad$ `name`: (`RDBMSElement` , `Int`, `String`)}
$\varphi =$ ( ∀ t:`Table` , `pkey`(t) ∈ `cols`(t) ) ∧
$\quad$ ((∀ c: `Column` , ∃ t: `Table` , c ∈ `cols`(t))) ∧
$\quad$ (∀ f: `FKey` , ∃ t: `Table`, f ∈ `fkeys`(t)) ∧
$\quad$ (∀ f:`FKey`, `columns`(f) ⊆ `cols`(`references`(f)))

We note that in this example we express $\varphi$ as a conjunction of first order logic clauses listed following the order of appearance of the fact they represent in the `RDBMS` module

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

**Definition 2 (Alloy Instance)** An Alloy instance of an Alloy module $m$, also called m-instance, is a triplet $x = (X, Y, m)$ 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 $m$ and composed of atoms in $X$. We write $x \vDash \varphi$ if an instance $x$ of $m$ satisfies $\varphi$ and call valid instances of $m$ the subset of instances of $m$ which satisfy $\varphi$[7]. We denote the set of valid instances of $m$ by $I(m)$. Formally:

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

An instance $(X, Y, m)$ is an *(m-)sub-instance* of $(X', Y', m')$ if $X \subseteq X'$ , $Y \subseteq Y'$ and $m'$ is equal to $m$ or $m'$ imports $m$.

As an example, let us consider the table named "A" depicted in Fig. 4. The Alloy instance corresponding to that table is given in Fig. 9. To illustrate the formalism introduced in Definition 2 without providing an uselessly verbose textual representation of the instance, we simply note that the set of atoms $X$ contains, amongst other atoms, $\texttt{Column2}^{\texttt{Column}}$ and $\texttt{"A"}^{\texttt{String}}$; that the set of tuples $Y$ contains $(\texttt{Table1}^{\texttt{Table}}, \texttt{Column1}^{\texttt{Column}})^{\texttt{cols}}$ and $(\texttt{Column3}^{\texttt{Column}}, \texttt{1}^{\texttt{Int}}, \texttt{"b1"}^{\texttt{String}})^{\texttt{name}}$, amongst other tuples; and that the module $m$ is the RDBMS module as defined in the previous illustration of Alloy modules.

Alloy comes with its dedicated tool, the Alloy analyzer, enabling the automatic analysis of Alloy modules. This analysis returns for a given Alloy module $m$ the subset of valid instances of $I(m)$ that fit within the given scope.

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

### 4.2 An Informal Alloy Introduction Based on Ecore

For those readers familiar with Ecore [27], we propose to draw a parallel between the previously defined Alloy notions and Ecore concepts:

– Alloy modules corresponds to Ecore metamodels enhanced with OCL in the sense that they are entities used to define static constructs and their well-formedness rules.
– Signatures correspond to EClass in the sense that they allow the definition of concepts. Signatures can also be abstract and inherited. Signatures, to the contrary of EClasses, do not support multiple inheritence per-se but support the broader notion of set inclusion: each signature representing a set of atoms in Alloy, a signature can be defined as a subset of several others. Note that when a signature $A$ extends a signature $B$ in Alloy it does not only enforce that atoms in $A$ are contained in $B$. It also enforces that the set of atoms defined by $A$ is disjoint from any other set of atoms defined by signatures extending the same class as $A$ (in short, inheritance declared using the `extend` keyword is a stronger notion than set inclusion defined using the `in` keyword).
– Fields are more expressive than any corresponding EStructuralFeature (EReference, EAttribute, ...) in the sense that they allow the definition of relations of any arity. Yet a field has no properties as opposed to, *e.g.*, EReferences, hence properties like containement or EOpposite are to be defined through constraints.
– Alloy Instances are comparable to Ecore instantiations represented by an XMI file. Saying that an instance is conforming to a module, or calling an instance a valid instance of a given module is similar as saying that an Ecore instance conforms to a given Ecore model.
– Atoms and Tuples are elements of an Alloy instance just like objects and links are of an Ecore instance.

## 5 Functional Alloy Modules

In this section, we introduce *functional Alloy modules* as a way to define transformations in Alloy.

We have seen earlier that the Alloy language is well suited to express relations between concepts through the declaration of fields.

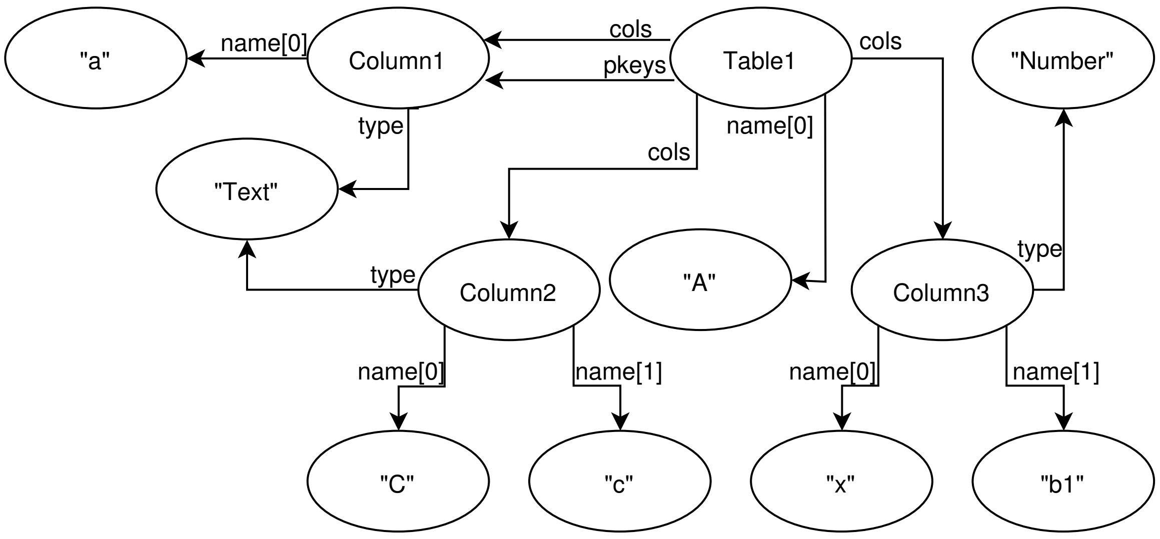


Fig. 9: The Alloy instance corresponding to the "A" table depicted in fig. 4, visualized as a graph whose nodes and edges are its composing atoms and tuples, respectively.

A natural approach to define a transformation from a source module $m_{\text{src}}$ to a target module $m_{\text{dst}}$ as an Alloy module $m$ is thus to declare, in a signature, fields relating signatures of $m_{\text{src}}$ to signatures of $m_{\text{dst}}$. Those fields, that we call *mappings* from now on, should then be suitably constrained so that in any valid $m$-instance, the presence of certain $m_{\text{src}}$-elements (atoms and tuples typed by signatures or fields declared in $m_{\text{src}}$) enforce the presence of their expected images (w.r.t. the transformation defined). This approach is illustrated in Fig. 10. Given such an Alloy module $m$, executing the transformation it defines on an $m_{\text{src}}$-instance $x_{\text{src}}$ consists in using the Alloy analyzer to find $m$-instances in which the $m_{\text{src}}$-sub-instance is $x_{\text{src}}$.

We illustrate the previously used notation by noting that in our case studies $m$ denotes the modules in which the `CD2RDBMS` and `CDRefinement` transformations are defined, respectively. In the context of both transformations, $m_{\text{src}}$ denotes the `CD` module. $m_{\text{dst}}$ denotes the `RDBMS` module in the context of the `CD2RDBMS` transformation and the `CD` module in the context of the `CDRefinement` transformation.

To formally define the kind of model transformation expressible in Alloy, we introduce the notion of *transformation functions* which may be viewed as the mathematical representation of model transformations expressed in terms of Alloy instances.

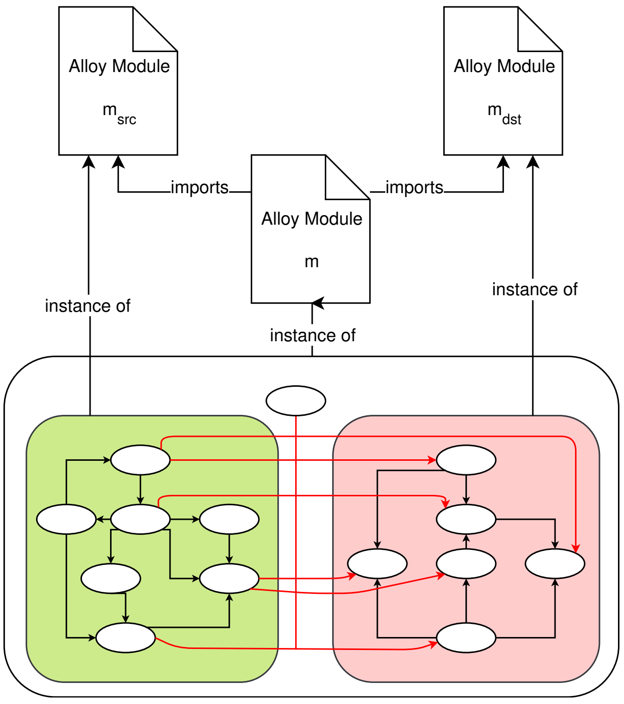


Fig. 10: Illustration of the approach to represent model transformation in Alloy

**Definition 3 (Transformation Function)** Let $m$ and $m'$ be two Alloy modules. A transformation function $f$ from $m$ to $m'$, noted $f : I(m) \rightarrow I(m')$, is a function that takes as input a valid instance of $m$ and returns as output a valid instance of $m'$.

As motivated in the introduction, we are interested in formalizing those Alloy modules that embody functions, *i.e.*, deterministic single-output model transformations. We call those Alloy modules *functional Alloy modules* and define them formally as follows:

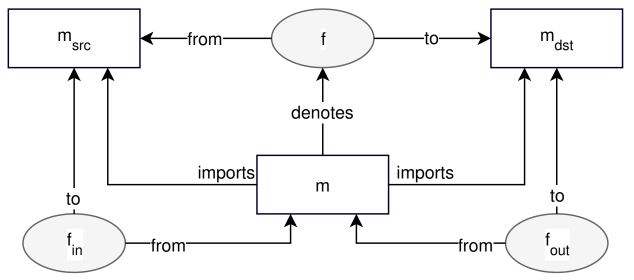


Fig. 11: Illustration of the usage of transformation functions w.r.t. the definition of functional Alloy modules

**Definition 4 (Functional Alloy Module)** Let $m$, $m_{\text{src}}$ and $m_{\text{dst}}$ be Alloy modules[8] and let $f_{\text{in}} : I(m) \to I(m_{\text{src}})$ and $f_{\text{out}} : I(m) \to I(m_{\text{dst}})$ be two transformation functions. We say that $m$ is a functional Alloy module with respect to $f_{\text{in}}$ and $f_{\text{out}}$ if the following holds:

$$\forall x, x' \in I(m), f_{\text{in}}(x) = f_{\text{in}}(x') \implies f_{\text{out}}(x) = f_{\text{out}}(x')$$

We then say that $m$ specifies a (possibly partial) transformation function $f : I(m_{\text{src}}) \to I(m_{\text{dst}})$, such that:

$$\forall x \in I(m), f(f_{\text{in}}(x)) = f_{\text{out}}(x)$$

In the previous definition, functions $f_{\text{in}}$ and $f_{\text{out}}$ return, given an $m$-instance, the $m_{\text{src}}$ and $m_{\text{dst}}$ sub-instance corresponding to the input model and output model of the transformation expressed in $m$, respectively. We illustrate the use of those functions in Fig. 11.

We note that the nature of the transformation (endogenous or exogenous) influences the way those input and output models are obtained from an $m$-instance. Those transformation functions are defined for both endogenous and exogenous transformations in Sections 7.3.1 and 7.3.2, respectively.

## 6 Syntax of F-Alloy

This section aims at providing a formal yet comprehensive introduction to the F-Alloy syntax. To achieve this goal, and because F-Alloy is based on Alloy, we first introduce the syntax of Alloy, that will then be reused in the definition of F-Alloy's syntax. We finish the section by giving the purpose of each F-Alloy syntactic construct defined and by showing that those constructs are valid Alloy constructs.

[8] with the possibility that $m_{\text{src}} = m_{\text{dst}}$

### 6.1 Alloy's Syntax

The grammar of Alloy is given in Listing 5, as provided in [17]. To avoid confusion and for this work to be self contained, we list the remarks given in [17] concerning the notation used and invite readers unfamiliar with BNF notations to go through it before proceeding to the grammar.

"The grammar uses the standard BNF operators:

– $x^*$ for zero or more repetitions of $x$;
– $x^+$ for one or more repetitions of $x$;
– $x|y$ for a choice of $x$ or $y$;
– *[x]* for an optional $x$.

In addition,

– $x,^*$ means zero or more comma-separated occurrences of $x$;
– $x,^+$ means one or more comma-separated occurrences of $x$;

To avoid confusion, potentially ambiguous symbols – namely parentheses, square brackets, star, plus and the vertical bar – are set in bold type when they are to be interpreted as terminals rather than as meta symbols. The string name represents an identifier and number represents a numeric constant, ..."[17].

