Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Agile validation of model transformations using compound F-Alloy specifications

Loïc Gammaitoni [*], Pierre Kelsen, Qin Ma

University of Luxembourg, Luxembourg

A R T I C L E   I N F O

Article history:
Received 3 January 2017
Received in revised form 26 June 2017
Accepted 2 July 2017
Available online xxxx

Keywords:
Model transformation
Validation
Alloy
Compound model transformations
Higher-order transformations

A B S T R A C T

Model transformations play a key role in model driven software engineering approaches. Validation of model transformations is crucial for the quality assurance of software systems to be constructed. The relational logic based specification language Alloy and its accompanying tool the Alloy Analyzer have been used in the past to validate properties of model transformations. However Alloy based analysis of transformations suffers from several limitations. On one hand, it is time consuming and does not scale well. On the other hand, the reliance on Alloy, being a formal method, prevents the effective involvement of domain experts in the validation process which is crucial for pinpointing domain pertinent errors. Those limitations are even more severe when it comes to transformations whose input and/or output are themselves transformations (called compound transformations) because they are inherently more complex.

To tackle the performance and scalability limitations, in previous work, we proposed an Alloy-based Domain Specific Language (DSL), called F-Alloy, that is tailored for model transformation specifications. Instead of pure analysis based validation, F-Alloy speeds up the validation of model transformations by applying a hybrid strategy that combines analysis with interpretation. In this paper, we formalize the notion of "hybrid analysis" and further extended it to also support efficient validation of compound transformations.

To enable the effective involvement of domain experts in the validation process, we propose in this paper a new approach to model transformation validation, called Visualization-Based Validation (briefly VBV). Following VBV, representative instances of a to-be-validated model transformation are automatically generated by hybrid analysis and shown to domain experts for feedback in a visual notation that they are familiar with. We prescribe a process to guide the application of VBV to model transformations and illustrate it with a benchmark model transformation.

1. Introduction

Alloy [16] is a formal specification language based on first order relational logic. It was developed to support agile modeling of software designs. Alloy models can be automatically analyzed by the Alloy Analyzer to check the correctness of software designs. By providing immediate visual feedback to users, the use of Alloy is meant to facilitate identifying design errors early in the software design life cycle.

[*] Corresponding author.
E-mail addresses: loic.gammaitoni@uni.lu (L. Gammaitoni), pierre.kelsen@uni.lu (P. Kelsen), qin.ma@uni.lu (Q. Ma).

http://dx.doi.org/10.1016/j.scico.2017.07.001
0167-6423/© 2017 Elsevier B.V. All rights reserved.

In the context of model driven software engineering, Alloy and its accompanying tool the Alloy Analyzer have been used to validate properties of models [1,6] and model transformations [2,4]. However, the analysis of model transformation specifications expressed in Alloy has some important limitations. Alloy employs a SAT based back-end and performs bounded exhaustive search during analysis. The effective execution of an Alloy analysis heavily relies on the specification of the bounds, called *scopes* in Alloy, which define the maximal number of elements of each type in instances of an Alloy specification. The Alloy Analyzer populates instances with numbers of elements of a given type up to the given scope and performs an exhaustive search in this finite space of model instances to find a solution or a counter-example. As a consequence, two major problems hamper the practical application of Alloy for model transformations:

1. The task of deciding minimal scope that is sufficient to find a solution or a counter-example is by itself non-trivial (in fact it is undecidable). This becomes even more problematic for complex Alloy specifications such as those expressing *compound model transformations*, i.e., transformations whose input and/or output are themselves transformations.
2. With the increase of complexity of Alloy specifications for model transformations, the corresponding minimal sufficient scopes increase subsequently. Despite many advances in the performance of SAT solvers, the analysis can still become quite time consuming especially when a specification requires a larger scope to find a suitable instance, increasing scopes leading to a combinatorial explosion.

In previous work [10], we proposed an Alloy-based Domain Specific Language (DSL), called F-Alloy, that is tailored for model transformation specifications. Instead of pure SAT-based analysis which can be computationally heavy, a lightweight hybrid strategy to the analysis of F-Alloy specifications was envisioned. However, only model to model transformations can be expressed in F-Alloy in [10].

The first contribution of this paper is an extension of F-Alloy to enable the specification and efficient computation of compound transformations.

As the second contribution, we formally define *hybrid analysis* and explain how hybrid analysis integrates the interpretation of *F-modules* – i.e., modules written in F-Alloy specifying basic or compound transformations – with the SAT-based analysis of Alloy modules in a seamless fashion. Furthermore, the notion of hybrid analysis is also generalized to the case of compound transformations.

Those additions give rise to a third contribution, a *Visualization-Based Validation* (*VBV*) approach enabling domain experts to play an integral role in the process of model transformation validation. Note that Alloy natively offers a visualization of model instances as graphs, with each element of the instances being represented by a node and relations between those by edges. Yet it has been shown in [9] that this native visualization falls short in providing an intuitive visual feedback, hence preventing effective involvement of domain experts. We showcase the VBV approach by letting a domain expert (without F-Alloy knowledge) validate an F-Alloy specification of the benchmark model transformation CD2RDBMS [5].

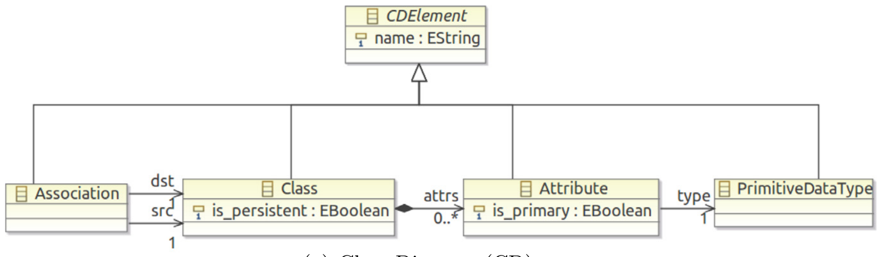
The rest of the paper is structured as follows. In Sect. 2 we introduce the Alloy specification language and its accompanying analysis tool. In Sect. 3 we review F-Alloy for transformation specifications and extend the language to also support compound transformations. We define the hybrid analysis strategy in Sect. 4 by integrating Alloy's analysis with F-Alloy's interpretation. We then present the VBV model transformation validation approach as an application of this newly defined hybrid analysis in Sect. 5, and illustrate this approach in Sect. 6 in the context of validating the benchmark CD2RDBMS transformation. We evaluate in Sect. 7 the performance of hybrid analysis in general and the VBV approach in particular. We discuss related work in Sect. 8 and present concluding remarks and future work in the final section.

Extension statement: A previous version of this work has been published in the proceedings of the 10th International Symposium on Theory and Aspects of Software Engineering (TASE 2016) [12]. Thanks to the valuable comments and suggestions from the anonymous reviewers of TASE 2016, we here present an extended and improved version of our work. Notably, we extend Sect. 3 and Sect. 4 with formal definitions, complete proofs, and additional illustrative examples. Moreover, two new pieces of work are added in this version, namely, the introduction of a visualization-based approach for model transformation validation (VBV) in Sect. 5 and the application of VBV to the validation of an F-Alloy implementation of the benchmark CD2RDBMS transformation (reported in Sect. 6). The rest of the sections are updated and completed as well to work with the new sections as a consistent whole.

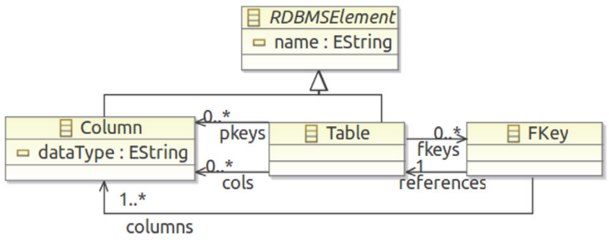
Please also note that in the previous version [12], compound transformations were mistakenly called higher-order transformations. Higher-order transformations, as introduced in [22], take a metamodel of a transformation language (e.g., the ATL metamodel) as input or output. We note that such metamodel can be specified in an Alloy module and thus the expression of higher-order transformation is natively supported by F-Alloy. In contrast, compound transformations take a transformation model (e.g., an F-module) as input or output.

2. Alloy and the Alloy analyzer

Alloy [16] is a textual modeling language based on first order logic and relational calculus. It combines the precision of formal specifications with powerful automatic analysis features offered by the accompanying analyzer tool.



(a) Class Diagram (CD)



(b) Relational Database Management System (RDBMS)

Fig. 1. CD and CD2RDBMS metamodels (adapted from [5]).

2.1. Metamodels in Alloy

In Alloy, a metamodel is defined by an *Alloy module*, which is saved as a `.als` file and contains the corresponding Alloy specifications. An Alloy module may import other Alloy modules. The central constituents of an Alloy module are *signatures* (similar to the notion of "classes" in metamodels), *fields* (similar to the notion of "associations" in metamodels), and *facts* (similar to the notion of invariants in metamodels).

```
module CD
open util/boolean

abstract sig CDElement{
  name: disj String
}
sig Class extends CDElement{
 attrs: disj set Attribute,
 is_persistent: Bool
}{
  //persistent class should declare a primary
       attribute
  is_persistent=True implies True in attrs.
       is_primary
}
sig Attribute  extends CDElement {
 is_primary: Bool,
 type: PrimitiveDataType
}{
  //no orphan attributes
  this in Class.attrs
}
sig Association  extends CDElement{
 src: Class,
 dest: Class
}
abstract sig PrimitiveDataType{} extends
     CDElement
run {} for 5
```

Listing 1: CD.als: Alloy specification of the class diagram metamodel.

```
module RDBMS

abstract sig RDBMSElement{
  name: disj seq String
}
sig Table extends RDBMSElement{
  cols: disj set Column,
  pkeys: set 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
}
sig FKey{
  references: one Table,
  columns: some Column
}{
// no orphan FKeys
  this in Table.fkeys
//fkey columns are in owning table
  columns in this.~fkeys.cols
}
run {} for 5
```

Listing 2: RDBMS.als: Alloy specification of the relational database management system metamodel.

A signature **sig** A{} defines a basic type A and represents a set of A-*atoms*. The **extends** keyword can be used in signature declarations to introduce subtypes (and consequently subsets of atoms) of another signature. Fields can be declared inside signatures to define relations over sets of atoms. For example **sig** A{f:B −> C} defines a ternary relation $f \subseteq A \times B \times C$. In addition, *functions* and *predicates* can be declared to define named parameterized relational or boolean expressions that can be invoked in other places. Any expression contained in a *fact* is bound to always hold. Finally *assertions* are named constraints that can be checked to look for counter-examples.

As an example, the class diagram (CD) and relational database management system (RDBMS) metamodel depicted in Fig. 1(a) and 1(b) are defined by the Alloy modules given in Listing 1 and 2, respectively. We note that those Alloy modules contain extra well-formedness constraints that do no appear in Fig. 1(a) and 1(b).

A model *conforming* to an Alloy module a is a set of atoms (typed by signatures of a) and tuples of atoms (typed by fields of a) so that all the constraints (mainly expressed through facts) of a are satisfied. In the Alloy community, those conforming models are called *valid instances*. We further use the term *a-instances* to refer to models conforming to module a.

