CoqTL: A Coq DSL for Rule-Based Model Transformation

Zheng Cheng, Massimo Tisi, Rémi Douence

Software & Systems Modeling manuscript No.
(will be inserted by the editor)

CoqTL: A Coq DSL for Rule-Based Model Transformation

Zheng Cheng · Massimo Tisi · Rémi Douence

Received: date / Accepted: date

Abstract In model-driven engineering, model transformation (MT) verification is essential for reliably producing software artifacts. While recent advancements have enabled automatic Hoare-style verification for non-trivial MTs, there are certain verification tasks (e.g. induction) that are intrinsically difficult to automate. Existing tools that aim at simplifying the interactive verification of MTs typically translate the MT specification (e.g. in ATL) and properties to prove (e.g. in OCL) into an interactive theorem prover. However, since the MT specification and proof phases happen in separate languages, the proof developer needs a detailed knowledge of the translation logic. Naturally, any error in the MT translation could cause unsound verification, i.e. the MT executed in the original environment may have different semantics from the verified MT.

We propose an alternative solution by designing and implementing an internal domain specific language, namely CoqTL, for the specification of declarative MTs directly in the Coq interactive theorem prover. Expressions in CoqTL are written in Gallina (the specification language of Coq), increasing the possibilities of reusing native Coq libraries in the transformation definition and proof. CoqTL specifications can be directly executed by our transformation engine encoded in Coq, or a certified implementation of the transformation can be generated by the native Coq extraction mechanism. We ensure that CoqTL has the same expressive power of Gallina (i.e. if a MT can be computed in Gallina, then it can also be represented in CoqTL). In this article, we introduce CoqTL, evaluate its practical applicability on a use case, and identify its current limitations.

Zheng Cheng
ICAM, LS2N (UMR CNRS 6004), Nantes, France
E-mail: zheng.cheng@icam.fr

Massimo Tisi
IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France
E-mail: massimo.tisi@imt-atlantique.fr

Rémi Douence
IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France
E-mail: remi.douence@imt-atlantique.fr

Keywords Model-Driven Engineering · Model Transformation · Domain-specific Language · Interactive Theorem Proving · Coq

1 Introduction

Model-driven engineering (MDE), i.e. software engineering centered on software models and MTs, is widely recognized as an effective way to manage the complexity of software development. With the increasing complexity of MTs (e.g., in automotive industry [31], medical data processing [38], aviation [4]), it is urgent to develop techniques and tools that prevent incorrect MTs from generating faulty models. The effects of such faulty models could be unpredictably propagated into subsequent MDE steps, e.g. code generation.

Deductive verification is a promising approach for quality assurance in MT: *correctness* is specified by MT developers using contracts (i.e. pre/postconditions), then the semantics of the MT language together with contracts and metamodels are encoded into a deductive theorem prover. Thanks especially to recent advancements in SMT solvers, automatic deductive verification is giving good results in several scenarios [7,6,10,27]. However, because of the general undecidability, interactive deductive verification is inevitable for complex tasks (for instance, automatic deductive theorem provers usually lack support for induction, or finding witnesses for existential quantifiers).

Coq is an interactive theorem prover. The user can use Coq to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of proofs (in the sense that routine proofs can be automatically performed while difficult proofs require human guidance). It has been used to prove non-trivial mathematical theorems, or as an environment for developing formally certified software and hardware (e.g. [25,16]). Moreover, a certified executable program (e.g., in OCaml, Haskell) can be generated from a Coq function by the native Coq extraction mechanism. While not strictly needed for understanding this paper, we refer the reader to [29] for an introduction to the Coq system.

Previous work aiming at simplifying the interactive verification of MTs, has already proposed translations from MT specifications (e.g. in MT languages like ATL) and properties to prove (e.g. in OCL) into Coq. However, the practical applicability of this translational approach is hampered by the fact that the two phases of MT specification and correctness proof require developments in languages (e.g. ATL+OCL and Coq, respectively) at two different levels of abstraction. The proof developer needs a deep knowledge of the translation logic to be able to write meaningful proofs. Any change in the MT code propagates through the translator, and it is difficult to predict the proof steps that will be invalidated. Naturally, any error in the MT translation could cause unsound verification, i.e. the MT executed in the original environment may have different semantics from the verified MT. Certifying that the semantics of the MT language has being correctly axiomatized in the back-end theorem prover is a hard task, and very few attempts exist [10,1].

Coq includes Gallina, a functional programming language with pattern matching and a rich type system, well suited as a platform for embedding domain-specific programming languages (DSLs) (e.g. [12]). In this work, we draw on this aspect of Coq and propose a DSL, namely CoqTL, to turn Coq into a tool for developing

certified MTs. We argue that using an internal DSL for the MT specification phase simplifies the iterative process of MT development and proof in MDE. Moreover, expressions in CoqTL are directly written in Gallina, increasing the possibilities of reuse of sophisticated native Coq libraries during the transformation definition and proof.