### 6.2 F-Alloy's Syntax

We now introduce the syntax of F-Alloy by giving a BNF description in Listing 6.

We split the BNF definition of F-Alloy into two parts in order to ease its understanding. While the first part reveals the structure of *f-modules* – i.e., modules expressed in F-Alloy – the second part defines the subset of boolean-valued Alloy expressions, called *rules*. We bring the reader's attention to the fact that the syntactical constructs `expr`, `name`, `qualName`, `decls`, `paraDecls`, `moduleDecl`, `import` and `block` present in the BNF are coming from the Alloy BNF given in Listing 5.

The presented F-Alloy BNF allows the expression of both endogenous and exogenous specifications. We note the presence of `UPDATE` and `DELETE` signatures, which together with `CREATE` are used to define endogenous transformation following a standard CRUD approach [29], namely by enabling the specification of mapping enforcing the creation, update and deletion of elements in a given instance.

The nature of an f-module specification is determined by the number of open statements it contains (one for endogenous, two for exogenous) and by the

```
alloyModule::= [moduleDecl] import* paragraph*
moduleDecl::= module qualName [[ name,+]]
import::= open qualName [[ qualName,+]] [ as
     name]
paragraph::= sigDecl | factDecl | predDecl
       | funDecl | assertDecl | cmdDecl
sigDecl::= [abstract] [mult] sig name,+ [sigExt]
     { decl,*} [block]
sigExt::= extends qualName | in qualName
     [+qualName]*
mult::= lone | some | one
decl::= [disj] name,+: [disj] expr
factDecl::= fact [name] block
predDecl::= pred [qualName.]name[paraDecls] block
funDecl::= fun [qualName.]name[paraDecls] : expr
     {expr}
paraDecls::= ( decl,*)|[ decl,*]
assertDecl::= assert [name] block
cmdDecl::= [name :] [run|check]
     [qualName|block][scope]
scope::= for number [ but typescope,+]
      | for typescope,+
typescope::= [exactly] number qualName
expr::= const | qualName | @name | this
      | unOp expr | expr binOp expr
      | expr arrowOp expr | expr[expr,*]
      | expr [!|not] compareOp expr
      | let letDecl,+ blockOrBar
      | quant decl ,+ blockOrBar
      | {decl,+ blockOrBar}
      | (expr)| block
const::= [-] number | none | univ | iden|
unOp::= ! | not | no | mult | set | # | ~ | * | ^
binOp::= || | or | && | and | <=> | iff | => |
     implies
      | & | + | - | ++ | <: | :> | .
arrowOp::=[mult|set] -> [mult|set]
compareOp::= in | = | < | > | =< | >=
letDecl::= name = expr
block::= {expr*}
blockOrBar::= block | bar expr
bar::= |
quant::= all | no | sum | mult
qualName::= [this/] (name/)* name
```

Listing 5: Alloy BNF as given in [17]

```
fmodule::= moduleDecl import+ fparagraph*
fparagraph::= fsigDecl | guardDecl | valueDecl
fsigDecl::= one sig sigName { mappingDecl,*}
sigName::= CREATE | UPDATE | DELETE
mappingDecl::= name : qualName(->qualName)*,
guardDecl::= pred guard_name[paraDecls] block
valueDecl::= pred value_name[paraDecls]  rBlock
rBlock::= {rule*}

rule::= strict | loose | step | conditional
conditional::= expr implies rule
strict::= name.name[[expr]]= expr
loose::= name in image.name[[expr]]
image::= sigName.name[expr]
step::=  all i:Int| range implies
     name.name[add[i,1]] = expr
range::= i (>|>=) expr and i (<|<=) expr
```

Listing 6: F-Alloy BNF

presence or absence of the UPDATE and DELETE signatures (those are only allowed in endogenous specifications). We generally call f-module from $m_{\text{src}}$ to $m_{\text{dst}}$ any f-module defining a transformation from $m_{\text{src}}$ to $m_{\text{dst}}$.

## 6.3 A Formal Definition of Mappings

Before illustrating the usage of the previously introduced F-Alloy's syntax, we provide a formal definition to mappings and their surrounding concepts as we often refer to those in the remaining sections.

**Definition 5 (Mappings, domain, range)** In an f-module $m$ defining a model transformation from $m_{\text{src}}$ to $m_{\text{dst}}$, we call the fields relating signatures of $m_{\text{src}}$ to signatures of $m_{\text{dst}}$ *mappings*. Considering a mapping of the form `map: X1 -> ... -> Xn -> Y` the *domain* denotes the tuple of signatures $(\texttt{X}_1, \ldots, \texttt{X}_n)$ and the *range* denotes the signature `Y`. Mappings declared in a DELETE signature being of the form `map: X`, they do not have range.

To illustrate this definition, let us consider the following mappings:

- from the `CD2RDBMS` transformation given in Listing 3:
  - `association2column : Association -> Attribute -> Column`
    The domain of `association2column` is the tuple of signature `(Association, Attribute)` and the range is the signature `Column`.
- from the `CDRefinement` transformation given in Listing 4:
  - `associationClass2Class : AssociationClass -> Class`
    The domain of `associationClass2Class` is the signature `AssociationClass` and the range is the signature `Class`.
  - `associationClass : AssociationClass`
    The domain of `associationClass` is the signature `AssociationClass`. This mapping being declared in the `DELETE` signature, it does not have any range.
  - `fixAbstract : Class -> Class`
    The domain and range of `fixAbstract` is the signature `Class`.

## 6.4 F-Alloy's Syntax Usage

To help the reader understand why F-Alloy's syntax is defined the way it is, and to give a short introduction on how the syntax is used, we list in the following itemization the intentions behind each concept declared in Listing 6, and exemplify their usage with excerpts of the CD2RDBMS and CDRefinement case studies given in Listing 3 and 4:

– f-module: Any specification written in F-Alloy is called an F-module and consists of an Alloy module declaration (moduleDecl, identifying the module), some import declarations corresponding to the transformation's source and target metamodels (expressed in Alloy modules) and of a body composed of several fparagraph.
  Example: The CDR2RDBMS specification given in Listing 3 is a syntactically valid f-module.
– fparagraph: A paragraph in F-Alloy takes either the form of a signature declaration (fsigDecl), a guard declaration (guardDecl) or a value declaration (valueDecl). Guards and values are both Alloy predicates.
  Example: The CDR2RDBMS specification given in Listing 3 is composed of 9 fparagraph (a CREATE signature, four guard and four value predicates).
– fsigDecl: In F-Alloy, signatures are declared as singleton (one) and have as sole purpose to act as container for the mappings (mappingDecl) composing the transformation. In exogenous transformations, only one signature, named CREATE is allowed. In endogenous transformations, there are three signatures named CREATE, UPDATE and DELETE after the different kind of operations used in endogenous transformations:
  – CREATE mappings are used to express the creation of atoms typed by their range for given tuples typed by their domain.
  – UPDATE mappings have their domain and range bound to be the same signature. They are used to express a substitution: if an atom $a$ is mapped via an update mapping to an atom $b$, then $a$ will be replaced by $b$. (all the tuples referring to $a$ will then refer to $b$ instead).
  – DELETE mappings do not have a range and are used to express that an element is to be deleted from an instance.

  Example: The following excerpt from Listing 3 consists of a signature declaration and four mapping declarations.

```
one sig CREATE{
  class2table: Class -> Table,
  attribute2column: Attribute -> Column,
  association2column: Association -> Attribute -> Column,
  association2FKey: Association -> FKey,
}
```

– guardDecl: A guard predicate contains the precondition under which a mapping is to be triggered. The sequence of parameters it takes (paraDecls) corresponds to the domain of the mapping it is associated to. The association between mapping and guard predicate is done by name, *i.e.*, the guard of a mapping called $g$ will be named "guard_$g$".
  Example: The following excerpt from Listing 3 shows the guard predicate of the association2column mapping. We can see that the types of the declared parameters ass and att match the type of the signatures in the domain of the association2column mapping.

```
pred guard_association2column(ass:Association, att:
     Attribute){
  att.is_primary= True and att in ass.dest.attrs
}
```

– valueDecl: A value predicate contains a set of rules (rule) defining the values of fields of the elements created by the mapping it is associated with. Association between mapping and value predicate is done by name, just like for guard predicate. The sequence of parameters (paraDecls) a value predicate takes corresponds to the domain and range of the associated mapping.
  Example: The following excerpt from Listing 3 shows the value predicate associated to the class2table mapping. Parameters c and t are typed after the domain and range of the class2table mapping.

```
pred value_class2table(c:Class ,  t:Table){
  t.name[0]= c.name
}
```

– rule: Rules in F-Alloy are Alloy expressions that restrict the value of fields in the output model.
  – conditional rules enable each rule to be preceded by a condition with effect that the rule is applied if and only if the condition is satisfied.
    Example: a loose rule here is to be applied if and only if the attribute a given as parameter is primary.

```
a.is_primary= True implies c in CREATE.class2table[a.~
     attrs.*parent].pkeys
```

  – strict rules are direct restrictions of the value of a created or updated atom's field.
    Example: We restrict the name of a table $t$[9] to be the name of the class $c$.

```
  t.name[0]= c.name
```

  – loose rules are restrictions on the value of fields of an atom created or updated by another mapping.
    Example: The created column $c$ composes the set of pkeys of the table created, through the

[9] at index 0 as name is declared as a sequence of string

`class2table` mapping, from the parents of the class declaring $a$ (or the class itself if its has no parent).

```
c in CREATE.class2table[a.~attrs.*parent].pkeys
```

– `step` rules are used to inductively restrict a field of a created or updated atom to be composed of certain tuples.
Example: each entry in the name of the column $c$ corresponds to the name of the parent of the previous entry, with the first entry (index 0) being already defined through the use of a strict rule.

```
all i:Int| i>=1 and i< #(a.~attrs.^parent) implies
c.name[add[i,1]]= c.name[i].~name.parent.name
```

For completeness sake, we now give well-formedness constraints for the BNF given in Listing 6.

## 6.5 F-Alloy's Well-Formedness Constraints

1. Concerning F-Alloy's overall structure:

   (a) For specifying exogenous transformations, only a `CREATE` signature is included, as exogenous transformations define how to "create from scratch" an output instance given an input instance.
   (b) Mappings in the `CREATE` signature are associated to one guard and one value predicate each. Semantically, an atom typed by the range of the mapping is created for each tuples typed by the domain of the mapping and satisfying the guard predicate. The values of the fields of the created atom are then defined by the value predicate. Drawing a parallel with QVTr and TGG's terminology, guard and value predicates are thus the F-Alloy equivalents of QVT's checkonly and enforce patterns and of TGG's source and target graph, respectively.
   (c) In the `UPDATE` signature (present only in endogenous specifications), mappings are binary relations having the same signature as domain and range. They are also associated to one guard and one value predicate each. Semantically, each atom typed by the domain of the mapping and satisfying the guard is related to a new output atom of same type, with the intention that the former is to be replaced by the latter. The values of the fields of the updated atom are defined by the value predicate. Unassigned fields are left unchanged.
   (d) In the `DELETE` signature (present only in endogenous specifications), "mappings" are unary relations that represent atoms to be deleted. They are hence solely associated to a guard predicate with the intention that each atom typed by the domain of the mapping and satisfying the guard, as well as all references to that atom, are removed from the instance.
   (e) Correspondences between mappings and predicates are done by name, i.e., predicates are named after the mapping they are associated with, prefixed by the nature of the predicate ("`guard_`" or "`value_`").
   (f) In an `UPDATE` signature, occurrences of `qualName` composing each `mappingDecl` should be the same as an update operation is not meant to change the type of an atom.
   (g) For a `CREATE` mapping `map:` $\texttt{X}_1$`->...->`$\texttt{X}_n$`->Y` there are $n$ parameters in the guard predicate associated to the mapping with effect that a new atom `Y` is created for each tuple, typed by the domain of the mapping, satisfying the guard. Guard predicates of `UPDATE` and `DELETE` mappings take exactly one parameter.
   (h) For a `CREATE` mapping `map:` $\texttt{X}_1$`->...->`$\texttt{X}_n$`->Y` there are $n+1$ parameters in the value predicate associated to the mapping (this $n+1^{\text{th}}$ parameter representing the atom created of type `Y`), while value predicates of `UPDATE` mappings take exactly two parameters, namely, the old and new version of the updated atom. We note that there are no value predicates for `DELETE` mappings, atom satisfying the guard of a `DELETE` mapping being directly removed.
   (i) Two open statements (line 2) mean that the specification is exogenous, one statement means it is endogenous.
   (j) `expr` in guard predicates are boolean valued Alloy expressions that may solely contain features of $m_{\text{src}}$ and input parameters of the enclosing predicate, guards having as role to define patterns to be matched in the input instance.

2. Concerning F-Alloy's rules:

   (a) `expr` is an Alloy expression that may contain features of $m_{\text{src}}$ and input parameters of the enclosing predicate as well as occurrences of `image`. This restriction allows to enforce that the value of fields in the output instances solely depends on the input instance (as we will prove in Section 7.3).