A run command, declared inside an Alloy module defines a configuration of the finite space of instances in which the Alloy Analyzer looks for a valid instance. For example, in Listing 1, this finite space is defined by the run command “**run** \{\} for 5”, in the last line of the module. Here 5 is a global scope that applies to all the top-level signatures (i.e., signatures not extending another one). With such a scope set, the Alloy Analyzer will only search in a finite space consisting of instances with up to 5 atoms of each top-level signatures for those satisfying the invariants (i.e., facts) of the module. Note that atoms of a signature extending another signature are also atoms of the latter. Please also note that, in the run command, a predicate can be specified, too, to enforce extra restrictions on the valid instances. In such a case, the set of instances found by the Alloy Analyzer do not only reside in the scope, satisfy the invariants, but also satisfy the additional predicate. In our examples (Listing 1 and Listing 2) this predicate is left blank by en empty pair of curly brackets.

Notation 1 *(Alloy analysis).* We denote the set of instances obtained by Alloy analysis of an Alloy module a within a global scope $s \in \mathbb{N}$ by $\mathcal{Z}(a, s)$.

2.2. Model transformations in Alloy

Alloy modules can also be exploited to specify model transformations [2,4]. In this paper we focus on the study of exogenous model transformations[20] that take one source model as input and produce one target model as output. The source model conforms to a source metamodel and the target model conforms to a target metamodel. Suppose the source metamodel is specified in an Alloy module a_1 and the target metamodel in an Alloy module a_2. The Alloy module a representing the transformation *imports* both a_1 and a_2, and specifies a set of relations and predicates to express the transformation rules.

The Alloy Analyzer analyzes the Alloy module a and searches (within a scope) for all pairs of instances of a_1 and a_2 such that the relations and predicates corresponding to the transformation rules hold. Each pair of instances represents an input model of the transformation and the corresponding output model. The idea of using Alloy for specifying model transformations is appealing. In addition to automatic input and output model instantiation, one can also exploit Alloy for various analysis tasks such as the validity of output models, properties of input–output model pairs, and so on. However, depending on the size of a_1 and a_2, and the scope, the analysis can become very resource consuming. For example, the Alloy Analyzer fails to analyze the class diagram (CD) to relational database management systems (RDBMS) benchmark transformation [5] in a scope of 20, on a computer with a Quad-Core Intel i7 CPU and 8 GB memory. A memory overflow error is reported before a result can be found.

In order to fully leverage the analysis power offered by Alloy within reasonable computing resources, in a previous work [10], we proposed an Alloy-based Domain Specific Language (DSL), called F-Alloy, that is tailored for model transformation specifications and efficient model transformation interpretation.

3. An interpretable model transformation language: F-Alloy

The F-Alloy language [10] is an Alloy-based Domain Specific Language (DSL) for model transformations. Syntactically, F-Alloy resembles Alloy in the sense that every F-module (i.e., module expressed in F-Alloy) is also a valid Alloy module. F-module specifications are written in `.fals` files, hence differing from plain Alloy modules (stored in `.als` files). The semantics of F-Alloy is defined by a translation from F-modules to plain Alloy modules. This translation adds constraints to make explicit the properties that are inherent to model transformations, such as the output model of a transformation contains and only contains the elements that are transformed from the input model.

3.1. Syntax of F-Alloy as given in [10]

```
    fmodule ::= module qualName import paragraph*
     import ::= open qualName open qualName
  paragraph ::= sigDecl | guardDecl | valueDecl
    sigDecl ::= one sig CREATE { mappingDecl,* }
mappingDecl ::= name : qualName( ->qualName)+,
  guardDecl ::= pred guard_name param { expr* }
  valueDecl ::= pred value_name param { iexpr* }
      param ::=   [(name : expr,)+]
   qualName ::= [this/] (name/)* name
```

Listing 3: An overview of F-Alloy BNF grammar.

Listing 3 gives an overview of F-Alloy’s grammar. It uses the following BNF notations:

- x^* denotes 0 or more occurrences of x;
- x^+ denotes at least one occurrence of x;
- $x|y$ denotes the occurrence of either x or y;
- $[x]$ denotes 0 or one occurrence of x.

In addition, we abuse the star and plus notation so that:

- $x,^*$ denotes 0 or more comma-separated occurrences of x;
- $x,^+$ denotes at least one comma-separated occurrence of x;

We differentiate terminals from syntactic constructs by representing the former in bold face (which allows to ease the reading of the BNF by getting rid of angle brackets in the references of syntactical constructs). We also note that:

- `name` is used to denote a sequence of alphanumerical symbols (a–z, A–Z, 0–9) starting by an alphabetic character (a–z, A–Z).
- `expr` is used to denote Alloy expressions as defined in [16]
- `iexpr` is used to denote the subset of Alloy expressions which can be handled by the F-Alloy interpretation [10].

The BNF grammar is further completed by a set of well-formedness (WF) constraints:
ImportWF In the `import` rule, the two `qualName` should refer to two distinct Alloy modules respectively. We call those two modules the *input module* and the *output module* as they represent the input and output metamodels of the specified transformation. We note that, just like in Alloy, import hierarchies are acyclic, i.e., a module cannot be imported by itself nor by any of its recursively imported modules.
SigWF There is exactly one signature declaration (produced by the `sigDecl` rule) with name "CREATE". This signature aggregates all the model transformation rules in terms of mapping declarations (produced by the `mappingDecl` rule).
MappingWF For each mapping declaration (produced by the `mappingDecl` rule), the last occurrence of `qualName` should refer to a signature defined in the output module. It is called the *range* of the mapping. Other occurrences of `qualName` refer to signatures defined in the input module. This sequence of signatures constitute the *domain* of the mapping. A mapping indicates that atoms in the input model typed by the domain will be transformed into atoms in the output model typed by the range.
GuardWF For each mapping declaration produced by the `mappingDecl` rule, there exists exactly one guard declaration (produced by the `guardDecl` rule) whose name is the name of the mapping prefixed by "guard_". The guard declaration defines a predicate describing what should hold on the input model in order for the mapping to take effect. The parameters of the guard predicate should be typed by the domain of the mapping.
ValueWF Similarly, for each mapping declaration produced by the `mappingDecl` rule, there exists exactly one value declaration (produced by the `valueDecl` rule), whose name is the name of the mapping prefixed by "value_". The value declaration defines a predicate describing what should be in the output model after applying the mapping. The parameters of the value predicate should be typed by the domain and range of the mapping.

```
module Class2Table
open CD
open RDBMS

one sig CREATE{
   class2table: Class  ->  Table,
}
pred guard_class2table(c:Class){
   c.is_persistent=True
}
pred value_class2table(c:Class ,  t:Table){
   t.name[0]=c.name
}
```

Listing 4: A simple class to table transformation.

As an example, consider a simple model transformation from Class Diagram to Relational Database Management System with only one transformation rule: for each persistent class present in the input class diagram, a table is created in the output model and the table carries the same name as the class. Listing 4 presents the corresponding specification in F-Alloy. The F-module `Class2Table` imports two Alloy modules: `CD`, the input module defining the Class Diagram metamodel (Listing 1), and `RDBMS`, the output module defining the Relational Database Management System metamodel (Listing 2). One mapping is declared in the `CREATE` signature called `class2table` to represent the unique transformation rule. The domain of the mapping `class2table` is the `Class` signature defined in the input module and the range is the `Table`

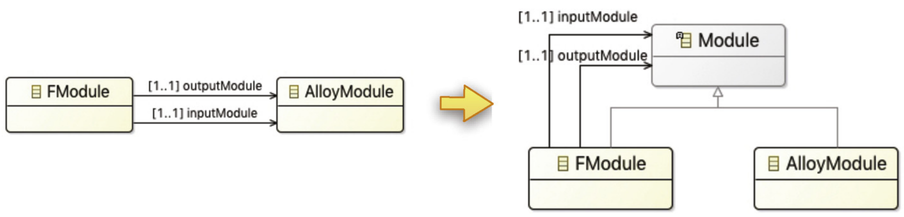


Fig. 2. Extending F-Alloy (from left to right) to support both basic and compound F-modules.

signature defined in the output module, to indicate that atoms of `Class` will be transformed into atoms of `Table`. For `class2table` to be applicable, the corresponding guard predicate `guard_class2table` checks that the class being transformed is persistent. Finally, the corresponding value predicate `value_class2table` states that after transforming the class to a table, the name of the table should be the same as the name of the class.

3.2. *Extending F-Alloy to specify compound transformations*

F-Alloy as defined in [10] (and sketched above) only supports the specification of basic transformations, namely the input and output are both metamodels (expressed by Alloy modules). However, compound transformations, whose input and/or output modules represent themselves transformations (expressed by F-modules) are useful in various model driven software engineering activities. For example, in addition to enabling reuse of existing model transformation specifications, compound model transformations constitute the cornerstone of a new approach to model transformation validation (see Sect. 6.) In this paper, we extend F-Alloy to also support compound transformations.

3.2.1. *Syntactical adaptation*

Fig. 2, from left hand side to right hand side, conceptually summarizes the syntactical change made to F-modules. This syntactical change is realized by modifying the **ImportWF** well-formedness constraint from the F-Alloy definition given in Sect. 3 (the two `qualName` should now refer to two distinct module (Alloy Module or F-module)).

F-modules and Alloy modules are now unified under a common supertype referred to as *modules*. In the following, we use a to range over plain Alloy modules (i.e., `.als` files), f to range over F-modules (i.e., `.fals` files) and m to range over both of them. Moreover, we abstract F-modules into the form of $f : m_1 \rightarrow m_2$, where m_1 is the input module and m_2 the output module. In this case we say that *f is an F-module from m_1 to m_2*.

We distinguish two kinds of F-modules:

Basic F-modules which are F-modules whose input and output are both Alloy modules, defining basic transformations;
Compound F-modules which are F-modules whose input and/or output are themselves F-modules, defining compound transformations.

3.2.2. *Semantical extension*

In [10], a translational semantics is given that maps a basic F-module $f : a_1 \rightarrow a_2$ to a plain Alloy module referred to as the *augmented module*. We denote this translation by $\mathcal{A}(^{\bullet})$. The translation amounts to systematically adding constraints following the templates identified in [10]. We illustrate the translation by providing, in Listing 5, the augmented module corresponding to the F-module given in Listing 4.

In this example, we see that the upper part of the augmented module (Listing 5) is exactly the same as the basic F-module (Listing 4). The lower part of the augmented module aggregates the additional constraints organized into four facts. To illustrate the usefulness of these facts, let us consider the last fact named `MinimumOutput`. This fact enforces that any RDBMS elements present in an instance obtained by analyzing the augmented module should be mapped from a class following the `class2table` mapping. Without this fact, the Alloy analyzer, when analyzing the augmented module, would be free to add any extra RDBMS elements.

We extend the translational semantics in this paper to also accommodate compound F-modules. To avoid ambiguity, from now on, we refer to the translational semantics as defined in [10] (denoted $\mathcal{A}(\bullet)$) as the *basic translation*.