Our main contributions are:

- We design and implement CoqTL, to our knowledge the first DSL for rule-based MT in Coq (Section 3.2). The language is both functional and declarative in style, its syntax and semantics is inspired from ATL [21] (hence it should be familiar also to users of other rule-based MT languages, like ETL [22], or RubyTL [14]). Thus, CoqTL aims to lighten the cognitive load of MT developers while building certified MTs in Coq.
- We design and implement a transformation engine in Coq that interprets programs written in CoqTL to transform models (Section 3.4). The engine includes an on-the-fly parser that transforms the domain-specific syntax into a Coq data structure to interpret. The parser is transparently invoked by the Coq Notation mechanism, so that any existing Coq development environment is able to support the domain-specific CoqTL syntax without requiring ad-hoc modifications.
- CoqTL has the same expressive power of Gallina: if a MT can be computed in Gallina, then it can also be represented in CoqTL. We provide a constructive proof for this result in Section 3.5. Gallina is not Turing-complete, since it does not allow for nonterminating computation. However the class of computable functions in Gallina is very large (any function that can be shown to be total in ZFC with countably many inaccessibles can be defined in Gallina [39]), hence CoqTL can represent all transformations of any practical interest.

We show the practical applicability of CoqTL in proving MTs (Section 4.1), by using it to specify a sample transformation, prove non-trivial contracts over it and automatically extract a certified implementation. We also show the effectiveness of CoqTL in specifying MTs, by comparing the effort to implement sample transformations using CoqTL, ATL and pure Gallina (Section 4.2). We make CoqTL publicly available as open source[1]. The repository contains also the example and proofs described in this paper.

This paper extends an article contributed to the ICMT 2018 conference [34] that introduced the fundamental design of CoqTL. Here we extend CoqTL by adding iterative rules (Section 3.3 and 3.4). This extension has primary importance, since it increases the expressive power of CoqTL to its theoretical maximum: we can now prove that the extended version of CoqTL reaches the same expressive power as Gallina in representing computable MTs (Section 3.5). We also add new case studies to exemplify the effectiveness of CoqTL in specifying MTs w.r.t. to ATL and standard Gallina (Section 4.2).

Paper organization. We motivate our work by a sample transformation in Section 2. Section 3 illustrates our design of CoqTL in detail. In Section 4 we discuss the applicability of CoqTL. Section 5 compares our work with related research, and Section 6 draws conclusions and lines for future work.

[1] CoqTL (online). `https://github.com/atlanmod/CoqTL`

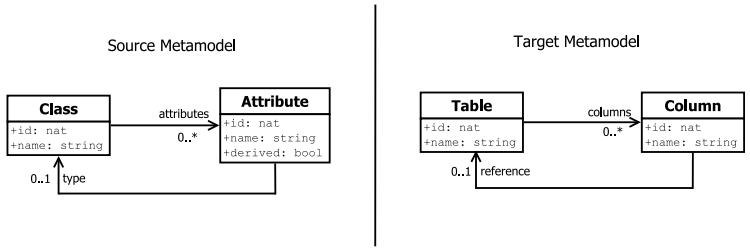

Fig. 1 A simplified structural metamodel for class diagrams (left), and relational schemas (right)

2 Class to Relational in CoqTL

We consider a simplified MT from class diagrams to relational schemas (arguably, the Hello World transformation in the MT community). The example is intentionally minimal, so that it can be completely illustrated within this paper. However we believe it to be easily generalizable by the reader to more complex scenarios. The structure of the involved metamodels is shown in Fig. 1.

The left part of Fig. 1 shows the simplified structural metamodel of class diagrams. Each class diagram contains a list of named classes with identities. Each class contains a list of named and typed attributes with unique identities. In this simplified model we do not consider attribute multiplicity (i.e., all attributes are single-valued). Primitive data types are not explicitly modeled, thus we consider every attribute without an associated *type* to have primitive data type. A *derived* feature identifies which attributes are derived from other values. The simplified structural metamodel of relational schemas is shown on the right part of Fig. 1. *Tables* contain *Columns*, *Columns* can *refer* to other *Tables* in case of foreign keys.

In Listing 1 we use the CoqTL language to specify how to transform class diagrams to relational schemas (we refer to Grammar 1 for the concrete syntax of CoqTL). A transformation is a Coq *Definition*. First, we declare that a transformation named *Class2Relational* is to transform a model conforming to the *Class* metamodel to a model conforming to the *Relational* metamodel, and we name the input model as *m* (lines 2- 3).

Then, the transformation is defined via two rules in a mapping style: one maps *Classes* to *Tables*, another one maps *non-derived Attributes* to *Columns*. Each rule in CoqTL has a *from* section that specifies the input pattern to be matched in the source model. A boolean expression in Gallina can be added as guard, and a rule is applicable only if the guard evaluates to true for a certain assignment of the input pattern elements. Each rule has a *to* section which specifies elements and links to be created in the target model (output pattern) when a rule is fired. The *to* section is formed by a list of labeled *outputs*, each one including an *element* and a list of *links* to create. The *element* section includes standard Gallina code to instantiate the new element specifying the value of its attributes (line 11). The *links* section contains standard Gallina code to instantiate links related to the previous element (lines 14-16).