(b) `strict` rules are used to express direct assignments, i.e., the assignment of a value to a field The first `name` occurrence is the name of the output parameter (i.e., typed after the range of the mapping), and the second is the name of a field declared in the signature typing that parameter. The `expr` in square brackets is to appear if and only if this field has an arity greater than 2.
(c) `loose` rules are used to express indirect assignments, i.e., state that a given value is composing a given field. the first `name` is the name of the output parameter and the second `name` is a field of the signature typing the `image`. The `expr` in square brackets is to appear if and only if this field has an arity greater than 2.
(d) `step` rules are used to express inductive assignments. They are thus always preceded by a `strict` rule, composing the base of the induction, and featuring an integer-valued index. In `step`, the first `name` occurrence is the name of the output parameter and the second is the name of the field assigned in the preceding `strict` rule. The right-hand side – `expr` – of `step` should then contain an occurrence of `name.name`**`[i]`**.
(e) In `range`, the two `expr` are integer-valued and the first `expr` value is bound to be equal to the `index` value of the `strict` rule composing the base of the induction for well-formedness sake.

## 6.6 F-Alloy to Alloy Correspondences

We finish this introduction to the F-Alloy syntax by showing, based on the given Alloy and F-Alloy BNFs, that any F-Alloy syntactic construct is also a valid Alloy syntactic construct (in other words an f-module is a syntactically valid Alloy module).

Correspondences between F-Alloy's syntactic constructs and their Alloy counterparts are detailed in Table 1.

## 7 Translational Semantics of F-Alloy

### 7.1 Overview

To ease the reading of this section, we picture the relation between Alloy and F-Alloy's semantics in Fig. 12.

The semantics of Alloy is given in [17] using a denotational approach – each syntactic construct being mapped to a function from instance to boolean value, with the effect of defining which properties are enforced in an instance by the construct in question. In short, an

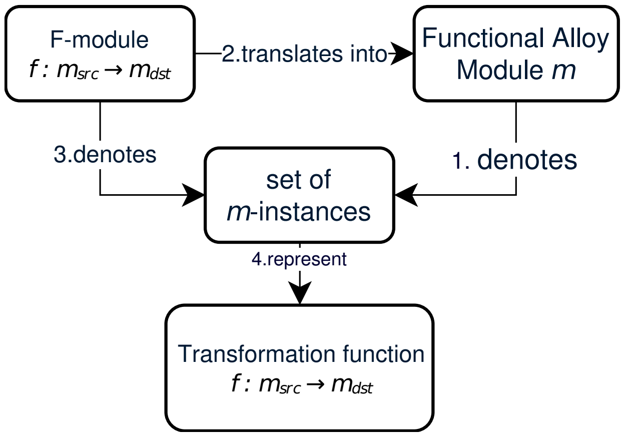


Fig. 12: Relation between Alloy and F-Alloy's semantics

Alloy module denotes a set of instances – edge number 1 of Fig. 12.

One of the great advantages in reusing the syntax of Alloy in the definition of the F-Alloy transformation language is the ease of translation between F-Alloy specifications and Alloy. Indeed, f-modules are by essence valid Alloy modules. Yet the meaning we give to f-modules differ from their original Alloy meaning. This difference stems from our design choice of leaving out redundant constraints needed to ensure the "functional behaviour" of f-module for conciseness sake.

In the next subsection we thus define a translation function $\mathcal{T} : \mathcal{F}alloy \to \mathcal{A}lloy$, with $\mathcal{F}alloy$ and $\mathcal{A}lloy$ being the set of all possible f-modules and Alloy modules, respectively, – corresponding to edge number 2 of Fig. 12 – and set the meaning of an f-module $m$ to be that of $\mathcal{T}(m)$ (as defined in [17]).

In a nutshell, $m$ denotes the same set of instances as $\mathcal{T}(m)$.

In Subsection 7.3, we will show that for any f-module $m$, $\mathcal{T}(m)$ is a functional Alloy module, hence showing that instances conforming to $\mathcal{T}(m)$, and thus transitively $m$, denote a transformation – corresponding to edge number 4 of Fig. 12.

### 7.2 Translating F-Alloy to Alloy

In this subsection, we define the translational semantics of F-Alloy by defining a translation function $\mathcal{T}$ mapping each f-module $m$, defining either an endogenous or an exogenous model transformation, to an Alloy Module $m_{\mathcal{T}}$. We define then the meaning of $m$ to be the one Alloy gives to $m_{\mathcal{T}}$.

When considered as an Alloy module (we recall that f-modules are syntactically valid Alloy modules), an f-module $m$ only defines relations from signatures of the

| F-Alloy construct | Alloy construct | explanation |
|---|---|---|
| `range` | `expr` | is of the form `expr compareOp expr binOp expr compareOp expr` |
| `step` | `expr` | is of the form `quant decl bar expr binOp expr compareOp expr` |
| `image` | `expr` | is of the form `expr binOp expr` |
| `loose` | `expr` | is of the form `expr compareOp expr binOp expr` |
| `strict` | `expr` | is of the form `expr compareOp expr` |
| `conditional` | `expr` | is of the form `expr binOp expr` |
| `rule` | `expr` | `strict`, `loose`, `conditional` and `step` are valid `expr` |
| `rBlock` | `block` | `rule` is a valid `expr` |
| `valueDecl` | `predDecl` | `rBlock` is a valid `block` |
| `guardDecl` | `predDecl` | straightforward |
| `mappingDecl` | `decl` | `expr -> expr` is a syntactically valid `expr` |
| `sigName` | `name` | straightforward |
| `fsigDecl` | `sigDecl` | mappingDecl is a valid `decl` |
| `fparagraph` | `paragraph` | straightforward |
| `fmodule` | `alloyModule` | hold from all previous statements |

Table 1: Syntactical correspondences between F-Alloy and Alloy syntactic constructs

source to signatures of the target module. Those relations provide enough structure to obtain transformation instances from analysis. Yet, the way elements are related is not constrained. Only some predicates are present to give information on how elements of the source and target should relate to each other. Constraints enforcing the properties defined in the predicates as well as other expected functional properties such as disjointness of the input and output instance should thus be made explicit as Alloy facts to convey the real intent of the F-Alloy specification.

In the definition of $\mathcal{T}$, we list those constraints that should be added to any f-module $m$ in order to obtain the Alloy module defining the meaning of $m$. We recall from Def. 5 that for a mapping `map:` $X_1$`->..->`$X_n$`->Y`, the domain denotes the sequence of signatures $X_1,..,X_n$ and the range denotes the signature `Y`. We introduce the term of *input tuple* and *output atom* here to denote the set of tuples typed after the domain of a mapping and satisfying its associated guard and those atoms typed after the range of a mapping and who have been mapped to input tuples through that mapping, respectively. We now list the constraints to be added to an f-module $m$ in order to obtain $m_{\mathcal{T}}$. Pseudo codes showing how each of those constraints can be procedurally generated are given in Annexe A.

- Constraints to be added to any f-module $m$
  - **Map Disjunction.** Mappings declared in `CREATE` and `UPDATE` signatures define partial functions which have disjoint ranges.
    The intent is to ensure that for any rule, the triggering of a guard will lead to the creation of a new atom. This constraint enables, when interpreting f-modules (see Section 8), to consider mappings one at a time when creating output elements.
    Example (`CD2RDBMS`): columns representing attribute and association should be disjoint.

```
CREATE.attribute2column[Attribute] & CREATE.
     association2column[Association,Attribute] =none
```

  - **Map Injectiveness.** Functions defined by `CREATE` and `UPDATE` mappings are injective.
    The intent is again to simplify the process of creating output elements when interpreting f-modules. This time, this constraint enables us to consider candidates to trigger a mapping one at a time, a new output element being created at any triggering.
    Example (`CD2RDBMS`): distinct attributes should be mapped to distinct columns, at most one attribute being mapped to a given column.

```
all y: Column |  lone CREATE.attribute2column.y
```

  - **Predicate Association.** For each `CREATE` and `UPDATE` mapping there is an output atom exactly for those tuples in the domain that satisfy the guard predicate. The values of fields of the output atom are defined in the value predicate. For a `DELETE` mapping there is a tuple of the form $(d, s)$ where $d$ is a single `DELETE` atom for each atom $s$ in the domain of the mapping satisfying the guard predicate.
    The intent is to ensure that guard and value predicates play their expected role of pre- and post-conditions for the mappings they are associated to.
    Example (`CD2RDBMS`): 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_attribute2column[x] and one CREATE.
     attribute2column[x]
and value_attribute2column[x, CREATE.attribute2column[x
     ]])
or
(not guard_attribute2column[x] and no CREATE.
     attribute2column[x])
}
```

– **Minimal Assignment.** The values of the fields of output atoms are limited to those explicitly specified through rules, except for UPDATE mappings, in which case values of fields which are not specified through rules are equal to the values of the same fields prior to update.
The intent of this constraint is to provide an upper-bound to fields for which values where partially specified, *e.g.*, using "in" instead of strict equality. The extra clause of this constraint concerning UPDATE mappings is here to make endogenous transformations less verbose. The idea is to avoid rewriting existing values by limiting the transformation to the expression of what has to change.
Example (CD2RDBMS): The name of a column in the range of the association2column mapping is a sequence of string whose size is equal to the number of elements returned by all the expressions explicitly assigned through rules.

```
#c.name=add[#att.name,#ass.name]
```

Example (CDRefinement): isAbstract is the only field present in the noUselessAbstract value predicate. Constraints are thus added to force other fields to keep their original values.

```
pred value_noUselessAbstraction(c:Class,y:Class){
  y.isAbstract=False
  y.attrs=c.attrs //added constraint
  y.parent=c.parent //added constraint
  y.name=c.name //added constraint
}
```

– Constraints to be added to any f-module $m$ defining an endogenous model transformation:
  – **IO Disjunction.** The set of all atoms in the input tuples of CREATE, UPDATE, and DELETE mappings are disjoint with the set of all output atoms of CREATE and UPDATE mappings.
  The intent is to clearly separate through constraints the input instance from the output instance. We note that these constraints are only needed for endogenous transformations as the type of atoms in an exogenous transformation are signatures being declared either in the input or in the output module of the transformation, hence enabling to clearly dissociate atoms belonging to the input instance from those in the output instance.
  Example (CDRefinement): the set of classes which are output atoms in the range of the mapping associationClass2Class and the set of classes in input tuples in the domain of the mapping newAssociation should be disjoint.

```
no associationClass2Class[AssociationClass] &
     newAssociations.Association.Association
```

  – **Constraints Framing** Transformation instances conforming to an endogenous f-module are composed of an input-instance and of tuples, typed by CREATE, UPDATE and DELETE mappings, embodying the operations to be performed (see Fig. 14). This becomes problematic when considering the constraints declared in the source model of the transformation. Indeed, when running an analysis on such an endogenous specification, those aforementioned constraints should be satisfied in the transformation instance. Yet this later contains source model elements other than those present in the valid source model instance given as input. This particularity of endogenous transformation makes it likely that analysis fails to obtain all expected transformation instances. A solution to this problem is to give a context to those source model constraints, so that they apply only on the input instance and output instance, respectively (instead of the transformation instance). To do so, all facts of the input module are to be rewritten as predicates, taking as parameter the set of atoms they are to be applied on, *i.e.*. Those predicates are then called in a fact in the transformation module, where the parameter given correspond to the set of atoms composing the input instance (returned by $f_{\text{in}}$) and the output instance (returned by $f_{\text{out}}$), respectively.
  Example (CDRefinement): the attribute disjointness constraint is bound to be violated when updating a class without updating its set of attributes (as a result two classes with the same set of attributes will appear in the transformation instance). To prevent this violation and still ensure that attributes of classes are disjoint in the input-instance and output-instance, the disjointness constraint in the input module is replaced by the following predicate:

```
  pred attrDisj(context:set univ){
    no disj x,y: ((Class+AssociationClass) & context) |
          x.(attrs& context->context) & y.(attrs&
         context->context) !=none
  }
```

In this predicate, all the sets and tuples of atoms present in the formula are restricted to the context given as parameter.
This predicate is then called in the transformation module with parameters consisting of all the atoms present in the input and output instance, respectively. Those set of atoms can be defined in Alloy, following the definition of $f_{\text{in}}$ and $f_{\text{out}}$ given in Definitions 7 and 8. Following is how those sets of atoms are defined for our `CDRefinement` case study:

```
let input = univ - (CREATE + DELETE + UPDATE + UPDATE.
      fixAbstract[Class] + CREATE.
      associationClass2Class[AssociationClass] + CREATE
      .newAssociations[Class, Association])

let output = univ - (CREATE + DELETE + UPDATE + DELETE
      .(associationWithClass + associationClass) +
      UPDATE.fixAbstract.Class)
```

– Constraints to be added to any f-module $m$ defining an exogenous model transformation:
  – **Minimum Output.** For an f-module $m$ from $m_{\text{src}}$ to $m_{\text{dst}}$, we enforce that the set of atoms typed by signatures declared in $m_{\text{dst}}$ should be limited to the output atoms of `CREATE` mappings declared in $m$. The intent is for the output of the transformation to be composed of only those elements defined through mappings (no extra elements are allowed to be generated by analysis). We note that this constraint does not apply to endogenous transformations because any $m_{\text{src}}$ element not present in the range of a mapping is considered as part of the transformation's input-instance.
  Example (`CD2RDBMS`): RDBMS elements are limited to the output atoms of declared mappings.

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

For any given f-module $m$, $m_{\mathcal{T}} = \mathcal{T}(m)$ is the Alloy module $m$ to which the aforementioned constraints are added. We will see in the next subsection that those added constraints enforce the "functional" behavior expected from the Alloy module represented by an F-Alloy specification, that is, that any Alloy module $\mathcal{T}(m)$ is a functional Alloy Module.

## 7.3 From F-modules to Functional Alloy Modules

We are now interested in proving that any Alloy module $m_{\mathcal{T}}$ obtained from the translation of an f-module $m$ is a functional Alloy module. This enables us to ensure that F-Alloy specifications indeed denote transformation functions.

To achieve our goal, we first define the transformation functions $f_{\text{in}}$ and $f_{\text{out}}$ used to extract from an $m_{\mathcal{T}}$-instance the input and output of the transformation denoted by $m$. Next we examine the influence of rules defined in value predicates on $m_{\mathcal{T}}$-instance. We then show by construction that for any $m_{\mathcal{T}}$, there cannot be two distinct $m_{\mathcal{T}}$-instance $x_{\mathcal{T}}$ and $x'_{\mathcal{T}}$ s.t. $f_{\text{in}}(x_{\mathcal{T}}) \neq f_{\text{in}}(x'_{\mathcal{T}})$ which will allow us to finally conclude that any Alloy module $m_{\mathcal{T}}$ obtained from the translation of an f-module $m$ is a functional Alloy module (with respect to the defined transformation functions $f_{\text{in}}$ and $f_{\text{out}}$).

### *7.3.1 Exogenous transformation functions*

We are about to prove that the module $m_{\mathcal{T}}$ obtained by translation is a functional Alloy module. Before we can actually do this, we need to define the functions $f_{\text{in}}$ and $f_{\text{out}}$ that extract the input and output instances from a valid instance of $m_{\mathcal{T}}$. In the present subsection we define these functions for the exogenous case. In the next subsection we then deal with the endogenous case.

In the exogenous case we call instance projection the operation those functions embody and define it as follows:

**Definition 6 (Instance Projection)** The projection of an instance $x : (X, Y, m)$ on a module $m' : (S', F', \varphi')$, with $m'$ being in the import hierarchy of $m$, 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 $X' = \{a^s \in X | s \in S'\}$ and $Y' = \{y^f \in Y | f \in F'\}$ .

Thus for exogenous transformations:

$f_{\text{in}}(x) = x \Downarrow m_{\text{src}}$ and $f_{\text{out}}(x) = x \Downarrow m_{\text{dst}}$

We illustrate this application of instance projection in Fig. 13.

This figure represents a `CD2RDBMS` instance (nodes represent atoms and arrows represent tuples.). The projections of this instance on $m_{\text{src}}$ and $m_{\text{dst}}$ are represented in red and bold, respectively. We notice that String typed atoms are part of both projections as the built-in type String is well defined in both $m_{\text{src}}$ and $m_{\text{dst}}$.

### *7.3.2 Endogenous transformation functions*

We now define the functions $f_{\text{in}}$ and $f_{\text{out}}$ that extract the input and output instances from a valid instance of $m_{\mathcal{T}}$ in the endogenous case. We first give informal definitions and then provide formal definitions.

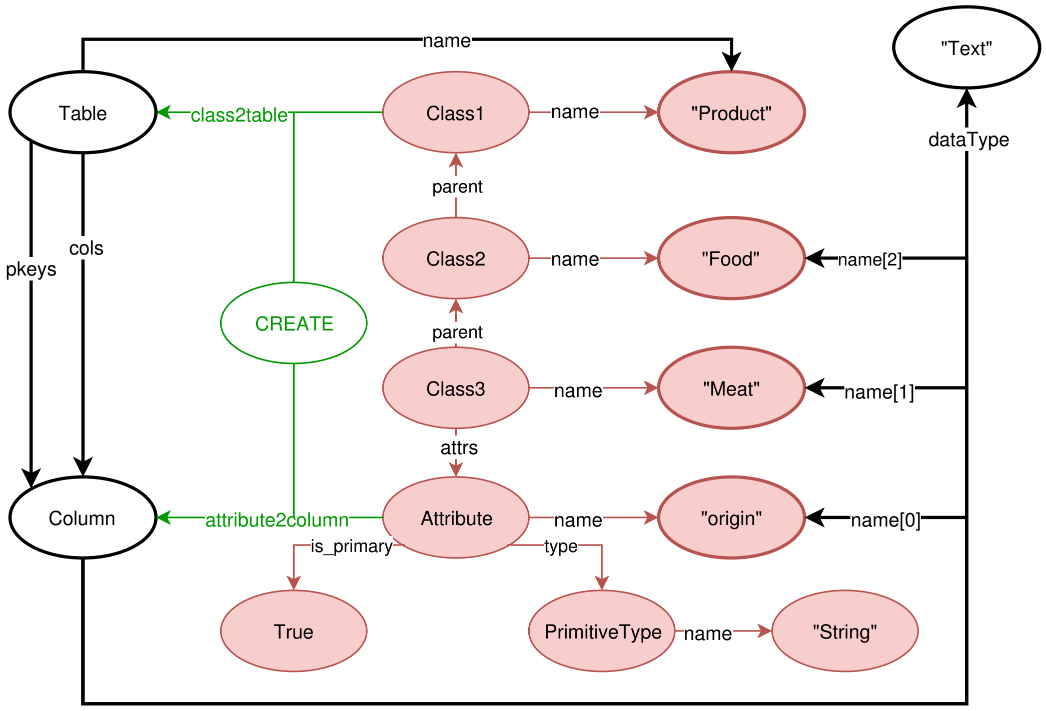


Fig. 13: A CDRDBMS-instance $x$ with $f_{\text{in}}(x)$ highlighted in red and $f_{\text{out}}$ highlighted in bold

The function $f_{\text{in}}$ extracts the input instance of the transformation from a given $m_{\mathcal{T}}$-instance, namely, the sub-instance made up of atoms typed by signatures in $m_{\text{src}}$ and tuples typed by fields in $m_{\text{src}}$, with all atoms in the range of a CREATE or UPDATE mappings (and their associated tuples) being removed.

The function $f_{\text{out}}$ extracts the output instance of the transformation from a given $m_{\mathcal{T}}$-instance, namely, the sub-instance made up of atoms typed by signatures in $m_{\text{src}}$ and tuples typed by fields in $m_{\text{src}}$ in which elements in the domain of UPDATE mappings are replaced by their images, preserving links to the replaced element, and where elements in the domain of DELETE mappings are removed.

We call those operations embodied by $f_{\text{in}}$ and $f_{\text{out}}$ "transformation-aware input projections" and "transformation-aware output projections", respectively, and define them as follows. In the following two definitions, we denote by $f_{\text{d}}$ and $f_{\text{r}}$ the set of input tuples and output atoms composing tuples typed by mapping $f$, respectively. We also write $a^s \in t^f$ if the atom $a$ typed by signature $s$ is contained in the tuple $t$ typed by field $f$.

**Definition 7 (transformation-aware input projection)** Given an instance $x : (X, Y, m)$, with $m : (S, F, \varphi)$ being a functional Alloy module expressing an endogenous transformation on $m_{\text{src}} : (S', F', \varphi')$, the *m-aware input projection* of $x$, denoted $x \Downarrow_m^{in}$ is the projection of $x$ on $m_{\text{src}}$ from which we subtract output atoms of mappings in $F$.

Formally : $x \Downarrow_m^{in} = x \Downarrow m_{\text{src}} - (X', Y', m_{\text{src}})$ with :

– $X' = \{a^s \in X : \exists f \in F \text{ s.t. } a^s \in f_{\text{r}}\}$
– $Y' = \{t^f \in Y : \exists a^s \in X' \text{ s.t. } a^s \in t^f)\}$

In the following definition, we denote $C$, $U$ and $D$ the set of CREATE, UPDATE and DELETE mappings declared in $m$, respectively.

**Definition 8 (transformation-aware output projection)** Given an instance $x : (X, Y, m)$ – with $m : (S, F, \varphi)$ being a functional Alloy module expressing an endogenous transformation on $m_{\text{src}} : (S', F', \varphi')$ – the $m$-aware output projection of $x$, denoted $x \Downarrow_m^{out}$ is the projection of $x$ on $m_{\text{src}}$ from which we subtract input tuples of UPDATE and DELETE mappings in $F$.

Formally : $x \Downarrow_m^{out} = x \Downarrow m_{\text{src}} - (X', Y', m_{\text{src}}) + (\emptyset, Y'', m_{\text{src}})$ with :

– $X' = \{a^s \in X : \exists t^f \in Y \text{ s.t. } f \in U \cup D \wedge a^s \in f_{\text{d}})\}$
– $Y' = \{t^f \in Y : \exists a^s \in X' \text{ s.t. } a^s \in t^f)\}$
– $Y'' = \bigcup_{i \in [1,n], t^f : (x_1,..,x_n) \in Y : f \in U} \{(x_1, .., f(x_i), .., x_n) | \exists a^s \in f_d : a^s = x_i\}$ ,
where $f(x_i)$ is the atom mapped to $x_i$ through the update mapping $f$.

We then define for endogenous transformations:

$f_{\text{in}}(x) = x \Downarrow_m^{in}$ and $f_{\text{out}}(x) = x \Downarrow_m^{out}$

We illustrate the application of transformation-aware projections in Fig. 14. In those visualizations, atoms are represented by nodes and tuples by links. Note that the tuples typed by the `CDElement`'s field `name` have been filtered out for readability's sake.

Figure 14(a) represents a `CDRefinement` instance in which `CREATE`, `UPDATE` and `DELETE` atoms and tuples are depicted in green, purple and red, respectively. All atoms being in the domain of mappings as well as tuples being exclusively composed of those aforementioned atoms are highlighted in boldface. Those atoms and tuples compose the transformation aware input projection of the depicted `CDRefinement` instance. The transformation aware output projection of the studied instance is given in Fig. 14(b). In this instance, atoms in the range of `CREATE` and `UPDATE` mappings, as well as tuples conforming to the value predicate of those mappings, are highlighted in green and purple, respectively.

### *7.3.3 The meaning of rules*

Considering an f-module $m$, its associated Alloy module $m_{\mathcal{T}} = \mathcal{T}(m)$ and $x_{\mathcal{T}} \in I(m_{\mathcal{T}})$, the predicate association constraints defined previously enforce that, given a mapping $\mu$ declared in $m$, only those input tuples for which the guard predicate associated to $\mu$ is satisfied are associated through $\mu$ to an output atom. Moreover, the value predicate associated to $\mu$ holds in $x_{\mathcal{T}}$ when given as parameter any such pair of input tuples and output atom.

Rules inside a value predicate are boolean expressions that may use the parameters of the value predicate. In the following lemma we prove that each rule can be rewritten in the form

$$V \texttt{ in f}$$

where $V$ is an Alloy expression denoting a relation, `in` denotes set inclusion, and `f` is a field of $m_{\text{dst}}$. Since $V$, seen as a set of tuples, generally depends on the rule $r$, the input instance $f_{\text{in}}(x_{\mathcal{T}})$, and the parameters $x_1, \ldots, x_n$ and $y$ of the enclosing value predicate, we refer to $V$ as $V(r, f_{\text{in}}(x_{\mathcal{T}}), x_1, \ldots, x_n, y)$.

To understand the importance of this lemma, note that the value predicate is a conjunction of rules, each stating that a set of tuples is contained in the relation representing a field of $m_{\mathcal{T}}$. This will imply that the relation of each field can be written as the union of the $V$-sets of the contributing rules, since no other tuples can be in the relation of the field due to the minimal assignment constraints. This fact will be used to derive an explicit formula for the "shape" of a valid instance of $m_{\mathcal{T}}$ in Lemma 2.

**Lemma 1** *Each rule $r$ can be rewritten as a logically equivalent formula*

$$V(r, f_{in}(x_{\mathcal{T}}), x_1, \ldots, x_n, y) \texttt{ in f}$$

*where $V$ is an Alloy expression denoting a relation,* `in` *denotes set inclusion, and* `f` *is a field of $m_{dst}$.*

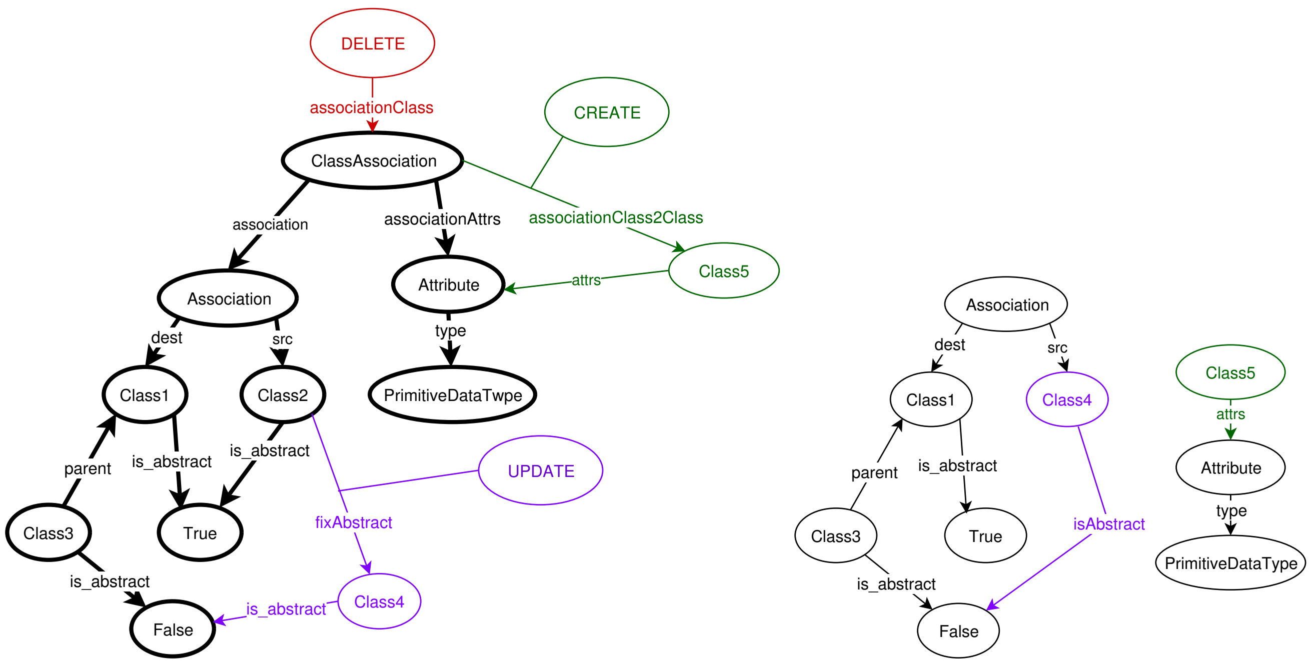
*Proof.* We describe how to rewrite each type of rule in the required form. In the following, occurrences of `name` in each syntactic construct are replaced by `y` when they denote the output atom (according to the well-formedness of F-Alloy), or by `f` if they denote the field the rule is contributing to. The rewriting of rules is done as follows:

– For `strict` rules:
  1. `y.f[[expr₁]] = expr₂`
  2. `[expr₁-> ]expr₂ in y.f`
  3. `y-> [expr₁-> ]expr₂ in f`
– For `loose` rules:
  1. `y in image.f[[expr]]`
  2. `[expr-> ]y in image.f`
  3. `image-> [expr-> ]y in f`
– For `step` rules:
  1. `all i:Int|range implies y.f[add[i,1]] = expr`
  2. `all i:Int|range implies [add[i,1]]-> expr in y.f`
  3. `all i:Int|range implies y-> [add[i,1]]-> expr in f`

Note that in all cases:

– item 1. is the base case directly taken from F-Alloy's BNF
– item 2. is obtained by refining item 1., of the form `y.f = v`, to an assignment of the form `v in y.f`. Both forms are semantically equivalent if we consider Minimal Assignment constraints.
– item 3. is obtained by rewriting item 2. by passing relation joins from one side of the equation to the other.

We list in Table 2 the set of tuples returned by the function $V$. For strict rules `expr₁` and `expr₂` may use input parameters as well as features of $m_{\text{src}}$ (by the well-formedness constraints given in Subsection 6.5). Thus the $V$-set for strict rules only depends on the input instance and the parameter values. For loose rules, image denotes the output atom for an input tuple given by the specified expressions, which depends only on the input instance and values of input parameters. Therefore, the tuples in the $V$-set for loose rules depend only on the input instance and the parameter values as well. The same holds for step rules since the `expr` satisfies similar restrictions and the range is given by an expression that depends on the input instance and values of



(a) A `CDRefinement`-instance $x$ with $f_{\text{in}}(x)$ highlighted in bold

(b) $f_{\text{out}}(x)$ (with $x$ depicted in (a))

Fig. 14: A `CDRefinement`-instance, and the input and output of the transformation `CDRefinement` specifies as defined by $f_{\text{in}}$ and $f_{\text{out}}$, respectively.

input parameters. Since the conditional expression in conditional rules satisfy the same restrictions, the corresponding $V$-set only depends on the input instance and the values of the parameters. □

We now exemplify the valuation of `t` given in Table 2 by explicitly listing the tuples assigned through rules of the value predicate associated to the mapping `attribute2column` (taken from the `CD2RDBMS` case study). To do so, let us consider the `CD2RDBMS`-instance $x_{\mathcal{T}}$ depicted in Fig. 13. Given the two parameters `a= Attribute` and `c=Column` and $x_{\text{src}} = f_{\text{in}}(x_{\mathcal{T}})$ we have:

– With $r$ being the strict rule