Definition 1 *(Extended translational semantics).* Let $[\bullet]_{\mathcal{A}}$ denote the extended semantics. Given a module m as defined in Fig. 2, right, $[m]_{\mathcal{A}}$ stands for the corresponding augmented module of m. Depending on whether m is an Alloy module or an F-module, basic and compound, we define:

$$[f : m_1 \rightarrow m_2]_{\mathcal{A}} \overset{\text{def}}{=} \mathcal{A}(f : [m_1]_{\mathcal{A}} \rightarrow [m_2]_{\mathcal{A}})$$

$$[a]_{\mathcal{A}} \overset{\text{def}}{=} a$$

```
module Class2Table

open CD
open RDBMS

one sig CREATE{
   class2table: Class  ->  Table,
}

pred guard_class2table(c:Class){
   c.is_persistent=True
}
pred value_class2table(c:Class ,  t:Table){
   t.name[0]=c.name
}
// ================== AUGMENTED MODULE CONSTRAINTS ===================
// provide an upper bound to fields which are not expressively assigned.
fact MinimalAssignment{
  all t:Table| t in CREATE.class2table[Class] implies {
    #t.name= 1
    t.cols= none
    t.pkeys= none
    t.fkeys= none
  }
}
// enforce the functions denoted by mappings to be injective.
fact MapInjectiveness{
   all t:Table | one CREATE.class2table.t
}
//enforces the guard and value predicates to act as pre and post conditions
fact PredicateAssociation{
   all x : Class {
      (guard_class2table[x] and one CREATE.class2table[x] and value_class2table[x, CREATE.class2table[x]])
      or
      (not guard_class2table[x] and no CREATE.class2table[x])
   }
}
//limits RDBMS elements to be those created through mapping
fact MinimumOutput{
   RDBMSElement = CREATE.class2table[Class]
}
```

Listing 5: Augmented module for the F-module given in Listing 4.

The extended translational semantics makes use of the basic one as defined in [10]. Briefly, in case m is a basic F-module, the basic translation ($\mathcal{A}(\bullet)$) is applied. In case m is a compound F-module, the imported modules need to be translated recursively (by applying $[\bullet]_{\mathcal{A}}$) prior to applying the basic translation.

Based on the extended translational semantics, we define in the following instances of a module m, being either an Alloy module or an F-module.

Definition 2 *(Module instances).* Given a module m, an instance x conforms to m if and only if x is a valid instance of $[m]_{\mathcal{A}}$. We then call x an m-instance.

4. Hybrid analysis of (compound) model transformations

In the previous section, we have introduced the extended semantics of F-Alloy through the definition of the translation function $[\bullet]_{\mathcal{A}}$. With this translation yielding an Alloy module (i.e., the augmented module), it is possible to derive the set of instances conforming to an F-module for validation purposes. The straightforward, but time consuming, way of obtaining this set of instances consists in first translating the F-module to the corresponding augmented module, then performing Alloy analysis on the latter.

In this section, we propose a more efficient approach to this problem of instance generation. Instead of pure analysis based instance searching, we propose a hybrid strategy that combines the analysis of Alloy with the interpretation of F-Alloy. We hence first introduce how F-Alloy specification can be interpreted before introducing in detail how our hybrid approach to the problem of instance generation is carried.

```
1  INPUT: -F-module $f$ from $m_1$ to $m_2$
2         -Instance $x_1$ of $a_1$
3  OUTPUT:-Instance $x$ of $f$
4  FUNCTION Interpretation(F-module $f$, Instance $x_1$)
5    INITIALIZE instance $x$ to contain $x_1$ + CREATE atom
6    FOR EACH mapping IN $f$ DO
7      ADD in $x$ new atoms for each tuples satisfying the guard + traceability links
8    DONE
9    FOR EACH mapping map IN $f$ DO
10     FOR EACH output atom created
11       ADD in $x$ tuples corresponding to fields of that atom as defined in value pred
12     DONE
13   DONE
14   IF $x$ conforms to $[f]_{\mathcal{A}}$ THEN
15     RETURN $x$
16   ELSE
17     RETURN NONE
18 END FUNCTION
```

Listing 6: Overview of the F-Alloy interpretation function $\mathcal{I}(f, x_1)$.

4.1. F-Alloy interpretation

We previously introduced F-Alloy as a domain specific language specifically designed to express model transformations. It has been shown in [10] that the syntax of F-Alloy has the interesting property of being interpretable. More specifically, considering a basic F-module $f : a_1 \to a_2$ and an a_1-instance x_1, it is possible to obtain an instance x conforming to f by *interpretation*.

In this work, we adapt the F-Alloy interpretation given in [10] to also support compound F-module. An overview of the adapted F-Alloy interpretation is provided in the form of pseudo code in Listing 6.

Compared to the version given in [10], the following two changes are introduced:

1. Line 1: the input F-module can now be a basic or a compound F-module (indicated by $f : m_1 \to m_2$);
2. Line 14: the output instance x is as before required to conform to the augmented module of f. However, we recall that we have introduced an extension to the translational semantics (cf. Sect. 3) to obtain augmented modules.

Briefly, the interpretation is carried out in two rounds:

- The first round (Lines 6–8) processes mappings. For each element present in x_1 satisfying the guard of a mapping, an element typed by the range of the mapping, as well as traceability links, are added to x.
- The second round (Lines 9–13) processes value predicates. Each element previously created has its fields assigned following the interpretable expressions (`iexpr`) declared in the associated value predicate.

As can be observed from the pseudo code, the interpretation – making use of mappings, guard and value predicates declared in f solely – behaves independently of the types of modules[1] that are imported by f. As a consequence, interpretation unifiedly works for both basic and compound F-modules, hence the introduction of the following notation:

Notation 2 *(Interpretation).* We denote the set of instances obtained by interpretation of a basic or compound F-module $f : m_1 \to m_2$, given an m_1-instance x_1, by $\mathcal{I}(f, x_1)$.

We note that, interpretation returning at most one instance given parameters, the set $\mathcal{I}(f, x_1)$ is composed of at most one instance. The correctness of interpretation with respect to the translational semantics of F-Alloy follows from the following two lemmas: We recall the notation $\mathcal{Z}(m, s)$ (Notation 1) denoting the set of instance obtained from alloy analysis of m in a scope s.

Lemma 1 *(Interpretation soundness). Let $f : m_1 \to m_2$ be an F-module and x_1 be a valid instance of m_1. $\mathcal{I}(f, x_1)$ is either empty or a valid instance of $[f]_{\mathcal{A}}$. Formally: for all natural number s, there exists a natural number s', such that,*

$$\forall x_1 \in \mathcal{Z}([m_1]_{\mathcal{A}}, s), \mathcal{I}(f, x_1) \subseteq \mathcal{Z}([f]_{\mathcal{A}}, s')$$

Proof. This lemma holds by construction for both basic and compound F-module thanks to the conformance check performed in Line 14 of the interpretation procedure given in Listing 6. □

The next lemma needs the following definition:

[1] Alloy module or basic or compound F-module.

Definition 3 *(Instance projection).* The projection of an instance x onto a module m is the m-instance composed of the atoms and tuples in x and typed by signatures and fields of m, respectively. We denote the projection of an instance x onto m by $x \Downarrow m$.

Corollary 1 *(Instance projection property). Given an F-module $m : m_1 \to m_2$, a scope s and an m-instance $x \in \mathcal{Z}([m]_{\mathcal{A}}, s)$, the following property holds:*

$$x \Downarrow m_1 \in \mathcal{Z}([m_1]_{\mathcal{A}}, s)$$

Proof. This corollary follows from the fact that constraints of m_1 should also hold in instances of m. Any instance x obtained from analysis of m thus contains a valid m_1-"sub-instance" that can be obtained using instance projection. □

We then have:

Lemma 2 *(Interpretation completeness). Let $f : m_1 \to m_2$ be an F-module. Any instance obtained by analysis of $[f]_{\mathcal{A}}$ given a scope s can be constructed by interpretation of f given its projection on m_1.*

Formally: for all natural number s, we have,

$$\forall x \in \mathcal{Z}([f]_{\mathcal{A}}, s), \mathcal{I}(f, x \Downarrow m_1) = \{x\}$$

Proof. This lemma has been proven by construction in [10] for basic F-modules (where m_1 and m_2 are Alloy modules) by showing that the interpretation follows the constraints added during translating basic F-modules to Alloy modules. Because the interpretation is independent to the types of modules that are imported by an F-module, as demonstrated in the pseudo code, this lemma also extends to compound F-modules. □

4.2. Hybrid analysis

We recall that the goal of this section is to define an efficient way to obtain instances of F-modules for validation purposes. We achieve this goal by proposing a hybrid strategy combining Alloy analysis with the previously introduced F-Alloy interpretation. We call this new hybrid approach *hybrid analysis* and define it as follows:

Definition 4 *(Hybrid analysis).* We denote the set of instances obtained by hybrid analysis of an Alloy module or of a (basic or compound) F-module within a scope s by the mathematical function $\mathcal{H}(m, s)$, defined as follows:

$$\mathcal{H}(m, s) = \begin{cases} \mathcal{Z}(m, s) & \text{if } m \text{ is an Alloy module} \\ \bigcup\limits_{x_1 \in \mathcal{H}(m_1, s)} \mathcal{I}(m, x_1) & \text{if } m \text{ is an F-module} \end{cases}$$

Briefly, given a module m, if m is an F-module from m_1 to m_2, the set of m-instances is obtained by first applying the hybrid analysis on m_1, then applying the F-Alloy interpretation on m for each instance of m_1 obtained previously. Otherwise, if m is a plain Alloy module, the set of m-instances is obtained by applying Alloy's analysis to m.

We demonstrate the correctness of the hybrid analysis strategy by the following theorem, which basically states that the result obtained by applying the hybrid analysis is equivalent to the result obtained by applying Alloy's analysis.

Theorem 1 *(Correctness of hybrid analysis). Given a module m and its corresponding augmented module $[m]_{\mathcal{A}}$, for all natural number s, there exists another natural number s', such that $\mathcal{H}(m, s) = \mathcal{Z}([m]_{\mathcal{A}}, s')$.*

Proof. By Definition 4 the hybrid analysis of an F-module is a recursive process, i.e., in case of an hybrid analysis of a compound F-module from m_1 to m_2 an hybrid analysis of m_1 is performed first. Knowing that the import hierarchy is acyclic (cf. **ImportWF** in Sect. 3), we prove the correctness of the above theorem by structural induction.