For instance in the *Class2Table* rule, once a class *c* is matched (lines 6 to 7), we specify that a table should be constructed by the constructor *BuildTable*, with

```
1  Definition Class2Relational :=
2    transformation from ClassMetamodel to RelationalMetamodel
3      with m as ClassModel := [
4
5      rule Class2Table
6        from
7          c class Class
8        to [
9          "tab" :
10           t class Table :=
11             BuildTable newId (getClassName c)
12           with [
13             ref TableColumns :=
14               attrs ← getClassAttributes c m;
15               cols ← resolveAll Class2Relational m "col" Column (singletons attrs);
16               return BuildTableColumns t cols
17           ]
18       ];
19
20     rule Attribute2Column
21       from
22         a class Attribute from ClassMetamodel
23           when (negb (getAttributeDerived a))
24       to [
25         "col" :
26           c class Column :=
27             BuildColumn newId (getAttributeName a)
28           with [
29             ref ColumnReference :=
30               cl ← getAttributeType a m;
31               tb ← resolve Class2Relational m "tab" Table [cl];
32               return BuildColumnReference c tb
33           ]
34       ]
35   ].
```

Listing 1 Class2Relational MT in CoqTL

a newly generated *id* and the same *name* of *c* (line 11)[2]. While the body of target element creation (line 11) can contain any Gallina code, it is type-checked against the *class* signature of target element (line 10), i.e. in this case it must return a *Table*.

In order to link the generated table *t* to the columns it contains, we get the *attributes* of the matched class (line 14), resolve them to their corresponding *Columns*, generated by any other rule (line 15), and construct new set of links connecting the table and these columns (line 16). While this is standard Gallina code, we use for this example an imperative style with a monadic notation (←) that makes the sequential code clearer in this case[3]. The *resolveAll* function will only return the correctly resolved attributes. In particular derived *Attributes* do not generate *Columns* (i.e. they are not matched by *Attribute2Column*), so they will be automatically filtered out by *resolveAll*. The result of this Gallina code (i.e. the constructed links) are type-checked against the *ref* signature of target element (i.e. in this case they must have type *TableColumns*, as specified at line 13).

[2] Detailed semantics of constructors and accessors in the expression code can be found in the metamodel interface illustrated in Section 3.1.

[3] ← is our notation for option monad, its intuitive semantics is: if the right-hand-side of the arrow is not *None*, then assign it to the variable in the left-hand side and evaluate the next line, otherwise return *None*

```
Theorem Table_name_definedness :
  ∀ (cm : ClassModel) (rm : RelationalModel),
    (* transformation *)
    rm = execute Class2Relational cm
    →
    (* precondition *)
    (∀ (i : Class), In i (allModelElements cm)→ length (getClassName i)>0)
    →
    (* postcondition *)
    (∀ (o : Table), In o (allModelElements rm)→ length (getTableName o)>0).
```

Listing 2 Name definedness theorem for the *Class2Relational* transformation

In the *Attribute2Column* rule we can notice the presence of a guard. When the *Attribute* is not *derived*, a *Column* is constructed with the same name of the *Attribute* and a fresh identifier. If the original attribute is *typed* by another *Class*, we build a reference link to declare that the generated *Column* is a foreign key of a *Table* in the schema. This *Table* is found by resolving (*resolve* function) the *Class* type of the attribute.

To make sure *Class2Relational* is correctly implemented, there are many deductive approaches have been proposed [7,6,10,27]. They aim to certify transformations and prevent them generating faulty models, thereby giving more confidence for the subsequent MDE steps, e.g. code generation. However, these approach are hampered by the two phases of MT specification and correctness proof require developments in languages (e.g. ATL+OCL and Coq, respectively) at two different levels of abstraction. Besides, the proof developer needs a deep knowledge of the translation logic to be able to write meaningful proofs. Any change in the MT code propagates through the translator, and it is difficult to predict the proof steps that will be invalidated.

CoqTL naturally enables deductive verification of MTs. Users can write theorems with the full power of Coq that apply pre/postconditions (correctness conditions, a.k.a. contracts [26], tracts [5]) to the MT. For example, Listing 2 defines a theorem stating that if all elements contained in the input model have not-empty *names*, by executing the *Class2Relational* MT, all generated elements in the output model will also have not-empty *names*. Interactively proving this simple theorem in Coq takes 56 lines of routine proof code (this short proof can be even automated by using modern automatic theorem provers [6,10]).