```
c.dataType=(a.type.name="String" implies "TEXT" else "
     NUMBER")
```

$$V(r, x_{\text{src}}, a, c) = \{c, a\texttt{.type.name="String"}\ \textbf{\texttt{implies}}\ \texttt{"TEXT"}\ \textbf{\texttt{else}}\ \texttt{"NUMBER"}\}^{\texttt{dataType}}$$
$$= \{\texttt{Column}, \texttt{"TEXT"}\}^{\texttt{dataType}}$$

– With $r$ being the strict rule

```
c.name[0]= a.name
```

$$V(r, x_{\text{src}}, a, c) = \{c, 0, a\texttt{.name}\}^{\texttt{name}}$$
$$= \{\texttt{Column}, 0, \texttt{"origin"}\}^{\texttt{name}}$$

– With $r$ being the strict rule

```
c.name[1]=((a.~attrs.parent)!=none implies a.~attrs.name
     else none)
```

$$V(r, x_{\text{src}}, a, c) = \{c, 1, \texttt{((}a\texttt{.~attrs.parent)!=none}\ \textbf{\texttt{implies}}\ a\texttt{.~attrs.name}\ \textbf{\texttt{else}}\ \texttt{none)}\}^{\texttt{name}}$$
$$= \{c, 1, \texttt{(Class2!=none}\ \textbf{\texttt{implies}}\ \texttt{Class3.name}\ \textbf{\texttt{else}}\ \texttt{none)}\}^{\texttt{name}}$$
$$= \{\texttt{Column}, 1, \texttt{"Meat"}\}^{\texttt{name}}$$

– With $r$ being the step rule

```
all i:Int| i>=1 and i< #(a.~attrs.^parent) implies c.name[
     add[i,1]]= c.name[i].~name.parent.name
```

$$V(r, x_{\text{src}}, a, c) = \bigcup_{i \in \texttt{[1,\#(a.~attrs.^parent)[}} \{(c, i+1, c\texttt{.name[i].~name.parent.name})^{\texttt{name}}\}$$
$$= \bigcup_{i \in \texttt{[1,2[}} \{(c, i+1, c\texttt{.name[i].~name.parent.name})^{\texttt{name}}\}$$
$$= \{(c, 2, \texttt{"Meat".~name.parent.name})^{\texttt{name}}\}$$
$$= \{(c, 2, \texttt{Class3.parent.name})^{\texttt{name}}\}$$
$$= \{(\texttt{Column}, 2, \texttt{"Food"})^{\texttt{name}}\}$$

– With $r$ being the conditional rule

```
a.is_primary=True implies c in CREATE.class2table[a.~
     attrs.*parent].pkeys
```

| Type of Rule $r$ | Syntax followed by $r$ | value of $V(r, x_{\mathcal{T}}, (x_1, ..., x_n), y)$ |
|---|---|---|
| `strict` | `y.f[[expr1]] = expr2` | $\{(y, \texttt{expr}_1, \texttt{expr}_2)^{\texttt{f}}\}$ |
| `loose` | `y in image.f[[expr]]` | $\{(\texttt{image}, \texttt{expr}, y)^{\texttt{f}}\}$ |
| `step` | **`all i:Int`**`|range` **`implies`** `y.f[add[i,1]] = expr` | $\bigcup_{\{i \in \texttt{range}\}} \{(y, i+1, \texttt{expr})^{\texttt{f}}\}$ |
| `conditional` | `expr` **`implies`** `rule` | $\begin{cases} V(\texttt{rule}, x_{\mathcal{T}}, (x_1, ..., x_n), y), & \text{if } x_{\mathcal{T}} \vDash \texttt{expr} \\ \emptyset & \text{otherwise} \end{cases}$ |

Table 2: Valuation of $V$ for the different rule constructs identified in Listing 6

$$V(r, x_{\text{src}}, a, c) = \begin{cases} V(r_2, x_{\text{src}}, a, c) \\ \qquad \text{if } x_{\text{src}} \vDash a.\texttt{is\_primary=True} \\ \emptyset \\ \qquad \text{otherwise} \end{cases}$$
$$= V(r_2, x_{\text{src}}, a, c)$$

with $r_2$ being the loose rule:

```
c in CREATE.class2table[a.~attrs.*parent].pkeys
```

$$V(r, x_{\text{src}}, a, c) = \{(\texttt{class2table[}a\texttt{.~attrs.*parent]}, c)^{\texttt{pkeys}}\}$$
$$= \{(\texttt{Table, Column})^{\texttt{pkeys}}\}$$

– With $r$ being the loose rule:

```
c in CREATE.class2table[a.~attrs.*parent].cols
```

$$V(r, x_{\text{src}}, a, c) = \{(\texttt{class2table[}a\texttt{.~attrs.*parent]}, c)^{\texttt{cols}}\}$$
$$= \{(\texttt{Table, Column})^{\texttt{cols}}\}$$

### *7.3.4 $m_{\mathcal{T}}$ is a functional Alloy module*

Let us consider an f-module $m$ from $m_{\text{src}}$ to $m_{\text{dst}}$ specifying an exogenous or endogenous transformation. The following lemma proves that valid instances of $m_{\mathcal{T}}$ have a particular "shape", namely, they can be written as the union of the input instance, atoms and tuples representing the mappings, as well as the union of $V$-sets of contributing rules. This lemma constitutes the final stepping stone that will allow us to prove (in Theorem 1) that $m_{\mathcal{T}}$ is a functional Alloy module.

In the following, we introduce the notation $\overrightarrow{x}$ as a shorthand for tuples of the form $(x_1, ..., x_n)$

**Lemma 2** *Any valid instance $x_{\mathcal{T}} : (X, Y, m_{\mathcal{T}})$ satisfies the equation:*

$$x_{\mathcal{T}} = f_{in}(x_{\mathcal{T}}) \cup A \cup \bigcup_{b \in A, \mu \in M_b} F(\mu) \tag{1}$$

*where*

$$F(\mu) = \bigcup_{(b, \overrightarrow{x}, y)^{\mu} \in Y} \Bigg( \{y\} \cup \{(b, \overrightarrow{x}, y)\} \cup \bigcup_{r \in \mu} V(r, f_{in}(x_{\mathcal{T}}), \overrightarrow{x}, y) \Bigg) \tag{2}$$

*with*

- *the union between an instance $(X, Y, m)$ and a set of atom $A$ being $(X \cup A, Y, m)$*
- *the union between an instance $(X, Y, m)$ and a set of tuples $T$ being $(X, Y \cup T, m)$*
- *the union between the two instances $(X, Y, m)$ and $(X', Y', m)$ being $(X \cup X', Y \cup Y', m)$*
- *$A$ being the set of singleton atoms typed after the signatures declared in $m$ (`CREATE`, and possibly `UPDATE` and `DELETE`)*
- *$M_b$ being the set of mappings declared in the signature typing $b$.*
- *$(b, \overrightarrow{x}, y)^{\mu} \in Y$ denoting that tuple $(b, \overrightarrow{x}, y)$ ranges over all tuples in $Y$ typed by the mapping $\mu$ with first component equal to $b$.*
- *$r \in \mu$ denoting that rule $r$ is declared in the value predicate associated to $\mu$.*
- *$V(r, f_{in}(x_{\mathcal{T}}), \overrightarrow{x}, y)$ as defined in the previous subsection.*

*Note that tuples typed by delete mapping are of the form $(b, \overrightarrow{x})$. Moreover no value predicate is associated to the mapping. This is why we can simply replace $V_r$ by $\emptyset$ in the given equation. Hence we would have for any $\mu$ declared in a value `DELETE` signature:*

$$F(\mu) = \bigcup_{(b, \overrightarrow{x})^{\mu} \in x_{\mathcal{T}}} \{(b, \overrightarrow{x})\}$$

*Proof:*

- $f_{\text{in}}(x_{\mathcal{T}})$ yields a sub-instance of $x_{\mathcal{T}}$ by construction, in both endogenous and exogenous cases.
- The presence of elements of $A$ in $x_{\mathcal{T}}$ is due to a syntactic constraint of F-Alloy. Indeed each signature declaration is preceded by the keyword `one` ensuring the presence of exactly one atom typed by it.
- The rest of the formula is enforced by predicate association constraints stating that for each signature in $m$, for each mapping, for each input tuple, an output atom $y$ should be part of the mapping (hence the addition of tuple $(b, \overrightarrow{x}, y)$). This output atom $y$ should have its field bounded by the value predicate – i.e., the set of tuples returned by the $V$-set of each rule $r$ declared in the associated value predicate. Note that Map Injectiveness, Map Disjunction

and IODisjunction constraints enforce every $y$ to be disjoint from $f_{\text{in}}(x_{\mathcal{T}})$ and from each other.
– Minimum Output constraints prevent $x_{\mathcal{T}}$ from being composed of any other elements in the case of an exogenous transformation. In the case of an endogenous transformation, any other element which is not part of a mapping is in $f_{\text{in}}(x_{\mathcal{T}})$.

We have shown by construction that the equation given in Lemma 2 is bound to hold in any instance of an Alloy module $m_{\mathcal{T}}$ obtained by translation of an F-module. □

**Theorem 1 (F-modules translate to functional Alloy modules)** *For any f-module $m$, the translated module $m_{\mathcal{T}}$ is a functional Alloy module with respect to $f_{in}$ and $f_{out}$ defined in Section 7.3.1 for exogenous transformations and in Section 7.3.2 for endogenous transformations.*

*Proof.* Let $x_{\mathcal{T}}$ and $x'_{\mathcal{T}}$ be two instances of $m_{\mathcal{T}}$ with $f_{\text{in}}(x_{\mathcal{T}}) = f_{\text{in}}(x'_{\mathcal{T}})$.

Let us take a look at the terms on the right-hand side of the top equation in Lemma 2.

The first term is the same for $x_{\mathcal{T}}$ and $x'_{\mathcal{T}}$ by the above assumption.

The second term $A$ is the same for $x_{\mathcal{T}}$ and $x'_{\mathcal{T}}$.

For the third term select a mapping $\mu$. Note that the set of argument tuples $\overrightarrow{x}$ that satisfy the guard of $\mu$ is the same in both cases because $f_{\text{in}}(x_{\mathcal{T}}) = f_{\text{in}}(x'_{\mathcal{T}})$. Now fix one such argument tuple $\overrightarrow{x}$. The tuple $(b, \overrightarrow{x}, y)$ is the same in $x_{\mathcal{T}}$ and $x'_{\mathcal{T}}$. Finally the set of tuples $V(r, f_{\text{in}}(x_{\mathcal{T}}), \overrightarrow{x}, y)$ is the same, for any rule $r$ in $x_{\mathcal{T}}$ and $x'_{\mathcal{T}}$.

We conclude that $x_{\mathcal{T}}$ and $x'_{\mathcal{T}}$ are identical. It follows that $f_{\text{out}}(x_{\mathcal{T}}) = f_{\text{out}}(x'_{\mathcal{T}})$ and hence $m_{\mathcal{T}}$ is a functional Alloy module. □

## 8 Interpreting F-Alloy specifications

We have seen in the previous section that any instance $x_{\mathcal{T}}$ conforming to an Alloy module $m_{\mathcal{T}} = \mathcal{T}(m)$ (with $m$ being an f-module) can be computed from its subinstance $f_{\text{in}}(x_{\mathcal{T}})$ (see Lemma 2). In this section, we present the procedure for performing this computation, a process we call *interpretation.*

First, we give the F-Alloy interpretation procedure in Listing 7. The interpretation of rules is given as a subprocedure in Listing 8.

The notation in which the pseudo-code is provided has the following properties:

– Keywords structuring the code are highlighted in bold and are written in uppercase.
– Variables names are highlighted in bold and are written in lowercase.
– Text and numbers are highlighted in italic.
– Function names (in declarations and in calls) are highlighted in bold and italic.
– Expressions of the form "`WITH var: description`" consists in providing a syntactic description of the variable, consisting of syntactical constructs and terminals. Each syntactical construct counts as a variable declaration, the variable being initialized following the value of the described variable `var`. Each of such "syntactic variable" is highlighted in blue, to differentiate them from terminals and other possibly already defined variables.

It may be helpful for the reader to consult the formal definitions of instance and module in Section 4 before reading the procedures below.

We note that in the following pseudo-code, the function `Eval(AlloyExpression e, Instance x)` represents the method `A4Solution.eval` provided in the Alloy API[10]. This method evaluates an expression `e` in an A4Solution (object representation of an instance) `x` and returns a set of atoms, tuples or a boolean value depending on the type of expression given as parameter.

The pseudo code, given in Listing 7, closely follows the structure of transformation instances detailed in the equation (1) of Lemma 2. We see that interpretation is performed in three steps:

1. **Initialization** [ lines 4 to 16]: The instance to be returned ($x_{\mathcal{T}}$) is set to contain atoms and tuples of the input instance ($x_{\text{src}}$) and arbitrary CREATE, UPDATE and DELETE atoms (two last ones only if the transformation is endogenous).
2. **Mapping Processing** [lines 17 to 45]: A first iteration over declared mappings in which guard predicates are evaluated for each combination of atoms in the input instance ($x_{\text{src}}$). In the case of `CREATE` and `UPDATE` mappings, a new atom $y$, typed by the range of the mapping is created each time a combination of atoms satisfying the guard is found. Both the created atom and a trace tuple (marking the relation between the triggering combination of atoms and the created atoms) are added to the instance to be returned ($x_{\mathcal{T}}$). In the case of `DELETE` mappings, no atoms are created, and thus only the trace tuple (marking the triggering combination of atoms to be deleted) is added to the instance to be returned.
3. **Rule Processing** [lines 47 to 54]: A second iteration assigning values to fields of created atoms by parsing rules of associated value predicates. Four

[10] `http://alloy.mit.edu/alloy/documentation/alloy-api/index.html`

```
1  /*INPUT : an F-module m from m_src to m_dst and an m_src-
         instance x_src
2   *OUTPUT: a valid mT-instance xT:(A, T, mT) s.t. f_in(xT) =
          x_src or NONE */
3  FUNCTION Interpretation(F-module  m, Instance  x_src)
4    WITH x_src: (A_src, T_src, m_src)
5    LET c = new Atom(CREATE), u = none , d = none
6    LET m_src = m.getImportedModule(1) //1st open statement
7    WITH m_src= (S_src, F_src, phi_src)
8    LET m_dst = m.getImportedModule(2) //2nd open statement
9    WITH m_dst: (S_dst,F_dst,phi_dst)
10   IF m_dst =  none THEN //endogenous case
11     u = new Atom(UPDATE)
12     d = new Atom(DELETE)
13   FI
14   // We will return xT:(A,T,mT)
15   LET A = c + u + d + A_src
16   LET T = T_src
17   FOR EACH mapping map DECLARED IN m DO
18     LET sig = signature in which map is declared
19     IF sig = CREATE THEN
20       WITH map: name: X1 -> .. -> Xn -> Y
21       FOR EACH tuple (x1,..,xn) of atoms IN A_src
22         WITH x1,..,xn typed by X1, .., Xn
23         IF Eval(guard_name(ax, .. ,xn), x_src) THEN
24           y = new Atom(Y)
25           A = A + y
26           T = T + (c, x1, .., xn, y)
27         FI
28       DONE
29     ELSE IF sig = UPDATE THEN
30       WITH map: name: X -> Y
31       FOR EACH atom x IN A_src typed by X DO
32         IF Eval(guard_name(x), x_src) THEN
33           y = new Atom(Y)
34           A = A + y
35           T = T + (u,x,y)
36         FI
37       DONE
38     ELSE IF sig = DELETE THEN
39       WITH map: name: X
40       FOR EACH atom x IN  A_src typed by X DO
41         IF Eval(guard_name(x), x_src) THEN
42           T = T + (d,x)
43         FI
44       DONE
45     FI
46   DONE
47   FOR EACH mapping map DECLARED IN m DO  // CREATE or UPDATE
           mappings
48     FOR EACH tuple t IN T typed by map DO
49       WITH t: (b,x1,..,xn,y) // b is either a create or
              update atom
50       FOR EACH rule r DECLARED IN value_map DO
51         \var{T} = \var{T} + ProcessRule(r,xT,(x1,..,xn),y)
52       DONE
53     DONE
54   DONE
55   LET xT=(A,T,mT)
56   IF Eval(phi_src,f_in(xT)) AND Eval(phi_src,f_out(xT)) THEN
57     RETURN xT
58   ELSE
59     RETURN none
60 END FUNCTION
```

Listing 7: F-Alloy Mapping Interpretation pseudo code

kind of rules are expressible in F-Alloy (strict, loose, conditional, step). For each of them it is possible to compute a set of tuples to be added to the instance to be returned, given the input instance ($x_{\text{src}}$) as defined in Listing 8.