- If m is a plain Alloy module, then:
  1. $\mathcal{H}(m, s) = \mathcal{Z}(m, s)$ (Definition 4)
  2. $m = [m]_{\mathcal{A}}$ (Definition 1)
  3. $\mathcal{H}(m, s) = \mathcal{Z}([m]_{\mathcal{A}}, s)$ (1. and 2.)
  4. the theorem holds (3.)
- If m is a basic F-module from a_1 to a_2, then:
  1. $\mathcal{H}(m, s) = \bigcup\limits_{x_1 \in \mathcal{H}(a_1, s)} \mathcal{I}(m, x_1)$ (Definition 4)
  2. $\mathcal{H}(a_1, s) = \mathcal{Z}(a_1, s)$ (Definition 4)

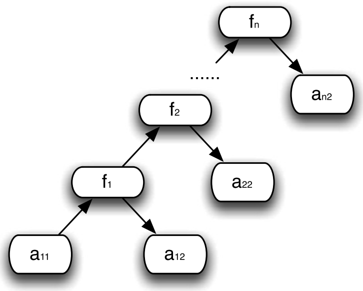


Fig. 3. Import hierarchy of an F-module.

3. $\mathcal{H}(m,s) = \bigcup_{x_1 \in \mathcal{Z}(a_1,s)} \mathcal{I}(m,x_1)$ (1. and 2.)
4. $\forall x_1 \in \mathcal{Z}(a_1,s), \mathcal{I}(m,x_1) \subseteq \mathcal{Z}([m]_{\mathcal{A}},s')$ (Lemma 1)
5. $\mathcal{H}(m,s) \subseteq \mathcal{Z}([m]_{\mathcal{A}},s')$ (3. and 4.)
6. $\forall x \in \mathcal{Z}([m]_{\mathcal{A}},s'), x = \mathcal{I}(m, x \Downarrow m_1)$ (Lemma 2)
7. $\mathcal{Z}([m]_{\mathcal{A}},s') \subseteq \left(\bigcup_{x \in \mathcal{Z}([m]_{\mathcal{A}},s')} \mathcal{I}(m, x \Downarrow m_1) \right)$ (6.)
8. $\left(\bigcup_{x \in \mathcal{Z}([m]_{\mathcal{A}},s')} \{x \Downarrow m_1\} \right) \subseteq \mathcal{Z}(a_1,s')$ (Corollary 1)
9. $\left(\bigcup_{x \in \mathcal{Z}([m]_{\mathcal{A}},s')} \{x \Downarrow m_1\} \right) \subseteq \left(\bigcup_{x_1 \in \mathcal{Z}(a_1,s')} \{x_1\} \right)$ (8.)
10. $\left(\bigcup_{x \in \mathcal{Z}([m]_{\mathcal{A}},s')} \mathcal{I}(m, x \Downarrow m_1) \right) \subseteq \left(\bigcup_{x_1 \in \mathcal{Z}(a_1,s')} \mathcal{I}(m,x_1) \right)$ (9.)
11. $\mathcal{Z}([m]_{\mathcal{A}},s') \subseteq \left(\bigcup_{x_1 \in \mathcal{Z}(a_1,s')} \mathcal{I}(m,x_1) \right)$ (7. and 10.)
12. $\mathcal{Z}([m]_{\mathcal{A}},s') \subseteq \mathcal{H}(m,s')$ (3. and 11.)
13. $\mathcal{H}(m,s) = \mathcal{Z}([m]_{\mathcal{A}},s')$ (5 and 12.)
14. the theorem holds

- If m is a compound F-module from m_1 to m_2, then:

1. $\mathcal{H}(m,s) = \bigcup_{x_1 \in \mathcal{H}(m_1,s)} \mathcal{I}(m,x_1)$ (Definition 4)
2. $\mathcal{H}(m_1,s) = \mathcal{Z}([m_1]_{\mathcal{A}},s')$ (induction hypothesis)
3. $\mathcal{H}(m,s) = \bigcup_{x_1 \in \mathcal{Z}([m_1]_{\mathcal{A}},s')} \mathcal{I}(m,x_1)$ (1. and 2.)
4. let $a_1 = [m_1]_{\mathcal{A}}$ ($[m_1]_{\mathcal{A}}$ being an Alloy module).
5. $\mathcal{H}(m,s) = \bigcup_{x_1 \in \mathcal{Z}(a_1,s')} \mathcal{I}(m,x_1)$ (3. and 4.)
6. The equation in 5. is the same than the one given in point 3. of the reasoning for basic F-module. Hence, following the same reasoning than for basic F-modules, the theorem holds for compound F-module.

We have proven, case by case, that the hybrid analysis of any module, independently of its nature, is equivalent to the Alloy analysis of its corresponding augmented module (in the sense that it produces, for given scopes, the same set of instances). □

Compared to pure Alloy based analysis, hybrid analysis substantially reduces computation complexity. Let f_n be an F-module with its transformation hierarchy depicted in Fig. 3. Instead of analyzing $[f_n]_{\mathcal{A}}$ using the Alloy Analyzer, which is very resource consuming especially in case of big scopes, one only needs to analyze the left-most leaf Alloy module a_{11} and all the rest can be built by applying the F-Alloy interpretation to $f_1, \ldots, f_n$, each taking polynomial time [10]. In addition, the problem of finding an optimal scope for a given analysis is also reduced accordingly: one only needs to define a scope for a_{11} instead of f_n which is much more complex. More specifically, the time complexity of hybrid analysis is given below.

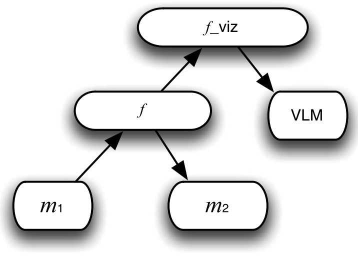


Fig. 4. Import hierarchy of f_viz.

Proposition 1 *(Hybrid analysis complexity). The time needed for hybrid analysis of an F-module f from m_1 to m_2 is a polynomial function of the time needed for the hybrid analysis of m_1.*

Proof. Let $t(m_1)$ denote the time needed for the hybrid analysis of m_1 and let $t(f)$ denote the time needed for the hybrid analysis of f. From Definition 4 it follows that:

$$t(f) \le c \cdot \sum_{x_1 \in \mathcal{H}(m_1,s)} t_{\mathcal{I}}(m_1, x_1)$$

where $t_{\mathcal{I}}(m_1, x_1)$ is the time for the interpretation of f on input x_1 and c is a constant. By [10], and the fact that interpretation is independent of the nature of the imported modules, there exists a polynomial p such that $t_{\mathcal{I}}(m_1, x_1) \le p(|x_1|)$ where $|x_1|$ is the size of instance x_1. The instance x_1 being obtained from hybrid analysis of m_1, it follows that $|x_1| \le t(m_1)$. Therefore:

$$t(f) \le c \cdot \sum_{x_1 \in \mathcal{H}(m_1,s)} p(t(m_1))$$

Since $|\mathcal{H}(m_1, s)| \le t(m_1)$, we conclude that:

$$t(f) \le c \cdot t(m_1) \cdot p(t(m_1))$$

thus implying the proposition. □

To summarize, we enumerate the advantages of using F-Alloy for expressing model transformations over plain Alloy:

1. F-Alloy allows to specify transformations more concisely by relieving the necessity of articulating a number of implicit constraints. The contrast between Listing 4 and Listing 5 can be seen as an evidence.
2. Transformations written in F-Alloy can be validated more efficiently using hybrid analysis. This has already been discussed above, and will be further demonstrated with empirical data in Sect. 7.

5. Visualization-based validation of model transformation

In the previous section, we have introduced hybrid analysis as a way to efficiently obtain the set of all instances conforming to a given F-Alloy specification. The fact that hybrid analysis can be applied to compound transformations of any depth paves a new way to the validation of F-Alloy transformations, namely Visualization-Based Validation (VBV), complementing the other existing model transformation validation techniques [18].

Given a transformation f specified in F-Alloy, the VBV of f relies on the definition of a compound transformation called f_viz from f to VLM as depicted in Fig. 4, to give a domain specific visualization to f-instances.

VLM, standing for Visual Language Model, is an Alloy module defining a set of graphical concepts such as shapes (e.g., `RECTANGLE,ELLIPSE`), colors, layouts, and arrows/lines allowing to connect shapes (i.e., `CONNECTOR`). It is accompanied by a tool, Lightning,[2] that can parse and graphically render VLM instances. We note that in the following, given a module m (specified in Alloy or F-Alloy), the notation m_viz denotes a transformation from m to VLM that provides a domain specific visualization to m-instances.

5.1. The VBV process

In Fig. 5, we provide an iterative process prescribing an effective application of VBV to a model transformation $f : m_1 \rightarrow m_2$. We note that two actors are part of this process:

[2] Lightning is available for download at: http://lightning.gforge.uni.lu.

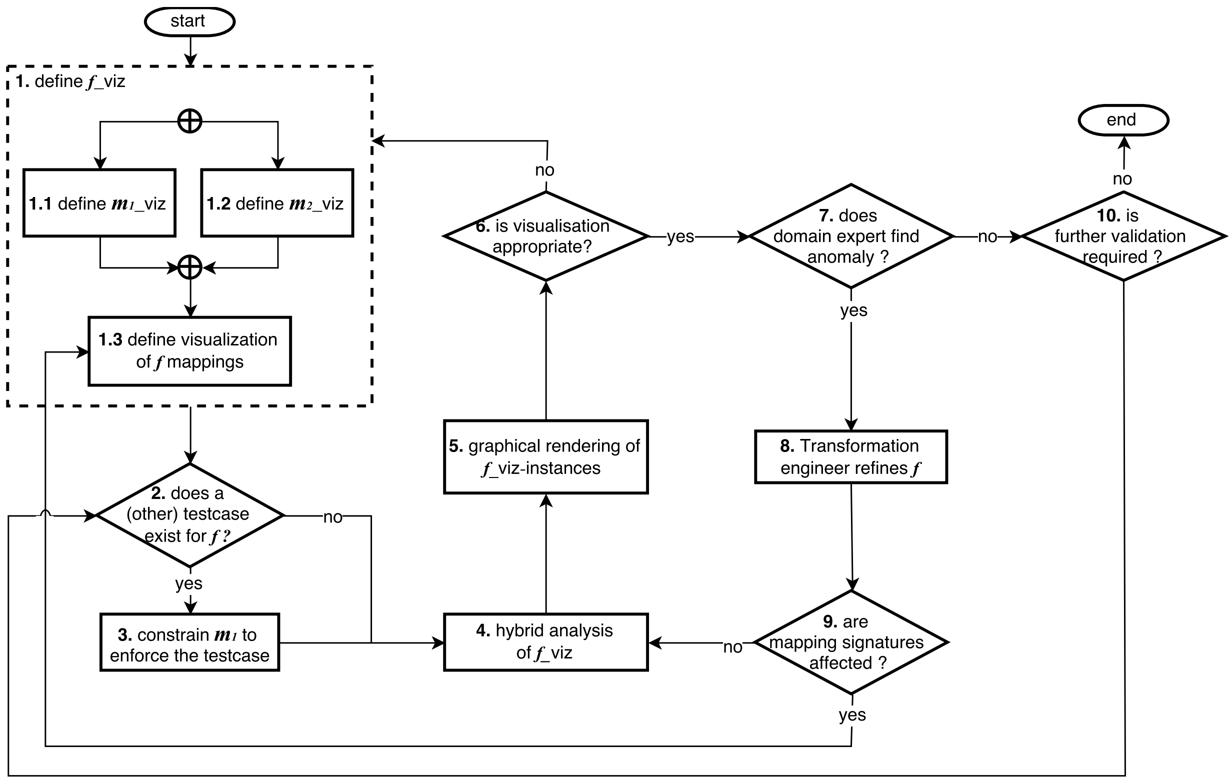

start
1. define f_viz
1.1 define m1_viz
1.2 define m2_viz
1.3 define visualization of f mappings
2. does a (other) testcase exist for f ?
yes
no
3. constrain m1 to enforce the testcase
4. hybrid analysis of f_viz
5. graphical rendering of f_viz-instances
6. is visualisation appropriate?
no
yes
7. does domain expert find anomaly ?
yes
no
8. Transformation engineer refines f
9. are mapping signatures affected ?
no
yes
10. is further validation required ?
no
end

Fig. 5. Visualization-Based Validation (VBV) of the model transformation $f : m_1 \rightarrow m_2$.