To illustrate more complex theorems, we prove that our transformation *preserves unreachability*. (Un)reachability is an important property for several models, e.g. one may typically need to demonstrate that error states in generated state machines are not reachable. In our simple Class2Relational example, one can inductively define reachability for classes (similarly for tables), i.e. a class is reachable from itself, and two classes are reachable if they are transitively linked by attributes. We can define an *unreachability preservation* theorem as follows: if a certain class is not reachable from a given class, their corresponding tables will not be reachable from each other. Interactively proving this theorem in CoqTL needs more than a thousand lines of proof code. The major difficulty comes from choosing the right induction strategy, and to our knowledge, the automatic proof of similar theorems

is not supported by existing work. The full proof in Coq is available on the paper website.

3 The Design of CoqTL

CoqTL is an internal DSL for MT in Coq. In this section, we describe four main parts of its design:

- (Section 3.1) Metamodels and models are encoded as graph structures that can be automatically translated from/to EMF.
- (Section 3.2) Transformation specifications are encoded as a data structure wrapped up in a user-friendly domain specific syntax.
- (Section 3.3) Within the transformation specifications, conditional repetitive generation of target-pattern instances is described by iterative mapping rules.
- (Section 3.4) A transformation engine executes the transformation specifications against input models.

Finally, we constructively prove that CoqTL has the same expressive power as full Gallina in the specification of computable MTs (Section 3.5).

3.1 Metamodels and Models

Our encoding of metamodels in Coq is based on inductive data types, as analogous encodings in related work. However, since expressions in CoqTL are written in native Gallina, transformation developers will need to directly access the metamodel interface in the expression code. As a consequence, a clear design requirement for the metamodel encoding in CoqTL is to provide the simplest native representation.

As an example, Listing 3 shows the basic definitions for encoding the *Relational* metamodel of Fig. 1. Each metaclass is represented by an inductive data type, with a single constructor whose arguments are the attributes of the metaclass. References between metaclasses are represented as separate inductive types, with a constructor requiring the source and target elements as arguments. Optional or multi-valued attributes and references are respectively represented using the *option* and *list* Coq types in the appropriate constructor argument (e.g. at line 15).

Constructing any model requires providing one list of model elements and one of links, as specified by the *Model* type in the CoqTL library (shown in lines 23-26 in the listing). These lists are typed by generic *ModelElement* and *ModelLink* types, that are meant to be the sum types for elements and links of the specific metamodel. For defining the type of Relational model, we first define the two sum types *RelationalModelElement*, sum of *Table* and *Column*, and *RelationalModelLink*, sum of *TableColumns* and *ColumnReference*[4]. The *RelationalModel* type is obtained by parameterizing *Model* with these sum types.

We create accessors for every attribute and reference of each metaclass. Notice that while attribute accessors need only to inspect the element passed as argument to retrieve the attribute value (e.g., *getTableId* and *getTableName* at lines 37-41), reference accessors need to pass through the list of links to find the ones connected

[4] For simplicity here we omit the definition of sum types, that requires familiarity with dependent types

```
(*** Metamodel classes and references ***)

Inductive Table : Set :=
    BuildTable :
      (* id *) nat →
      (* name *) string → Table.

Inductive Column : Set :=
    BuildColumn :
      (* id *) nat →
      (* name *) string → Column.

Inductive TableColumns : Set :=
    BuildTableColumns:
      Table → list Columns → TableColumns.

Inductive ColumnReference : Set :=
    BuildColumnReference:
      Column → Table → ColumnReference.

(*** Model (from CoqTL library) ***)

Inductive Model (ModelElement: Type) (ModelLink: Type): Type :=
    BuildModel:
        list ModelElement →
        list ModelLink → Model ModelElement ModelLink.

(*** Relational Model ***)

Inductive RelationalModelElement : Set := ...   (* sum type for elements *)
Inductive RelationalModelLink : Set := ...   (* sum type for links *)

Definition RelationalModel := Model RelationalModelElement RelationalModelLink.

(*** Table accessors ***)

Definition getTableId (t : Table) : nat :=
    match t with BuildTable id _ ⇒ id end.

Definition getTableName (t : Table) : string :=
    match t with BuildTable _ n ⇒ n end.

Definition getTableColumns (t : Table) (m : RelationalModel) :
    option (list Columns) := ...
```

Listing 3 Some basic definitions for the *Relational* models in Coq

to the element in parameter. Thus, reference accessors need to have the whole model as extra parameter (e.g., *getTableColumns* in the listing).

Listing 3 includes only a small portion of the encoding of the *Relational* metamodel in Fig. 1. The full encoding takes over 300 lines of Gallina code, and includes a reflective API. Briefly, metamodel classes are reified in a RelationalMetamodelClass type (with values corresponding to Table and Column), that is used as argument to reflective functions. The reflective API can be used for obtaining the metaclass of an element, checking that an element is an instance of a metaclass, and boxing/unboxing

a generic element from/to a specific metaclass. Similar functions are available for links.