```
/*INPUT : r the rule to be processed
          xT instance in which expressions are evaluated
          (x1,...,xn) and y parameters given to the value
               predicate containing the rule
 *OUTPUT: a set of tuples typed by the field assigned in r*/
FUNCTION ProcessRule(Rule r, Instance xT, Tuple (x1,..,xn) ,
     Atom y)
  LET solution=none // tuples to return
  IF r IS strict rule THEN
    WITH r: y.f[expr1]=expr2
    LET val = Eval(expr1,xT)
    LET val2 = Eval(expr2,xT)
    FOR EACH v IN val DO
      FOR EACH v2 IN val2 DO
        solution = solution + (y, v, v2)
      DONE
    DONE
  ELSE IF r IS step rule THEN
    WITH r: all i:Int|range implies y.f[add[i,1]] = expr
    LET val = Eval(range,xT)
    LET t = none // set of tuples
    FOR EACH v IN val DO
      LET val2 = Eval(expr, xT)
      LET i = Eval(add[v,1])
      FOR EACH v2 In val2 DO
        solution = solution + (y, i, v2)
      DONE
    DONE
  ELSE IF r IS conditional rule THEN
    WITH r: expr implies r2
    IF Eval(expr,xT) = True THEN
       solution = ProcessRule(r2, xT, (x1,...,xn), y)
    FI
  ELSE IF r IS loose rule THEN
    WITH  r: y in image.f[expr]
    LET val = Eval(image,xT)
    LET val2 = Eval(expr,xT)
    FOR EACH v IN val DO
      FOR EACH v2 IN val2 DO
        solution = (v,v2,y)
      DONE
    DONE
  FI
  RETURN  solution
END FUNCTION
```

Listing 8: F-Alloy Rule Interpretation pseudo code

Let us analyze the time complexity of interpretation. Let $n$ denote the number of atoms in $x_{\text{src}}$. There are two outer for-loops in the interpretation code. The first outer for-loop (l.17 –l.45) involves a polynomial number of evaluations (in terms of $n$) of Alloy expressions (guard predicates). If we assume that the evaluation of Alloy expressions can be done in polynomial time in $n$ - which can be shown by structural induction - then the overall time for the first outer loop will be at most polynomial in $n$. The second outer for-loop entails a polynomial number of invocations of the `ProcessRule` function. We claim that each call to `ProcessRule` terminates in polynomial time. To see this, note that `ProcessRule` itself evaluates at most a polynomial number of Alloy expressions, hence yielding our claim. We conclude that interpretation takes at most time polynomial in $n$. We finally note that analysis is done via SAT-solving of a Boolean formula whose size is larger than n [17]. Thus analysis takes super-polynomial time in n in the worst case (unless P=NP). . This will be confirmed empirically in the next section.

The following theorem states that the interpretation of f-modules implemented by the pseudo code of Listing 7 conforms to the translational semantics given to F-Alloy in Section 7.

**Theorem 2** *Given an f-module m from $m_{src}$ to $m_{dst}$ and a valid instance $x_{src}$ of $m_{src}$, interpretation returns a valid instance of the augmented module $m_{\mathcal{T}}$ if there exists such an instance, otherwise it returns none.*

*Proof:* The interpretation closely follows the structure of the equation given in Lemma 2, which holds in all augmented module instances. Furthermore the `ProcessRule` function closely follows the definition of the $V$-sets given in Lemma 1. By construction, it will thus yield an $m_{\mathcal{T}}$-instance $x$ in which the equation given in the aforementioned lemma holds. If some valid instance of $m_{\mathcal{T}}$ exists (for given $x_{\mathrm{src}}$), it must be the same as the constructed instance. Therefore the constructed instance must be valid as well and is indeed returned on line 56. If no valid instance of $m_{\mathcal{T}}$ exists (for the given $x_{\mathrm{src}}$) then the constructed instance cannot be valid and nothing will be returned. □

## 9 Evaluation

In this section, we provide a first step toward evaluating our approach by answering through empirical means the following questions:

1. Can F-modules be analyzed more efficiently than Functional Alloy Modules?
2. How does F-Alloy compare to other existing model transformation approaches in terms of execution time?

We note that the following experiments were performed on a machine running an Intel i5 CPU (3.20Ghz) with 16GB of RAM, and that all the files necessary at the reproduction of those experiments can be found online[11].

### 9.1 Speeding Up Analysis using F-Alloy Interpretation

To answer the first question, we compare the time needed to obtain the first valid instance of our `CD2RDBMS` and `CDRefinement` functional Alloy modules using either pure Alloy analysis or a *hybrid analysis* consisting of using Alloy and F-Alloy jointly. Hybrid analysis is first introduced in [11].

The manipulations performed to obtain the first instance are the following:

[11] `https://goo.gl/fyvqRg`

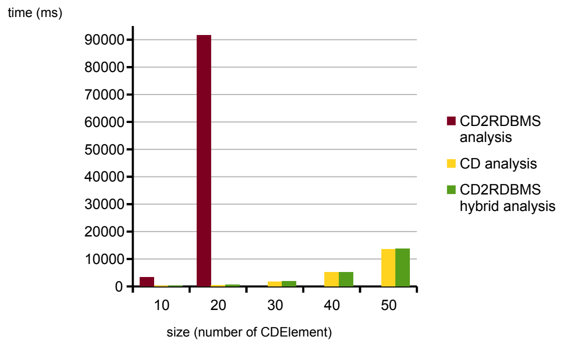


Fig. 15: Plot showing the time taken to complete the Alloy analysis of `CD2RDBMS` and `CD` in function of the scope defined (in term of number of `CDElement`) and to complete the hybrid analysis of the `CD2RDBMS` F-module in function of the size of `CD`-instances given as input

In the case of Alloy analysis: we use the Alloy analyzer (configured to use SAT4J) to generate the first instance conforming to the `CD2RDBMS` and `CDRefinement` Functional Alloy modules.

In the case of hybrid analysis: two steps are performed:

Step 1 consists in generating a valid `CD`-instance (`CD` being the input of both `CD2RDBMS` and `CDRefinement` transformations).

Step 2 consists in feeding this instance along with the F-Alloy specifications of the `CD2RDBMS` and `CDRefinement` transformations to the F-Alloy interpreter.

The time needed by those manipulations is displayed as bar charts in Fig. 15 and Fig. 16 for the `CD2RDBMS` and `CDRefinement` case study, respectively. Measurements were performed for a scope of up to 50 CDElements. The absence of a bar marks a failure to yield instances in reasonable time.

**Observation:** From measurements displayed in Fig. 15 and Fig. 16, we observe that F-Alloy and its interpreter can help in reducing the time needed to analyze both endogenous and exogenous functional Alloy modules. Indeed, the time needed to return a transformation instance using hybrid analysis is essentially reduced to the time needed by the Alloy analyzer to generate instances from the CD module.

### 9.2 Comparing F-Alloy with Existing Model transformations

To answer our second question, we compare the time needed by the F-Alloy interpreter to compute the `CD2RDBMS` and `CDRefinement` transformations to that of

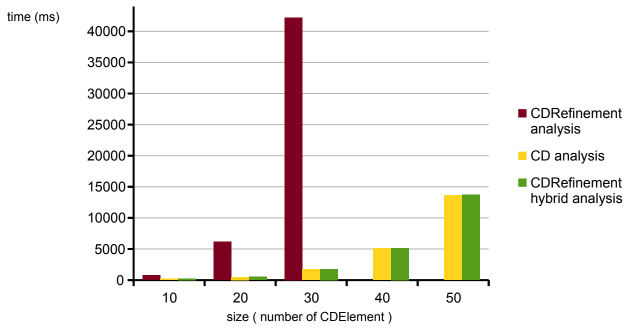


Fig. 16: Plot showing the time taken to complete the Alloy analysis of `CDRefinement` and `CD` in function of the scope defined (in term of number of `CDElement`) and to complete the hybrid analysis of the `CDRefinement` F-module in function of the size of `CD`-instances given as input

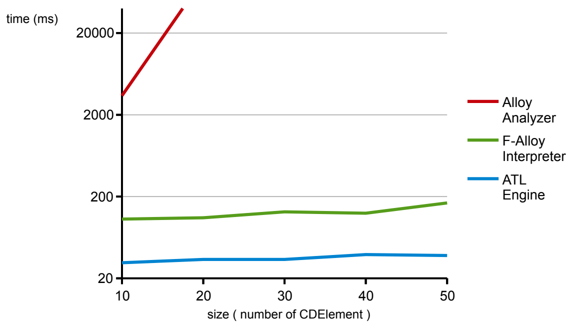


Fig. 17: Plot showing the time taken by ATL, Alloy and F-Alloy to execute the `CD2RDBMS` transformation in function of the size of `CD`-instances given as input

ATL and Henshin, respectively. Input instances used in the following manipulations are those obtained previously by Alloy analysis of the `CD` module. The time taken by ATL and F-Alloy to compute the CD2RDBMS transformation on this sample of input instances is plotted in Fig. 17 while the time taken by Henshin and F-Alloy to compute the CDRefinement transformation is plotted in Fig. 18. In the plot given in Fig. 17, we also show, as a reference, the time needed by the Alloy analyzer to compute the transformations for given CD instances. Computing a transformation with the Alloy analyzed for a fixed input instance is achieved by over-constraining the CD module so that the only input instance it allows is the fixed one. We note that this approach can't be applied to endogenous transformation as over-constraining the CD module doesn't only impact the input instance but also the output instance.

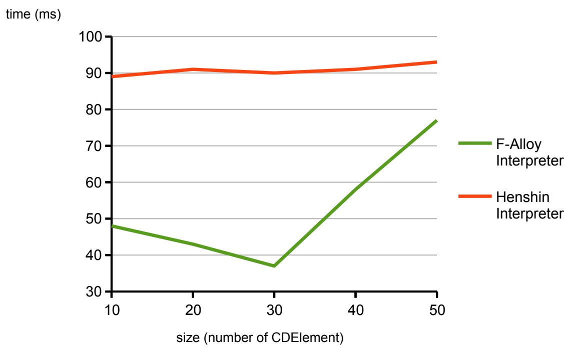


Fig. 18: Plot showing the time taken by Henshin and F-Alloy to execute the `CDRefinement` transformation in function of the size of `CD`-instances given as input

**Observation:** From the measurements plotted in Fig. 17 and Fig. 18, F-Alloy's performance is of the same order than those of ATL and Henshin and considerably outperforms the Alloy analyzer It seems that the Henshin interpreter undergoes an heavy initialization process before actually computing the transformation hence explaining the fact that the F-Alloy interpreter performs faster. We observe that compared to Henshin and ATL which are mature model transformation languages, the time needed by F-Alloy to complete its tasks varies more in function of the input size. This can be explained by the difference of maturity between the two tools and may also be partly attributed to the current lack of effort spent in optimizing the F-Alloy interpreter's code. As a result, it is natural to question the efficiency of F-Alloy interpretation on big input models. This is usually not a problem in practice as F-Alloy is designed as a language allowing the transformation of Alloy instances which are of relatively small size. For this particular problem we see that F-Alloy interpretation outperforms Alloy analysis in terms of execution time, hence widening the applicability of Alloy-based model transformation specifications.

## 10 Discussion and Related Work

**F-Alloy vs. Alloy.** Analysis with the Alloy analyzer is 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 analyzing Alloy modules, hence turning Alloy analysis into a non-practical solution for model transformation computation. Contrary to this we show that model transformations specified in F-Alloy can be computed in polynomial time through a process called interpretation (see Section 8). Analysis of F-Alloy specification for validation's sake is still possible using the translational semantics defined: any F-module can be automatically translated into an Alloy module expressing the same transformation. Furthermore, we recall that interpretation of modules written in F-Alloy relieves the analyst of having to determine proper scopes for the signatures itself a non-trivial problem. Concerning scalability, we recall that F-Alloy interpretation yields Alloy instances and note that there exists an untold limit in the size of Alloy instances.This limitation will cause the Alloy API to fail parsing big transformation instances built by interpretation ($\sim$ 1000 atoms). Parsing interpretation-built instance is only necessary to enable the evaluation of expression in the said instance. If this feature is not needed, then this scalability issue is usually not a problem.

**Related work on model transformation languages.** We can consider the F-Alloy language as a relational model transformation language. Relational model transformation languages (such as those given in [2], [23] and [14]) are those where the main concept is that of a mathematical relation [8]. Note that in F-Alloy the mathematical relations, represented by mappings, are in fact injective functions. In their pure form (e.g., [2]) relational specifications are not executable. In other cases (e.g., [23]) they are executable in principle but still lack proper tool support. In the case of QVTr there are some tools that execute QVTr specifications but none of them take into account all the features of the QVTr language yet. Note however that the QVTd project is working on a soon-to-be-finished implementation of QVTr[12]. 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 QVTr ([21]). In this paper we have shown that F-Alloy specifications are efficiently executable.

In the case of endogenous transformations, in-place transformations – defining how to obtain the output of the transformation via operations to be applied on a given input – are often considered as syntactic sugar for their out-place counterparts – defining how to build the output from scratch given the input. To illustrate this trend, the ATL refining mode, which allows the specification of in-place transformation, is executed as an out-place transformation (input-model not modified, elements copied from input to output) by the ATL 2004 compiler [13]. Our motivation to make in-place constructs an integral part of our functional Alloy modules, and consequently F-Alloy is to directly provide a formal ground to F-Alloy's in-place syntax, thus preventing any ambiguities in the semantics of each operation. Another language that natively supports endogenous in-place transformations is Henshin[5], which was used in the case study section as well as in the evaluation section and supports a formal graph transformation semantics.

One distinguishing feature of F-Alloy is that it is based on the formal language Alloy (we have defined F-Alloy reusing Alloy's syntax and semantics definition). Not all model transformation languages are based on formal languages For instance the previously mentioned model transformation language ATL [18] was defined semi-formally. A formal semantics in terms of rewriting logics was later given by [30]. 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 [25,26], used in the case study section, for which the authors of [15] describe an approach to show conformance of an existing implementation to the formal semantics. We have shown in Theorem 2 that the interpretation process (described using pseudo code) conforms to the translational semantics we have given to F-Alloy. A certain degree of confidence on F-Alloy interpretation's correctness is thus reached, provided that the actual interpreter's implementation conforms to the given pseudo-code.

**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. [6] 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 similarly use Alloy's analysis features to verify model transformations represented in F-Alloy. Furthermore, as we show in the evaluation section, in

[12] `https://projects.eclipse.org/projects/modeling.mmt.qvtd`

[13] we note that this is not the case anymore since the release of the ATL 2009 compiler

certain cases we can speed up the analysis using interpretation.

## 11 Conclusion and Future Work

In this paper we have introduced the notion of functional Alloy module as an Alloy module representing either an exogenous or an endogenous transformation. We have defined a sublanguage 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 two model transformations, an exogenous one called `CD2RDBMS`, and an endogenous one called `CDRefinement`. Several other case studies have been implemented using F-Alloy [14], but a more thorough evaluation of the approach by implementing further case studies is planned.

F-Alloy's semantics has been defined as a translation to Alloy, hence making the Alloy analysis of any F-Alloy specification possible. This contrasts with other approaches where a separate formal verification method is provided.

One of the potential limitations of F-Alloy is the prerequisite of being familiar with Alloy in order to use the language. We plan in future work to provide an alternative graphical syntax to make F-Alloy more accessible. Furthermore, work is underway to make languages defined in the Lightning tool [1] – a language workbench embedding the F-Alloy interpreter – usable by non-Alloy experts via a web interface. This interface will support the creation and edition of instances using their concrete syntax (a domain specific visualisation) as well as exercising model transformations on them.

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 synchronization between models. Future work will determine if we can make our approach bidirectional. This has already been achieved for existing relational model transformation languages such as QVTr but also graph based approaches such as triple graph grammars.

As seen in the evaluation section, the execution time of F-Alloy specifications using our own interpreter is less predictable than the one obtained by mature model transformation tools like ATL and Henshin. To make F-Alloy into a practical and appealing solution to the design of seamlessly verifiable model transformations, more work on optimizing our tooling is thus needed. Finally, we want to explore whether the more general definition of functional Alloy modules, parametrized by two transformation functions defining the domain and range of the function denoted, has applications beyond the definition of endogenous and exogenous model transformations.

[14] `http://lightning.gforge.uni.lu/examples`

## 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*, pages 215–239, 2003.
3. Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software & Systems Modeling*, pages 69–86, 2010.
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. Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *Model Driven Engineering Languages and Systems*, pages 121–135. 2010.
6. Luciano Baresi and Paola Spoletini. On the use of Alloy to analyze graph transformation systems. In *Graph Transformations*, pages 306–320. 2006.
7. 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, 2006.
8. 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*, pages 1–17, 2003.
9. Loïc Gammaitoni and Pierre Kelsen. F-Alloy: An Alloy based model transformation language. In *Theory and Practice of Model Transformations*, pages 166–180. 2015.
10. Loïc Gammaitoni, Pierre Kelsen, and Christian Glodt. Designing languages using lightning. In *International Conference on Software Language Engineering*, pages 77–82, 2015.
11. Loïc Gammaitoni, Pierre Kelsen, and Qin Ma. Agile validation of higher order transformations using f-alloy. In *Theoretical Aspects of Software Engineering (TASE), 2016 10th International Symposium on*, pages 125–131. IEEE, 2016.
12. Loïc Gammaitoni, Pierre Kelsen, and Fabien Mathey. Verifying modelling languages using lightning: a case study. In *11th MoDeVVa Workshop: Model-Driven Engineering, Verification and Validation*, pages 19–28. 2014.
13. Geri Georg, Jores Bieman, and Robert B France. Using alloy and uml/ocl to specify run-time configuration management: A case study. In *pUML*, pages 128–141. Citeseer, 2001.
14. Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In *Graph Transformation*, pages 90–105. 2002.
15. Holger Giese, Stephan Hildebrandt, and Leen Lambers. Toward bridging the gap between formal semantics and implementation of triple graph grammars. In *Proceeding of the 7th MoDeVVa Workshop: Model-Driven Engineering, Verification, and Validation*, pages 19–24, 2010.

16. Frank Hermann, Hartmut Ehrig, Fernando Orejas, and Ulrike Golas. Formal analysis of functional behaviour for model transformations based on triple graph grammars. In *International Conference on Graph Transformation*, pages 155–170. Springer, 2010.
17. Daniel Jackson. *Software abstractions*. MIT Press Cambridge, 2012.
18. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of computer programming*, pages 31–39, 2008.
19. Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. *Empirical study of the anatomy of modern sat solvers*, volume 6695 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 343–356. 2011.
20. K Lano. Catalogue of model transformations.
21. Nuno Macedo and Alcino Cunha. Implementing QVT-R bidirectional model transformations using Alloy. In *Fundamental Approaches to Software Engineering*, pages 297–311. 2013.
22. Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.
23. OMG. Meta object facility query/view/transformation specification, 2011.
24. Andy Schürr. Specification of graph translators with triple graph grammars. In *in Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D*. Springer, 1995.
25. Andy Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163, 1995.
26. Andy Schürr and Felix Klar. 15 years of triple graph grammars. In *Graph Transformations*, pages 411–425. 2008.
27. Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.
28. Mana Taghdiri and Daniel Jackson. A lightweight formal analysis of a multicast key management scheme. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 240–256. Springer, 2003.
29. Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. Refining models with rule-based model transformations. 2011.
30. Javier Troya and Antonio Vallecillo. A rewriting logic semantics for ATL. *Journal of Object Technology*, pages 1–29, 2011.
31. Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

## A F-Alloy to Alloy translation procedures

In this appendix, we provide procedures in pseudo-code describing how each type of constraint identified in section 7 is automatically generated.

We recall that the notation in which the pseudo-code is provided has the following properties:

- Keywords structuring the code are highlighted in bold and are written in uppercase.
- Variables names are highlighted in bold and are written in lowercase.
- Text and numbers are highlighted in italic.
- Function names (in declarations and in calls) are highlighted in bold and italic.
- Expressions of the form "`WITH var: description`" consists in providing a syntactic description of the variable, consisting of syntactical constructs and terminals. Each syntactical construct counts as a variable declaration, the variable being initialized following the value of the described variable `var`. Each of such "syntactic variable" is highlighted in blue, to differentiate them from terminals and other possibly already defined variables.

### A.1 Map Disjunction

The `Map Disjunction` function, given in Listing 9, first identifies (via nested loops) each pair of mappings (different from one another) having ranges typed by the same signature. For each such pair, a constraint is written enforcing that the intersection between atoms in the range of those mappings is empty, i.e., no disjoint mappings have overlapping ranges.