- The *Domain Expert* is acquainted with the input and output of the transformation to be validated and knows which output should the transformation yield for a given input.
- The *Transformation Engineer* is an expert in Alloy and F-Alloy and implements the transformation f following his/her understanding of the transformation as conveyed by the domain expert.

The first step of this process is to define the f_viz transformation. It is composed of a set of mappings defining (1) the domain specific visualization of m_1 and m_2, and (2) the visualization of applications of mappings declared in f – e.g., *traces* relating m_1 elements to m_2 elements.

For (1), the domain expert provides instructions on how m_1 and m_2 instances should be represented to the transformation engineer. The transformation engineer then specifies the transformations m_1_viz (step 1.1) and m_2_viz (step 1.2). Note that those transformations might already exist under certain circumstances (e.g., a transformation from (or to) m_1 (or m_2) has already been validated using VBV).

For (2), we propose a default visualization of traces: dashed arrows with distinct colors linking elements of m_1 to the elements of m_2. This default visualization can be automatically defined by the following rules:

- For each mapping $\mu_i : X \rightarrow Y$ of f, a mapping $\nu_i : X \rightarrow Y \rightarrow$ CONNECTOR is added to the f_viz transformation.
- The guard predicate associated to mapping ν_i specifies that only pairs of elements of X and Y that are part of mapping μ_i yield a new CONNECTOR.

```
pred guard_νi(x:X,y:Y){
  CREATE->x->y in μi
}
```

- The value predicate associated to mapping ν_i specifies that each created CONNECTOR has as source the graphical representation of x as defined in m_1_viz and as target the graphical representation of y as defined in m_2_viz. Moreover the CONNECTOR is labeled and colored accordingly to indicate that it represents a trace of μ_i.

```
/*  mapping XtoViz (resp. YtoViz) is defined in $m_1$_viz (resp. $m_2$_viz)
and provide a graphical representation of x (resp. y) */
pred value_$v_i$(x:X,y:Y,c:CONNECTOR){
  c.source = CREATE.XtoViz[x]
  c.target = CREATE.YtoViz[y]
  c.color = RED
  c.label = "$\mu_i$"
}
```

Once f_viz is defined, one has to decide how VBV will be performed (step 2). VBV can be performed either (1) on concrete testcases, or (2) on random instances within a finite scope.

In case of (1), the transformation engineer adds constraints to m_1 so that the only possible m_1-instances satisfying those constraints are the ones described in the testcases (step 3). Those constraints can be systematically generated given the testcases to enforce (see an example given in Listing 8).

In case of (2), the transformation engineer provides a scope to elements of m_1 (or the left-most leaf Alloy module in the import hierarchy of m_1 in case m_1 is also an F-module). Note that the effectiveness of this case is based on the small scope hypothesis claiming that most of design errors can be reproduced in small instances.

Once the m_1 module has been constrained as per step 3, a hybrid analysis is performed on f_viz (step 4), resulting in the production of the set of f_viz-instances whose m_1-sub-instances conform to those constraints.

VLM-sub-instances present in the obtained f_viz-instances are then rendered accordingly by the Lightning tool (step 5) to produce a visualization similar to those depicted in Figs. 8–11.

The visualization of those instances is then shown to the domain expert for validation. If it hinders the comprehension of a given instance, the domain expert can ask f_viz to be refined (step 6). If on the contrary the visualization is sufficiently intuitive, the domain expert carries on his review. As soon as an anomaly is found (step 7), it is communicated to the transformation engineer who refines f accordingly (step 8). We note that the visualization of traces can also help the transformation engineer in determining which mapping is faulty. Any structural changes brought during the refinement of f (i.e., changes in the mapping names and types) should be propagated to f_viz (step 9). Hybrid analysis can then once again be applied on f_viz for the domain expert to validate the changes brought to f by the transformation engineer (back to step 4). If no anomalies are found after reviewing the newly produced instances, the domain expert can request (step 10) another iteration of VBV (on other testcases or in different scope) or accept the transformation as valid (if a certain degree of confidence has been reached).

In the next section, we show how this process can be applied to validate the CD2RDBMS transformation.

6. Application of VBV to CD2RDBMS

In this section, we exemplify the process introduced previously. We show how we validate an F-Alloy implementation of the benchmark CD2RDBMS transformation [5], using VBV.

6.1. CD2RDBMS *specification and first F-Alloy implementation*

The F-module produced by the transformation engineer implements the CD2RDBMS transformation in terms of four mappings. For the sake of conciseness, we focus on the validation of the most error-prone mapping, namely `association2column` that defines how associations are to be transformed. We extract here the statements that are relevant for associations from the original CD2RDBMS specifications given in [5].

Rule 1 "... The resultant table (obtained from a persistent class) should contain ... one or more columns for every association for which the class is marked as being the source ..."

Rule 2 "... for each association whose `dst` is such a class (i.e., for each association whose destination class is non-persistent), each of the classes attributes should be transformed as per rule 3. The columns should be named `name_transformed_attr` where `name` is the name of the association in question, and `transformed_attr` is a transformed attribute ... The columns will be placed in tables created from persistent classes."

Rule 3 "Attributes whose type is a primitive data type (e.g. String, Int) should be transformed to a single column whose type is the same as the primitive data type."

Rule 4 "Attributes whose type is a persistent class (resp., association whose `dst` is a persistent class) should be transformed to one or more columns, which should be created from the persistent classes' primary key attributes. The columns should be named `name_transformed_attr` where `name` is the attributes' name. The resultant columns should be marked as constituting a foreign key, the `FKey` element created should refer to the table created from the persistent class. (The columns will be placed in tables created from persistent classes.)"

Rule 5 "Attributes whose type is a non-persistent class (resp., association whose `dst` is a non-persistent class) should be transformed to one or more columns as per rule 2. Note that the primary keys and foreign keys of the translated non-persistent class need to be merged in appropriately, taking into consideration that the translated non-persistent class may contain primary and foreign keys from an arbitrary number of other translated classes."

```
1  module CD2RDBMS
2
3  open CD
4  open RDBMS
5
6  one sig CREATE{
7     class2table: Class  ->  Table,
8     attribute2column: Attribute  ->  Column,
9     association2column: Association  ->  Attribute  ->  Column,
10    association2FKey: Association  ->  FKey
11  }
12
13 pred guard_association2column(ass:Association,att:Attribute){
14    ass.src.is_persistent = True and att.is_primary = True and att.~attrs in ass.dest
15 }
16 pred value_association2column(ass:Association  , att:Attribute, c:Column){
17    c.dataType = (att.type = STRING implies Text else Number)
18    c.label[0] = ass.name
19    all i:Int| i >= 0 and i<= sub[#(ass.dest.^(~src.dest)),1] implies c.label[add[i,1]] = ( c.label[i].~name.
          dest != att.~attrs and c.label[i] != none implies c.label[i].~name.dest.~src.name else none)
20    c.label[#(ass.dest.*(~src.dest))] = att.name
21    c in CREATE.class2table[ass.src].cols
22    ass.dest.is_persistent = False implies c in CREATE.class2table[ass.src].pkey
23    att.~attrs.is_persistent = False or att.is_primary = True implies c in CREATE.association2FKey[ass].
          columns
24 }
```

Listing 7: Extract of the CD2RDBMS transformation to be validated using VBV.

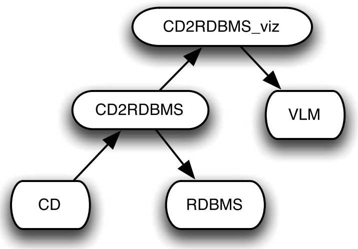


Fig. 6. Import hierarchy of CD2RDBMS_viz.

Listing 7 shows the first attempt of the transformation engineer in implementing the `association2column` mapping, namely prior to VBV.

The `guard_association2column` predicate enforces:

Line 14: for each association whose `src` class is persistent (as per Rule 1) and for each primary attribute `att` contained in its `dst` class the creation of a column (according to Rule 4, resp. Rule 5, if the `dst` class is persistent, resp. non-persistent).

The `value_association2column` predicate enforces:

Line 17: the type of the created column to be equivalent to the type of `att` (Rule 3).
Line 18–20: the name of the created column to follow Rules 2 and 4.
Line 21: the placement of the created column in the table representing the `src` class of `ass` (Rules 2 and 4).
Line 22: the created column to become a primary key of its table if the `dst` class is non-persistent (Rule 5).
Line 23: the created column to be part of a foreign key that refers to the table representing `dst` class if `dst` class is persistent and `att` is primary (Rule 4).

With the CD2RDBMS transformation defined, we now show how to validate it using VBV.

6.2. *Definition of* CD2RDBMS_viz

According to the process given in Fig. 5, when applying VBV to CD2RDBMS the first step is the definition of a visualization for CD2RDBMS. This visualization is expressed, as seen earlier in Sect. 5, by a compound F-module called CD2RDBMS_viz (depicted in Fig. 6) from CD2RDBMS to an Alloy module called VLM (Visual Language Model) that defines the set of graphical concepts recognized by the supporting tool (i.e., Lightning). To this end, a set of mappings defining the visualization for CD and RDBMS is first implemented (in CD_viz and RDBMS_viz, respectively) by the transformation engineer following the

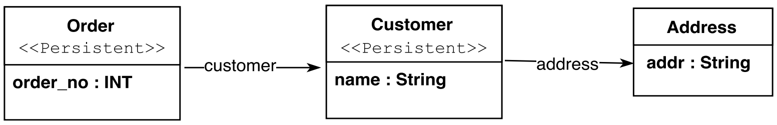


Fig. 7. Example of Class Diagram that can be given as input to the CD2RDBMS Transformation.

domain experts guidance. The set of mappings defining the default visualization of traces (as described in Sect. 5) is then added to obtain the final CD2RDBMS_viz F-module.