We provide automatic translators from EMF models/metamodels[5]. While our representation allows us to encode any model/metamodel, our current translator does not automatically handle several features that are found in EMF (or similar modeling frameworks), that need to be manually encoded. In particular,

- No automatic support is provided for metaclass inheritance: the instance of a superclass can be provided as parameter of a subclass constructor, but the two instances (of superclass and subclass) need to be managed separately. This require transformation developers to be more careful in specifying pattern matching of rules.
- Constraints for reference multiplicity or strong containment are not directly carried to our Coq encoding, but can be encoded manually by users via extra pre/postconditions. As a result, unless these constraints are explicitly stated as assumptions in proofs, transformation developers cannot assume models are born with them by default. Moreover, compared to the solution that taking these extra constraints as axioms, we think our current solution is more viable since it avoids of the issue of axiom inconsistency (which causes all theorems being trivially proven).
- Bidirectional references currently have no special treatment: both sides are encoded as separate references, that need to be separately assigned in the transformation code. While such design results more verbose transformation definition, it makes its semantics more explicit and clearer.
- Finally, differently from EMF, identifiers are considered as normal attributes and elements are considered equal when all their attributes are. This decision makes rewriting in proofs more natural.

3.2 Transformation Specification

The main part of the CoqTL design is the transformation specification as a data structure wrapped up in a user-friendly domain specific syntax. Grammar 1 describes the concrete syntax of CoqTL in EBNF. With respect to what we already discussed in Section 2, the grammar shows that CoqTL can specify either: 1) simple mapping rules that transform a single source pattern instance to a single target pattern instance, or 2) iterative mapping rules that transform a single source pattern instance to multiple target pattern instances (see Section 3.3 for more detail). In addition, as indicated by the *header* production rule, CoqTL currently supports only transformations from a single source model to a single target model.

The way we implement the concrete syntax of CoqTL relies on the Notation facility of Coq. A notation is a symbolic abbreviation to denote a Gallina expression, and is one of the main commands that modifies the way Coq parses and prints the representation of expressions.

For example, the first notation shown in Listing 4 implements the production rules *link-def* and *link-decl* in Grammar 1. After the declaration of this notation, when the expression on the left-hand-side is matched, it is expanded in memory

[5] EMF model/metamodel translators. `https://github.com/atlanmod/CoqTL/tree/master/fr.inria.atlanmod.coqtl.generators`

⟨*transformation*⟩ ::= ⟨*header*⟩ '`:=`' '`[`' ⟨*rule-list*⟩ '`]`'

⟨*header*⟩ ::= '`transformation`' '`from`' ⟨*id*⟩ '`to`' ⟨*id*⟩ '`with`' ⟨*id*⟩ '`as`' ⟨*id*⟩

⟨*rule-list*⟩ ::= ⟨*rule*⟩ '`;`' ⟨*rule-list*⟩ | ⟨*rule*⟩

⟨*rule*⟩ ::= '`rule`' ⟨*id*⟩ '`from`' ⟨*input-pattern*⟩ '`for`' ⟨*iteration*⟩ '`to`' ⟨*output-pattern*⟩
| '`rule`' ⟨*id*⟩ '`from`' ⟨*input-pattern*⟩ '`to`' ⟨*output-pattern*⟩

⟨*input-pattern*⟩ ::= ⟨*elem-decl-list*⟩ '`when`' ⟨*gallina-expr*⟩ | ⟨*elem-decl-list*⟩

⟨*elem-decl-list*⟩ ::= ⟨*elem-decl*⟩ '`,`' ⟨*elem-decl-list*⟩ | ⟨*elem-decl*⟩

⟨*elem-decl*⟩ ::= ⟨*id*⟩ '`class`' ⟨*id*⟩

⟨*iteration*⟩ ::= ⟨*id*⟩ '`in`' ⟨*gallina-expr*⟩

⟨*output-pattern*⟩ ::= '`[`' ⟨*output-list*⟩ '`]`'

⟨*output-list*⟩ ::= ⟨*output-elem*⟩ '`;`' ⟨*output-list*⟩ | ⟨*output-elem*⟩

⟨*output-elem*⟩ ::= ⟨*string*⟩ '`:`' ⟨*elem-def*⟩ '`with`' '`[`' ⟨*link-def-list*⟩ '`]`'

⟨*elem-def*⟩ ::= ⟨*elem-decl*⟩ '`:=`' ⟨*gallina-expr*⟩

⟨*link-def-list*⟩ ::= ⟨*link-def*⟩ '`;`' ⟨*link-def-list*⟩ | ⟨*link-def*⟩

⟨*link-def*⟩ ::= ⟨*link-decl*⟩ '`:=`' ⟨*gallina-expr*⟩

⟨*link-decl*⟩ ::= '`ref`' ⟨*id*⟩

Grammar 1 Concrete syntax of the CoqTL language in EBNF

```
1  (* Output Link Definition *)
2  Notation "'reference' reftype 'from' tinstance ':=' refends" :=
3    (BuildOutputPatternLinkDefinition tinstance reftype refends)
4      (right associativity, at level 60).
5
6  (* Output Pattern Element *)
7  Notation "'output' elid 'element' elname 'class' eltype
8    'from' tinstance := eldef 'links' refdef" :=
9    (BuildOutputPatternElement eltype elid eldef (fun elname ⇒ refdef))
10     (right associativity, at level 60).
```