```
FUNCTION MapDisjunction (F-Module m)
  WRITE fact MapDisjunction{
  LET m_dst= m.getImportedModule(2) //2nd open statement in m
  IF m_dst= none THEN //endogenous case
    m_dst= m.getImportedModule(1) //1st open statement in m
  FI
  FOR EACH signature y DECLARED IN m_dst DO
    LET  maps = FIND ALL mappings having y as range IN m
    FOR EACH mapping map IN maps DO
      WITH map: name : x1 -> .. -> xn -> y
      FOR EACH mapping map2 IN maps DO
        WITH map2: name2: x1' -> .. -> xn' -> y
        IF name != name2 THEN
          WRITE name[x1,..,xn] & name2[x1',..,xn'] = none
        FI
      DONE
    DONE
  DONE
  WRITE }
END FUNCTION
```

Listing 9: Pseudo code describing how Map Disjunction constraints can be generated from an F-module m

### A.2 Map Injectiveness

The `Map Injectiveness` function, given in Listing 10, iterates over each mapping declared in the f-module and writes an Alloy constraint enforcing that each element in the range of a given map has at most one preimage.

```
FUNCTION MapInjectiveness (F-Module m)
  WRITE fact MapInjectiveness{
  LET maps = FIND ALL CREATE and UPDATE mappings DECLARED IN m
  FOR EACH mapping map IN maps DO
    WITH map: name : x1 -> .. -> xn -> y
    LET sig= signature in which map is declared
    WRITE all y:y | lone sig.name.y
  DONE
  WRITE }
END FUNCTION
```

Listing 10: Pseudo code describing how Map Injectiveness constraints can be generated from an F-module m

### A.3 Predicate Association

The `Predicate Association` function , given in Listing 11, iterates over all mappings declared in the f-module and writes an Alloy constraint enforcing that for each mapping, each combination of atoms typed after the domain of the mapping is mapped to an atom in the range if and only if the said combination satisfies the guard. Also, the value predicate should be satisfied given as parameters the satisfying combination of domain atoms and their associated atom in the range.

As delete mappings don't have range, the function simply writes an Alloy expression that ensures that combinations of atoms in the domain are to be part of the delete mapping if and only if they satisfy the guard.

```
FUNCTION PredicateAssociation (F-Module m)
  WRITE fact PredicateAssociation{
  LET  maps =  FIND ALL mappings DECLARED IN m
  FOR EACH map IN maps DO
    WITH map: name : x1 -> .. -> xn -> y
    LET sig= FIND signature in which map is declared
    IF sig=CREATE OR sig=UPDATE THEN
      WRITE
        all x1:x1|..|all xn:xn{
        (guard_name[x1,..,xn] and
         one sig.name[x1,..,xn] and value_name
              [x1,..,xn,sig.name[x1,..,xn]])
        or ( not guard_name[x1,..,xn] and no
              sig.name[x1,..,xn]) }
      WROTE
    ELSE IF sig=DELETE THEN
      WRITE
        all x1:x1{
        (guard_name[x1] and x1 in DELETE.name
        or (not guard_name[x1] and x1 not in DELETE.name)
      WROTE
    FI
  DONE
  WRITE }
END FUNCTION
```

Listing 11: Pseudo code describing how Predicate Association constraints can be generated from an F-module m

### A.4 Minimal Assignment

The `Minimal Assignment` function, given in Listing 13, iterates over maps declared in the f-module. For each map, it iterates over fields declared in the signature typing its range. It then writes a constraint limiting the cardinality of that field to be that expected from the assignment defined by strict and step rules of the associated value predicate and by loose rules of other value predicates. To do so, it relies on the `COUNT` function which returns an Alloy expression yielding the expected cardinality.

```
FUNCTION MinimalAssignment (F-Module m)
  FOR EACH mapping map DECLARED IN m DO
    WITH map: name: x1 -> .. -> xn -> y
    LET sig= FIND signature IN which map is declared
    LET val= FIND predicate named value_name
    LET paramY = last parameter of val (typed by y)
    FOR EACH field f DECLARED IN signature y DO
      LET constraint = none
      FOR EACH rule r CONTAINING f IN val DO
        IF r  NOT CONTAINING loose rule THEN
          IF constraint= none THEN
            constraint= COUNT(r) //COUNT defined bellow
          ELSE
            constraint= add[ COUNT(r) , constraint]
          FI
        FI
      DONE
      APPEND TO val block
        #paramY.f=constraint
      APPENDED
      LET flag= false
      FOR EACH mapping map2 IN m DO
        WITH map2: name': x1' -> .. -> xn' -> y'
        LET val2= FIND predicate named value_name'
        LET paramY' = last parameter of val2 (typed by y')
        IF name != name' THEN
          FOR EACH rule r IN val2 DO
            IF r CONTAINS loose rule THEN
              flag= true
              APPEND TO val2 block
                #f.paramY'=COUNT(r])
              APPENDED
            FI
          DONE
        FI
      DONE
      IF sig= UPDATE THEN
        IF NO rule r CONTAINS f IN any value predicate of m
             THEN
          APPEND TO val2 block
            paramY'.f= x1'.f
          APPENDED
        FI
      ELSE IF flag= false
        APPEND TO val block
          paramY.f= none
        APPENDED
      FI
    DONE
  DONE
END FUNCTION
```

Listing 12: Pseudo code describing how Minimal Assignment constraints can be generated from an F-module m

```
/*Utility function returning an integer-valued Alloy
expression corresponding to the expected cardinality
of the field assigned by a given rule. */
FUNCTION COUNT[rule r]
  IF r IS a strict rule THEN
    WITH r: y.f[expr1]=expr2
    IF expr1= none OR expr1 IN INT THEN
      RETURN #expr2
    ELSE // multiply #expr1 and #expr2
      RETURN mult[#expr1,#expr2]
    FI
  FI
  IF r IS a step rule THEN
    WITH r: all i:Int|range implies y.f[add[i,1]] = expr
    WITH range: i > expr1 and i < expr2
    LET r= max[sub[#expr2,#expr1]+0]
    RETURN mult[#expr,c]
  FI
  IF r IS a conditional rule THEN
    WITH c: expr1 implies r2
    RETURN not expr1 implies 0 else COUNT(r2)
  FI
  IF r IS a loose rule THEN
    WITH r: y in image.name[expr]
    IF expr= none OR expr IN INT THEN
      RETURN #image
    ELSE
      RETURN mult[#expr,#image]
    FI
  FI
END FUNCTION
```

Listing 13: Pseudo code of the COUNT function used by the `MinimalAssignment` function

### A.5 IO Disjunction

The `IO Disjunction` function iterates (via nested loops) over pairs of mappings (not forcibly different), and writes an Alloy expression ensuring that the domain of one is disjoint from the range of the other.

```
FUNCTION IODisjunction (F-Module m)
  WRITE fact IODisjunction{
    FOR EACH mapping map1 DECLARED IN m DO
    WITH  map1: name: x1 -> .. -> xn -> y
    LET sig1=FIND signature in which map1 is declared
    FOR EACH mapping map2 DECLARED IN m DO
      WITH map2: name': x1' -> .. -> xn' -> y'
      LET sig2=FIND signature in which map2 is declared
      FOR EACH i IN RANGE [1, n]
        IF xi' = y THEN
          WRITE no sig1.name[x1,..,xn] & sig2.name'
                  [x1',..,xi-1'].xi+1'.(..).xn'.y'
        FI
      DONE
    DONE
  DONE
  WRITE }
END FUNCTION
```

Listing 14: Pseudo code describing how IO Disjunction constraints can be generated from an F-module m

## A.6 Constraint Framing

The first part of the `Constraint Framing` function, given in Listing 15, writes two Alloy expressions defining the set of atoms contained in the input and output instance of the f-module, respectively.

The second part rewrites facts of the input module as predicates and writes in a fact of the translated f-module an Alloy expression enforcing those predicates to hold given the set of all input and output atoms as parameters, respectively.

```
FUNCTION ConstraintFraming (F-Module m)
  WRITE fact ConstraintsFraming{
  LET in= let input= univ - ( CREATE + DELETE + UPDATE
  LET out= let output= univ - ( CREATE + DELETE + UPDATE
  FOR EACH mapping map DECLARED IN m DO
    WITH map: name: x1 -> .. -> xn -> y
    LET sig= signature in which map is declared
    IF sig= CREATE OR sig= UPDATE THEN
      in+= + sig.name[x1,..,xn]
    FI
    IF sig= UPDATE THEN
      out+= + sig.name.x1.(..).xn
    FI
    IF sig= DELETE THEN
      out+= + sig.name
    FI
  DONE
  in+= )
  out+= )
  WRITE in + out
  LET m_src= m.getImportedModule(1) //1st open statement in m
  REPLACE facts by predicates in m_src
  LET preds be the set of added predicates
  FOR EACH predicate p IN preds DO
    LET n = p's name
    WRITE n[input] and n[output]
  DONE
  WRITE }
END FUNCTION
```

Listing 15: Pseudo code describing how Constraint Framing constraints can be generated from an F-module m

## A.7 Minimum Output

The `Minimum Output` function, given in Listing 16, iterates over all signatures declared in the output module and writes an Alloy expression enforcing the set of all atoms typed by each of these signatures to be equal to the union of atoms in the range of mappings whose range is typed by the given signature.

```
FUNCTION MinimumOutput (F-Module m)
  WRITE fact MinimumOutput{
  m_dst= m.getImportedModule(2) //2nd open statement in m
  FOR EACH signature y DECLARED IN m_dst DO
    LET maps = FIND ALL mapping IN m having vary as range
    LET constraint= y=
    IF maps is empty THEN
      constraint += none
    FI
    FOR EACH map IN maps DO
      WITH map: name: x1 -> .. -> xn -> y
      LET sig= FIND signature in which map is declared
      constraint+= sig.name[x1,..,xn]
      IF NOT last iteration of this loop THEN
        constraint += +
      FI
    DONE
    WRITE constraint
  DONE
END FUNCTION
```

Listing 16: Pseudo code describing how Minimum Output constraints can be generated from an F-module