We provide the CD_viz F-module below to illustrate how such a visualization is defined.

```
module CD_viz

open CD
open VLM

one sig CREATE{
    class2Rect: Class -> RECTANGLE,
    class2Txt: Class -> TEXT,
    attr2Txt: Attribute -> TEXT,
    ass2Connector: Association -> CONNECTOR,
}

pred guard_class2Rect(c:Class){}
pred value_class2Rect(c:Class,r:RECTANGLE){
    r.layout=VERTICAL_LAYOUT
    r.composedOf[0]=CREATE.class2Txt[c]
}

pred guard_class2Txt(c:Class){}
pred value_class2Txt(c:Class,t:TEXT){
    t.textLabel[0]=c.name
    t.textLabel[1]=(c.is_persistent=False implies
          none else "<<Persistent>>")
    t.color=BLACK
    t.isBold=True
}

pred guard_attr2Txt(a:Attribute){
    a.type in PrimitiveDataType
}
pred value_attr2Txt(a:Attribute,t:TEXT){
    t.textLabel[0]=a.name
    t.textLabel[1]=":"
    t.textLabel[2]=a.type
    t.color= (a.is_primary=False implies BLACK
         else RED)
    t in CREATE.class2Rect[a.~attrs].composedOf
         [1]
}
pred guard_ass2Connector(a:Association){}
pred value_ass2Connector(a: Association ,c:
     CONNECTOR){
    c.connectorLabel[0]=a.name
    c.source=CREATE.class2Rect[a.src]
    c.target=CREATE.class2Rect[a.dest]
    c.color=RED
}
```

The mapping `class2Rect` defines that each class is represented by a `RECTANGLE`. This `RECTANGLE` contains a `TEXT`, defined by the `class2Txt` mapping, carrying the name of the class it represents as well as a "persistent" stereotype if that class is persistent. Each attribute is represented by a `TEXT` in the `RECTANGLE` representing the class containing it, as defined by the `attr2Txt` mapping. This `TEXT` is of the form "`att:type`" where `att` is the name of attribute and `type` its type. Furthermore, `TEXT`s representing primary attributes are highlighted in red. Finally, the `ass2Connector` mapping defines that each association is represented by a `CONNECTOR` from the `RECTANGLE` representing its source class to the `RECTANGLE` representing its destination class.

6.3. *Example of VBV iteration using a specific testcase*

In the first iteration, we use the example provided in [5] as testcase. More specifically, the class diagram depicted in Fig. 7 is used as input to test the behavior of the CD2RDBMS transformation.

To enforce the use of this input, additional constraints are appended to the original CD module (introduced in Listing 1). Those constraints are given in Listing 8, where, e.g., one class (`Class1`) is constrained to be named `"Order"`, to be persistent, and to have a primary attribute named `"order_no"` of type `INT`.

After constraining the CD module an hybrid analysis on CD2RDBMS_viz is performed. Following Definition 4, the hybrid analysis of CD2RDBMS_viz starts by applying Alloy analysis on CD, yielding the CD-instance depicted in Fig. 7. It then performs an F-Alloy interpretation on CD2RDBMS given the CD-instance, yielding the corresponding CD2RDBMS-instance. Finally, it builds a CD2RDBMS_viz-instance from the interpretation of CD2RDBMS_viz given the CD2RDBMS-instance.

After hybrid analysis this CD2RDBMS_viz-instance is rendered graphically as shown in Fig. 8 and submitted to the domain expert for validation.

The domain expert finds that the transformation is not behaving properly: the columns in the foreign key `FKEY` do not cover all the primary keys of the `Customer` table. Communication between domain expert and transformation engineer leads to the conclusion that Rule 4 is ambiguous. More specifically, the use of the terms "primary key attribute" was unclear: the transformation engineer understood from Rule 4 that a column is to be created for each primary attribute of the persistent `dst` class while the domain expert meant that a column is to be created for each primary key in the table corresponding to the persistent `dst` class.

To resolve this error, the domain expert rephrases Rule 4 and the transformation engineer accordingly refines the F-Alloy guard predicate in line 14 as follows:

```
// ASSOCIATIONS
one sig Ass1 extends Association{}{
name="customer"
src=Class1
dest=Class2
}
one sig Ass2 extends Association{}{
name="address"
src=Class2
dest=Class3
}

// CLASSES
one sig Class1 extends Class{}{
name="Order"
is_persistent=True
attrs=Attr1
}
one sig Class2 extends Class{}{
name="Customer"
is_persistent=True
attrs=Attr2
}
one sig Class3 extends Class{}{
name="Address"
is_persistent=False
attrs=Attr3
}

// ATTRIBUTES
one sig Attr1 extends Attribute{}{
name="order_no"
is_primary=True
type=INT
}
one sig Attr2 extends Attribute{}{
name="name"
is_primary=True
type=STRING
}

one sig Attr3 extends Attribute{}{
name="addr"
is_primary=True
type=STRING
}
```

Listing 8: Alloy snippet appended to the CD module to enforce the testcase given in [5].

```
14    ass.src.is_persistent=True and    att.is_primary=True and
      (att in (ass.dest.*(~src.dest & Class ->False.~is_persistent)).attrs)
```

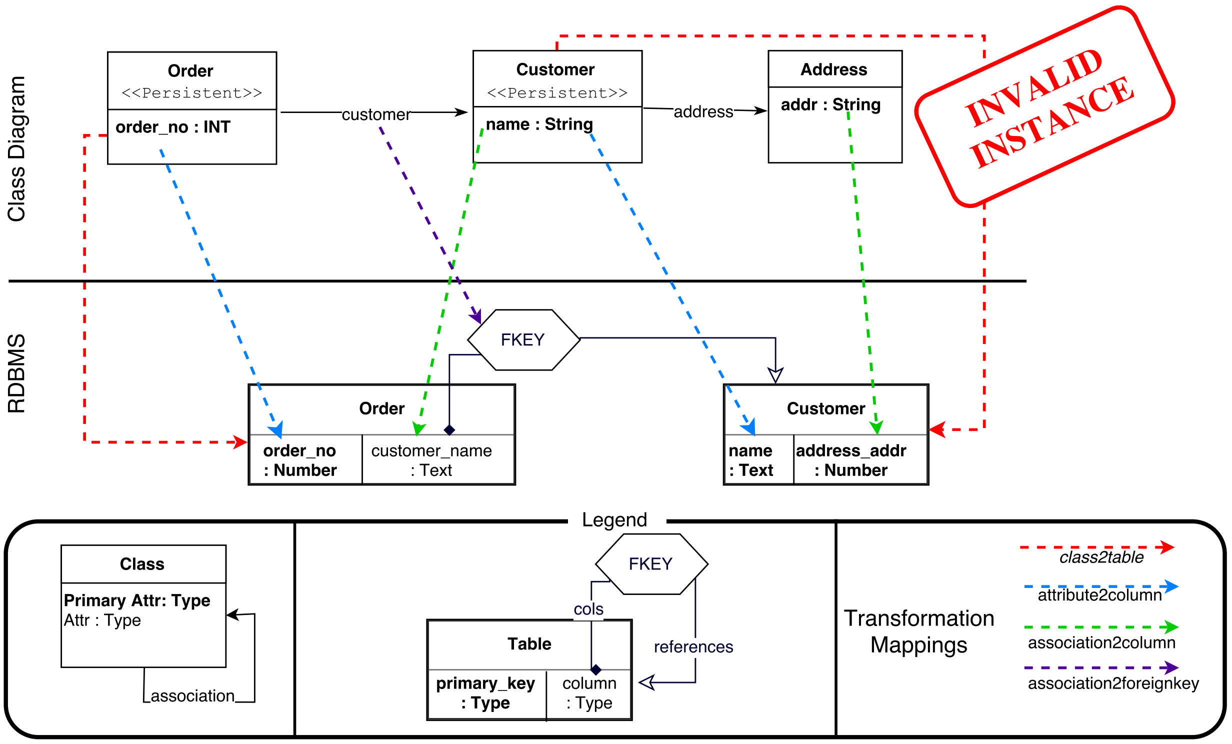


Fig. 8. Visualization of a bug in the computation of the CD2RDBMS transformation: FKey missing for primary key columns obtained from associations.

Columns are now not only created for each primary attribute of the dst class but also for all the other primary attributes which would lead to the creation of primary keys in the table corresponding to the dst class.

Hybrid analysis is performed again on CD2RDBMS_viz after this modification. This time, the correct instance is produced as shown in Fig. 9.

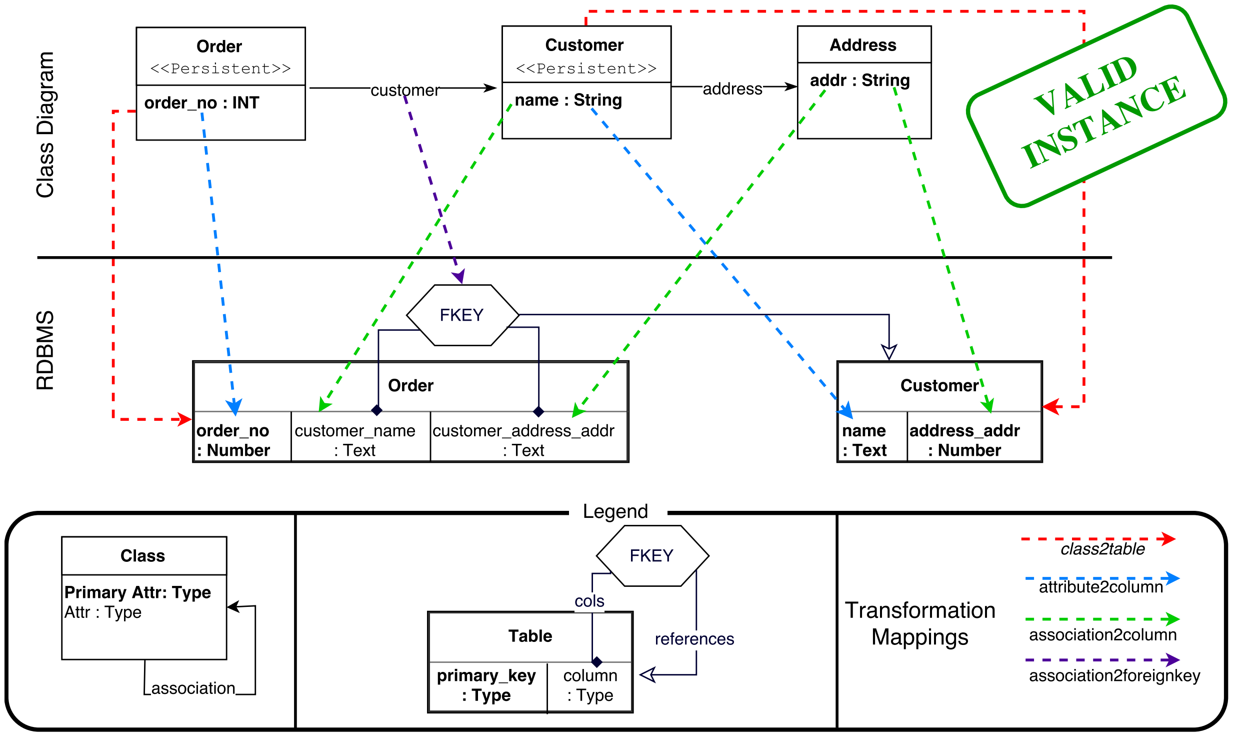


Fig. 9. Visualization of the case depicted in Fig. 8 after refinement of the CD2RDBMS transformation.