Listing 4 A few notations for CoqTL

to the right-hand-side. A notation allows also the specification of associativity and precedence levels, to solve parsing ambiguities. Notations can be seen as a very limited compiler, that compiles in one pass without memory. For this reason they strongly limit the classes of DSLs that can be implemented. In the implementation of CoqTL every notation is simply translated into an appropriate constructor, encapsulating the values matched by the notation (line 3). Whenever the notation is matching the declaration of some variable that needs to be visible to the rest of the code, we introduce a lambda expression as an argument of the constructor. This is shown in the second notation in Listing 4, that implements the *output-elem* production rule in the grammar. The created element *elname* needs to be visible in

the following *links* section, so we store the content of this section in an anonymous function with *elname* as argument (line 9).

The constructors used in our notations, like *BuildOutputPatternLinkDefinition* in Listing 4 build a representation of the abstract syntax of the CoqTL program. Hence CoqTL is a deeply embedded DSL for the rule structure part. CoqTL has however shallow embedding of expressions, to allow the direct use of the Gallina language for guards and output patterns (*gallina-expr* in the grammar).

Finally, CoqTL provides auxiliary functions meant to be used in Gallina expressions for guards and output patterns. The most important is the function *resolve* (and its corresponding multi-valued version, *resolveAll*) for element resolution. As illustrated at lines 15 and 31 in Listing 1, its signature requires the following arguments: the current transformation (*Class2Relational*), the source model (*m*), the label associated to the required output element, useful for rules with multiple output elements (*"col"*), the type of the expected result, useful for type checking (*Column*), the source pattern to resolve (or the list of source patterns in case of *resolveAll*). The main notable aspect in implementing the *resolve* function is that the matching phase is provided as a new application of the transformation in a specific *match* mode. While this choice negatively affects the global efficiency of the transformation, it simplifies the development of proofs, because it does not require to introduce a concept of transformation traces as side effects of the transformation execution.

3.3 Iterative Mapping Rules

Mapping rules (as found in several MT languages. e.g. ATL [21], ETL [22], QVTr [40], and RubyTL [14]) express mappings from single source pattern instances to single target pattern instances. However, not all computable MTs can be expressed purely in this style. Let us consider a transformation that unfolds a given (potentially cyclic) graph into a tree of a given height. We use the metamodel shown in Fig. 2 to represent graph (and tree) models, where each node can access its children nodes through a uni-directional reference *children*. An example of application of this transformation is shown in Fig. 3, where we specify as parameter that we want to generate a target tree of height 2.

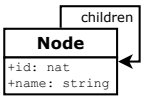

Fig. 2 A metamodel to represent graphs

The transformation algorithm starts at a given node, and keeps navigating the *children* reference of the input graph. At every iteration it copies the visited input nodes as children of the current output node. The recursion stops when the input node is leaf node (i.e. it has no children) or the given maximum height has been reached.

As we can see from Fig. 3, each node of the input graph needs to be replicated a number of times that depends on the global graph topology. E.g. *A* is replicated into *A1* and *A2* because it is reachable through two independent paths in the original

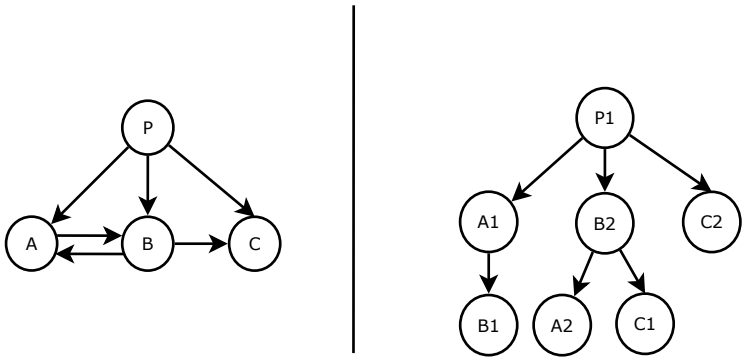

Fig. 3 A sample cyclic graph (left) and its corresponding cycle-free tree (right)

graph. A recursive function that implements this logic can be easily expressed in a functional language like Gallina. However, this transformation can not be expressed in CoqTL when using only mapping rules that transform from a single source pattern instance to a single target pattern instance.

To be able to define such kind of transformations, we need CoqTL to support a form of conditional repetition, either through recursion or iteration. In a proof-oriented language like CoqTL, we prefer iterative rule applications over recursive ones, since they induce a simple traceability from output to input, i.e. the capability to trace each target pattern instance back to the transformation state that triggered its generation. In our experience, such kind of traceability is crucial for simplifying proofs in MTs.

We include a primitive for iterative mappings in CoqTL. Specifically, rules in CoqTL can optionally become iterative mapping rules by specifying a *for* section, which is inspired by "foreach" constructs of general-purpose programming languages. The primitive is structured as `for i in coll` where: `coll` is an expression that computes a collection to be iterated on, and `i` is an iterator that indexes the current iteration on the collection, which will be passed to the target pattern generation (i.e. its value can be used when generating target elements/links). Each iterative mapping rule semantically means that: for each matched source pattern instance, a collection is computed by the expression `coll`, and a target pattern instance is constructed for each element of the collection, with `i` representing the current value in the collection. Thus, each target pattern instance generated by iterative rules can be traced back to a pair [source pattern instance, iterator value] that generated it, which constitutes a simple traceability relation for target elements.

In Listing 5, we demonstrate iterative mapping rules on the graph unfolding algorithm. The transformation is made of a single *Node2Node* rule. The source pattern matches all nodes (lines 6 - 7). The rule contains a *for* section that calculates how many times each matched node needs to be replicated (lines 8 - 9).

A user-defined Gallina function *allPathsTo* computes the collection of all paths in the original graph m, with at most length l (2 in this case), that starting by the initial node of the graph arrive to the matched node n. Each path is a sequence

```
1  Definition Graph2Tree :=
2    transformation from GraphMetamodel to GraphMetamodel
3      with m as GraphModel := [
4
5      rule Node2Node
6        from
7          n class Node
8        for
9          i in (allPathsTo m 2 n)
10       to
11         [
12           "n" :
13             n' class Node :=
14               BuildNode (getNodeId n i) (getNodeName n)
15             with
16               [
17                ref NodeEdgesEReference :=
18                  pth <- i;
19                  children <- getNodeEdges n m;
20                  iters <- Some (map (app pth) (singletons children));
21                  children' <- resolveAllIter Graph2Tree m "n" Node
22                                 (singletons children)
23                                 iters ;
24                  return BuildNodeEdges n' children'
25               ]
26         ];
27   ].
```

Listing 5 Graph2Tree MT in CoqTL

of nodes, where children follow fathers. For a matched source pattern, the target pattern specified in the *to* section (lines 11 - 26) will be created the number of times calculated by the *for* section. For example, in Fig. 3(left), among all paths with at most length 2 that start from the root node (7 paths in total), there are 2 paths that end with node *A*, i.e. P $\rightarrow$ A, and P $\rightarrow$ B $\rightarrow$ A. This triggers the two replications of A in Fig. 3(right).

The design of iterative mapping rules is driven by three design decisions. First, for each rule, the user can form the collection to be iterated on by specifying any computation in Gallina. Thus, CoqTL provide a significant flexibility in selecting the iteration collection that will facilitate the generation of target elements/links.

Second, because of the freedom in forming collections in the *for section*, any Gallina type can be used for iterators and different rules do not necessarily have the same iterator type. However, to enable a unified treatment by the engine, we require the user to communicate to the engine a sum type for all iterators used in its transformation, together with boxing/unboxing functions that cast to/from the sum type. The sum type is not necessary when the transformation contains just one iterator type: for instance in Listing 5, the transformation contains only one rule (*Node2Node*), whose *for section* type is a list of *Node*.

Third, we require the user to provide a proof for decidable equality of the used iterator types. For example, the transformation shown in Listing 5 requires the user to provide a proof for decidable equality of lists of nodes, which can be trivially proved on top of decidable equality of lists (by the standard Coq library), and decidable equality of nodes (whose proof is automatically generated by our metamodel translator). Our engine uses the proof of decidable equality to compare iterator values in a generic way.

The element resolution function needs to be aware of iterative rules. We define a new *resolveIter* function, that augments the signature of *resolve* with an extra argument for an iterator value. The additional argument alters the semantics of the *resolve* function, by constraining the resolution to the element generated on a specified iteration (indexed by the given iterator).

The multi-valued version *resolveAllIter* is implemented in a similar way, as demonstrated at lines 21-23 in Listing 5. It augments the signature of *resolveAll* with an extra argument for a list of iterator values, which resolves to a set of target elements. Each of them needs to be generated on the iteration indexed by the iterator at the corresponding position in the given list of iterators.

While users can specify rules without *for section*, all rules in the CoqTL engine are actually iterative. The CoqTL notations pads rules without *for section* with a default one, computing a singleton collection of the default *for section* type. Thus, when a mapping rule designed in this way is executed, each one of its matched source patterns will iterate exactly once to generate a single target pattern.

3.4 Transformation Engine

The CoqTL transformation engine, performs an interpretation of the transformation specification against an input model to generate an output model. Algorithm 1 illustrates in pseudo-code how our transformation engine works. This algorithm has been influenced by the execution algorithm of ATL [21] (notably in the distinction between a match/instantiate and an apply function), but is very different, having the objective to simplify the proof development, at the cost of sacrificing execution efficiency.