6.4. Example of VBV iteration using random instance generation

A second iteration of VBV is then performed. This time, the domain expert requests to add random CD elements to the previous testcase to see how the transformation would perform. The transformation engineer hence alters constraints in the CD module to fulfill this request.

A hybrid analysis of CD2RDBMS_viz is then performed and resulting instances are rendered and provided to the domain expert. In Fig. 10, we provide one of those instances containing an anomaly found by the domain expert, namely, there is no column corresponding to the `zip_code` attribute (emphasized in the visualization by the absence of connector originating from `zip_code`).

The lack of corresponding column for an attribute of a non-persistent `dst` class – `zip_code` is an attribute of the non-persistent class `Address` which is the destination of the association `address` – is a violation of Rule 2. After discussion, the transformation engineer realizes that he mistakenly applied Rule 4 to the case of non-persistent classes, ignoring completely the directives of Rule 2. To fix this error, the transformation engineer refines the guard predicate `guard_association2column`. This time, two distinct cases are implemented, to cater for the two rules, respectively. This refinement leads to the following modification of line 14:

```
14      (ass.dest.is_persistent=False implies  att in ass.dest.attrs)
        and
        (ass.dest.is_persistent=True implies
        (att in (ass.dest.*(~src.dest & Class ->False.~is_persistent)).attrs and att.is_primary=True))
```

This new version of the guard now check first whether the `dst` class of association `ass` is persistent or not:

- In case the association `dst` class is persistent, the behavior stays the same as in previous version: a column is created for each primary attribute which would lead to the creation of primary keys in the table corresponding to the `dst` class (Rule 4).
- In case the association `dst` class is not persistent, a column is now created for each attribute of the `dst` class (Rule 2).

Hybrid analysis is performed again on CD2RDBMS_viz after this modification yielding a set of instances which do not reproduce this error.

In this set, the instance containing the same class diagram as the one shown in Fig. 10 is shown in Fig. 11.

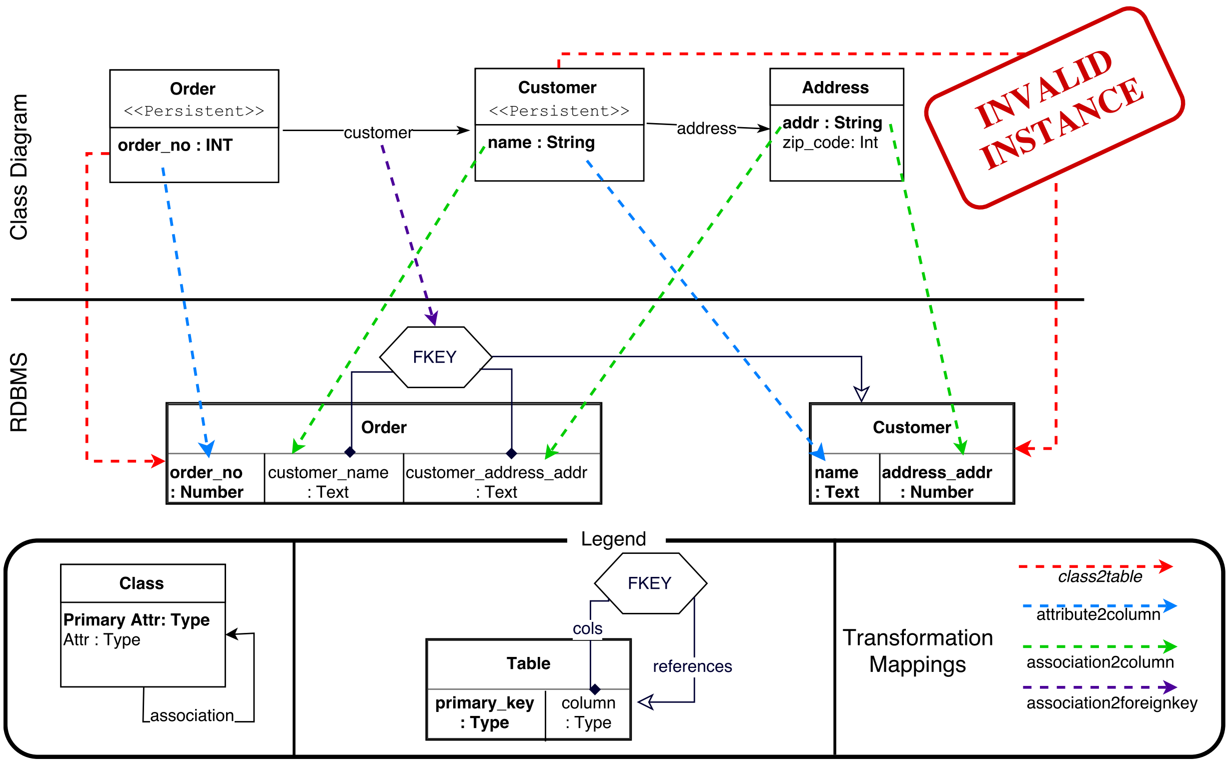


Fig. 10. Visualization of a bug in the computation of the CD2RDBMS transformation: non-primary attribute of non-persistent classes are lost in the transformation.

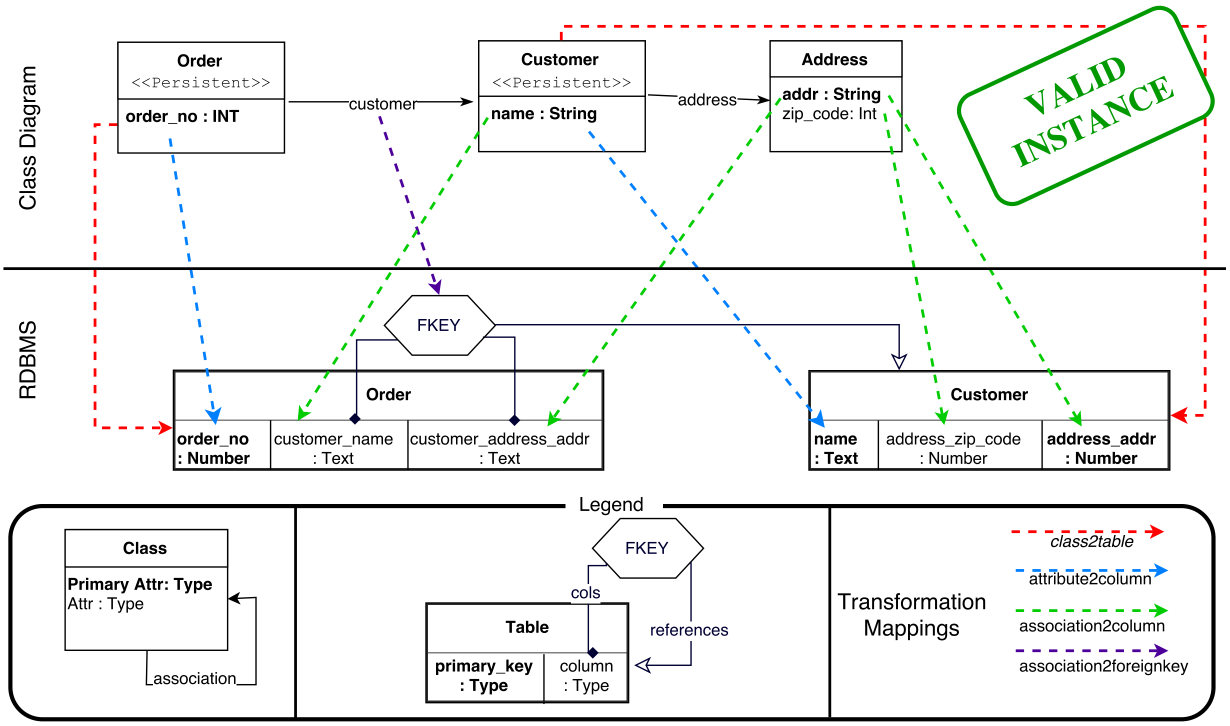


Fig. 11. Visualization of the case depicted in Fig. 10 after refinement of the CD2RDBMS transformation.

Table 1
Comparative table: time needed in milliseconds to find the first instance.

Alloy analysis scope	CD	CD2RDBMS		CD2RDBMS_viz	
	Alloy analysis	Alloy analysis	Hybrid analysis	Alloy analysis	Hybrid analysis
5	26	1720	95	3078	229
10	53	7251	181	15862	386
15	76	24671	173	∞	337
20	1844	∞	1918	∞	2053

7. Evaluation of hybrid analysis

Alloy analysis is commonly used throughout model driven software development processes for validating models, where instances of models are automatically generated and reviewed for potential errors. Such a scenario can often be found in case of agile software development where the designer incrementally improves his models after spotting errors in instances obtained by analysis [13].

In this section, we evaluate the efficiency of hybrid analysis by comparing its performance with Alloy analysis on two F-modules introduced in previous sections: the basic transformation from class diagrams to relational databases (i.e., the CD2RDBMS example), and the compound transformation CD2RDBMS_viz that defines a visualization for CD2RDBMS.

According to Definition 4, the hybrid analysis of module CD2RDBMS_viz performs as follows:

1. CD2RDBMS_viz being an F-module, hybrid analysis is performed on the CD2RDBMS module.
   (a) the CD2RDBMS module being an F-module, hybrid analysis is performed on the CD module.
   (b) the CD module being an Alloy module, CD-instances are generated using the Alloy analyzer.
   (c) For each instance obtained in step b, an interpretation of CD2RBDMS is performed – resulting in a set of CD2RDBMS-instances.
2. For each instance obtained in step c, an interpretation of CD2RDBMS_viz is performed – resulting in a set of CD2RDBMS_viz-instances.

Note that steps (a) to (c) also correspond to the hybrid analysis of CD2RDBMS.

Time measurements of Alloy and hybrid analysis applied to the CD2RDBMS and CD2RDBMS_viz modules, as well as the time needed for the Alloy analysis of the CD module can be found in Table 1 (times given in ms). Note that the symbol ∞ has been used to mark operations that failed to finish. All experiments were carried out on a computer with a Quad-Core Intel i7 CPU and 8 GB memory. Measurements show that the time needed to perform an Alloy analysis increases substantially with the scope and the size of the model. They also show that the time needed to find the first instance of module CD2RDBMS and of module CD2RDBMS_viz using hybrid analysis, is of the same order of magnitude as the time needed to find the first instance of the CD module, for a given scope. These observations provide us with positive support for Proposition 1.

Note that the hybrid analysis of CD2RDBMS and CD2RDBMS_viz surprisingly took less time to complete for a scope of 15 than for a scope of 10. This is due to the fact that the time needed for interpretation to complete is proportional to the size of the instances given to the interpreter and to the fact that the first instance obtained by the analysis of CD with a scope of 10 was bigger than the one obtained with a scope of 15.

8. Related work

Improve Alloy analysis performance: Our work takes its root in the general problem of speeding up the analysis of Alloy specifications. Different approaches have been proposed for this in the literature. To understand these approaches, we need to recall how the Alloy Analyzer works. The analysis is based on transforming the Alloy model into a propositional formula that is fed into an off-the-shelf SAT solver.

Alloy analysis can be optimized by improving the transformation from Alloy model to propositional formula. In [19] traditional compiler optimizations are used to improve this transformation, resulting in some cases in time reductions of an order of magnitude.

A different class of optimization techniques is based on applying slicing techniques to Alloy models. In [23] a sub model is identified, called a base slice, that is simpler and more efficiently solvable. Such a base slice can either be proven unsolvable, or extended in a systematic fashion into a full solution. This second approach is closer to our proposed approach: in our case the extension of a partial instance to a complete instance is done via interpretation, while in [23] it is done using a constrained analysis (guaranteeing that a solution for the whole model satisfies the constraints of the base slice).

Yet another type of approach, described in [14], proceeds by annotating Alloy models with meta-information that allows to use domain-specific solvers (e.g., String and Integer solvers) to solve sub-models and combine the output of those solvers with the SAT-based back-end of the Alloy Analyzer.

Rather than improving the speed of the analysis one can attempt to improve its applicability. More specifically, an approach based on SMT solvers allows to drop the finite scope assumption of Alloy's analyzer and actually prove properties of a model regardless of the scope [7]. The drawback of such an approach is the possible need for manual intervention.

Visualization based validation: In this work we proposed an approach to enable the systematic validation of F-modules through the domain specific visualization of a sample of conforming instances. The main incentive to support visualization based validation is to involve domain experts in the process of validation by providing them with material they are familiar with. Validation based on visual feedbacks is not something new, and can be encountered in various other domains. As an example, we have seen earlier that the Alloy language comes with a tool, the Alloy Analyzer, allowing the generation and visual rendering of instances as graphs (where atoms are nodes and links are edges). This tool and approach has been successfully applied in the validation of software systems [17,21], and generally contributed to close the gap between the engineer modeling the system in Alloy, and their client, the domain expert for which the system is designed [3]. However, it has been highlighted in [9] that despite recent advances in Alloy instances representations [25], intuitive visualization of large instances can only be achieved using the knowledge of their domain of application, namely domain specific visualization.

Following this line of reasoning, definitions of domain specific visualization for Alloy modules in terms of F-Alloy transformations from Alloy modules to VLM has been first introduced in [11]. In that work, those transformations enabled a language engineer and their client to seamlessly validate abstract syntax, concrete syntax and semantics of Domain Specific Languages (DSLs). Similarly but in a different application domain, [8] aimed to close the gap between Alloy experts and robotic engineers in the process of designing and validating robot perception systems [15].

In this work, we proposed to apply the domain specific visualization based validation approach to the engineering of model transformations. Compared to other domains reported in earlier work, this domain is inherently more complex, (because the validation targets are now transformations themselves, and we need compound transformations from them to VLM to enable the validation process), hence more error-prone to implement. We have seen in Sect. 6 that errors can not only come from transformation engineers e.g., when they failed to respect literally the transformation specifications written in natural language, but can also come from domain experts e.g., when they failed to be precise with the intention of the transformations. As a consequence, the involvement of domain experts in the validation process becomes even more necessary than beneficial. This work is, to our knowledge, the first that thoroughly goes in this direction for validating model transformations. In [24], a visualization techniques was proposed to visualize traceability links in (chains of) model transformations, to facilitate the tracking of the origin of errors in transformation chains. However, the absence of domain specific visualization of the source and target models prevent the effective involvement of domain experts in the loop. On the contrary, in our approach, the visualization of traces is not only a way to track the origin of errors in a transformation output, but also together with the domain specific visualization of both the source and target models, it offers additional means to the domain experts to validate the transformation themselves (as seen in Sect. 6).

9. Conclusion and future work

This paper presented an extension of F-Alloy, an Alloy based language for model transformations, for the specification, efficient computation and validation of (compound) transformations. In addition to leveraging the automatic instance generation facility of Alloy analysis for validation of model transformations, the hybrid analysis strategy proposed in this paper further speeds up the validation process, by combining Alloy analysis with F-Alloy interpretation. We evaluated the efficiency of our approach by applying it to two examples. According to the experiments, the complexity reduction is substantial, for both basic and compound transformations.

As a consequence of extending F-Alloy to compound cases, a new approach to validating model transformations is enabled. More specifically, we proposed the domain specific visualization based validation of model transformations, briefly VBV. The realization of VBV depends on the efficient handling of a special kind of compound transformations from the model transformations to be validated to VLM, a pre-defined Alloy module specifying a set of common graphical notations. Relevant instances of the to-be-validated model transformations are automatically generated by hybrid analysis and shown to domain experts for feedback in a visual format that they are familiar with. We prescribed a process to guide the application of VBV to model transformations and illustrated it with a case study, namely the validation of the benchmark CD2RDBMS transformation.

The combined power of hybrid analysis and domain specific visualization of generated instances provides us with a lot of potential in terms of formal method (in our case Alloy) based model transformation validation. On one hand, hybrid analysis gives a solution to the performance and scalability problems that are the most challenged when it comes to the application of Alloy. On the other hand, domain-specific visualization makes it possible for domain experts to effectively and efficiently be involved in the process of validation, which is, according to the experience gained in the case study, not only beneficial but also necessary in pinpointing some errors.

A more thorough evaluation of F-Alloy's expressiveness is ongoing (for both basic and compound model transformation specification) and several case studies have been implemented and available on the Lightning tool website.[3]

[3] http://lightning.gforge.uni.lu/examples.

As future work, we plan to apply our approach to more cases. In addition, despite the great increase of efficiency exposed in this paper, one might still shun using F-Alloy due to the complexity of its formal syntax. This is a common critique to formal method based approaches. To make our approach more accessible, we also plan to extend F-Alloy with a user-friendly graphical concrete syntax. Ideally, this user-friendly graphical concrete syntax would be based on the domain specific visualization of a transformation's input and output module, hence allowing domain experts to directly write or refine the transformation on top of validating it. Finally, our domain specific visualization based validation approach also complements the other model transformation validation techniques naturally. It can not only exist as a stand-alone technique, but can also be combined with other validation techniques to improve their usability. For example, the combination of assertion-based validation technique, which is also supported by Alloy, with domain specific visualization would make an interesting direction to look at.

References

[1] K. Anastasakis, A Model Driven Approach for the Automated Analysis of UML Class Diagrams, University of Birmingham, 2009.
[2] K. Anastasakis, B. Bordbar, J.M. Küster, Analysis of model transformations via Alloy, in: Proceedings of the 4th MoDeVVa Workshop: Model-Driven Engineering, Verification, and Validation, 2007, pp. 47–56.
[3] N. Asoudeh, R. Khosravi, Alloy as a Language for Domain Modeling, School of Electrical and Computer Engineering, Teheran, 2007.
[4] L. Baresi, P. Spoletini, On the use of alloy to analyze graph transformation systems, in: Proceedings of the 3rd ICGT: International Conference on Graph Transformations, in: LNCS, vol. 4178, 2006, pp. 306–320.
[5] J. Bézivin, B. Rumpe, A. Schürr, L. Tratt, Model transformations in practice workshop, in: Satellite Events at the MoDELS 2005 Conference, Springer, 2006, pp. 120–127.
[6] T.T. Dinh-Trong, S. Ghosh, R.B. France, A systematic approach to generate inputs to test UML design models, in: 17th International Symposium on Software Reliability Engineering, ISSRE'06, IEEE, 2006, pp. 95–104.
[7] A.A. El Ghazi, M. Taghdiri, Analyzing Alloy constraints using an SMT solver: a case study, in: 5th International Workshop on Automated Formal Methods, AFM, Edinburgh, 2010.
[8] L. Gammaitoni, N. Hochgeschwender, Rpsl meets lightning: a model-based approach to design space exploration of robot perception systems, in: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, 2016, http://ieeexplore.ieee.org/document/7862378/.
[9] L. Gammaitoni, P. Kelsen, Domain-specific visualization of Alloy instances, in: ABZ, Springer, 2014, pp. 324–327.
[10] L. Gammaitoni, P. Kelsen, F-Alloy: an Alloy based model transformation language, in: Proceedings of the 8th International Conference on Theory and Practice of Model Transformations, ICMT 2015, in: LNCS, vol. 9152, 2015, pp. 166–180.
[11] L. Gammaitoni, P. Kelsen, C. Glodt, Designing languages using lightning, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, 2015.
[12] L. Gammaitoni, P. Kelsen, Q. Ma, Agile validation of higher order transformations using F-Alloy, in: 2016 10th International Symposium on Theoretical Aspects of Software Engineering, TASE, 2016, pp. 125–131.
[13] L. Gammaitoni, P. Kelsen, F. Mathey, Verifying modelling languages using Lightning: a case study, in: Proceedings of the 11th MoDeVVa Workshop: Model-Driven Engineering, Verification, and Validation, Springer, 2014, pp. 19–28.
[14] S. Ganov, S. Khurshid, D.E. Perry, Annotations for Alloy: automated incremental analysis using domain specific solvers, in: Formal Methods and Software Engineering, Springer, 2012, pp. 414–429.
[15] N. Hochgeschwender, S. Schneider, H. Voos, G.K. Kraetzschmar, Declarative specification of robot perception architectures, in: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, 2014.
[16] D. Jackson, Software Abstractions, MIT Press, Cambridge, 2012.
[17] S. Khurshid, D. Jackson, Exploring the design of an intentional naming scheme with an automatic constraint analyzer, in: The Fifteenth IEEE International Conference on Automated Software Engineering, 2000. Proceedings ASE 2000, 2000, pp. 13–22.
[18] K. Lano, T. Clark, S. Kolahdouz-Rahimi, A framework for model transformation verification, Form. Asp. Comput. 27 (1) (2015) 193–235.
[19] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, M. Rinard, Optimizations for compiling declarative models into boolean formulas, in: Theory and Applications of Satisfiability Testing, Springer, 2005, pp. 187–202.
[20] T. Mens, P.V. Gorp, A taxonomy of model transformation, Electron. Notes Theor. Comput. Sci. 152 (2006) 125–142.
[21] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, D. Jackson, Software assurance by bounded exhaustive testing, Softw. Eng. Notes 29 (4) (2004) 133–142.
[22] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, J. Bézivin, On the use of higher-order model transformations, in: Proceedings of the 5th European Conference on Model Driven Architecture – Foundations and Applications, ECMDA-FA, 2009, 2009, pp. 18–33.
[23] E. Uzuncaova, S. Khurshid, Constraint prioritization for efficient analysis of declarative models, in: Proceedings of the 15th FM: International Symposium on Formal Methods, Springer, 2008, pp. 310–325.
[24] M.F. van Amstel, M.G.J. van den Brand, A. Serebrenik, Traceability visualization in model transformations with tracevis, in: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations, ICMT 2012, 2012, pp. 152–159.
[25] A. Zaman, I. Kazerani, M. Patki, B. Guntoori, D. Rayside, Improved Visualization of Relational Logic Models, Tech. Rep., University of Waterloo, 2013, pp. 152–159.