Our transformation engine is implemented in an *execute* function (called for instance in Listing 2) that takes as input a transformation specification R and an input model I (which contains elements I_e and links I_l). The output is elements O_e and links O_l, which form an output model.

First, the transformation engine records the maximum size (m) of input patterns among all the rules in the transformation specification. This value is used to calculate all the potential pattern instances P that the input model can produce to be matched against the transformation specification, i.e. all the subsets of I_e whose size is less or equal to m are enumerated.

Algorithm 1 Algorithm of the *execute* function

```
 1: m ← maxArity(R)
 2: P ← allPatterns(I_e, m)
 3: for each p ∈ P do
 4:    rs ← findRules(R, p)
 5:    for each r ∈ rs do
 6:       col ← evalForSection(r, m, p)
 7:       for each i ∈ col do
 8:          O_e ← O_e ∪ instantiate(r, I, p, i)
 9:          O_l ← O_l ∪ apply(r, I, p, i)
10:       end for
11:    end for
12: end for
```

Next, on line 3, the engine iterates on each potential pattern instance p, and seeks for **a list of rules** rs in R that matches it. This differentiates from languages like ATL, that allow for a single match of any given input pattern instance. The rationale being that when we restrict a source pattern to be matched by one rule, the engine can only produce either a single target pattern instance (as shown in Algorithm 1), or a set of homogeneous target pattern instances (by iterative rules). Multiple matching coupled with iterative rules, allow CoqTL to cover the most general case, where a source pattern is transformed to a heterogeneous set of target pattern instances.

For each rule r that matches the pattern instance p, its for section is evaluated to a collection of iterator elements *col*. When iterating on *col* with iterator *it*, the *instantiation* phase of r will be invoked to construct the corresponding output elements of p and add them to the output model. By doing so, each output element will always be generated from a target pattern on a certain iteration, thus establishing a clear trace relation for proofs. Finally the *apply* phase is invoked, i.e. to construct the corresponding output links and add them to the output model.

Notice that Gallina expressions for output links are only evaluated during the apply phase. The developer may include in these expressions calls to the *resolve*, *resolveAll*, *resolveIter* or *resolveAllIter* functions, whose evaluation requires the execution of the *instantiate* phase. As mentioned in Section 3.2, in our solution *resolve* depends on the result of the transformation execution in *match* mode. The algorithm implemented in *match* is similar to Algorithm 1, notably without line 9 (that would make the whole computation recur indefinitely). Multiple executions of the transformation for element resolution slow down the execution, but simplify the proofs, since no explicit traces are necessary as applications of *instantiate* and *apply* with identical inputs can be trivially checked for equality. Possible optimizations are however the subject of future work.

Note that rules are applied in their definition order in the CoqTL file, but this does not affect the result of the match phase, that produces a set of elements. However, the order of the rules affects the final transformation result, since the *resolve* function returns the target element generated by *the first rule* that matches the given source pattern.

As shown by Algorithm 1, a CoqTL transformation is essentially a *side-effect-free* functional program. The source model is read-only. The language allows for a limited read access to the target model while it is being constructed. The attributes of all the target elements can be read but the references can not be traversed.[6] Deletions and modifications target elements after their creation are never allowed. Such kind of limited read access is typical in relational transformation languages like ATL.

CoqTL transformations can be directly executed using the CoqTL engine (via "eval compute" command of Coq). This may be convenient during the transformation development. Finally, using the native Coq extraction mechanism, the transformation can be automatically *extracted* into a separate executable program in one of several languages (e.g., OCaml, Haskell) for interoperability or performance. The possible performance improvement is variable and depends on the Coq extractor and the target language implementation.

[6] Please refer to `https://bit.ly/2Y7b3cW` for an example of use of *resolve* to read a target element.

3.5 Expressive Power of CoqTL

The inclusion of iterative mapping rules have the important side effect of giving to CoqTL the same expressive power of its host language, Gallina, in representing computable MTs. Since the host language constitutes an upper bound to the expressive power of the internal one, we effectively maximize the expressive power of CoqTL. We prove this result by formally defining this concept as in Theorem 1, and providing a constructive proof for it.

Theorem 1 *Let $f_g : I \to O$ be a function in Gallina that transforms an input model of type I to an output model of type O, there exists a CoqTL specification $R : Tr_{CoqTL}$, such that executing R on any input model m will produce a target model that is isomorphic ($\cong$) to $(f_g\ m)$. That is:*

$\forall\ (f_g : I \to O),\ \exists\ (R : Tr_{CoqTL}),\ \forall\ (m : I),\ (f_g\ m) \cong (execute\ R\ m)$

Proof Let f_g be a MT purely encoded as a function in Gallina. We prove the theorem by constructing a CoqTL specification R and proving that executing R on any input model m will produce a target model that is isomorphic to $(f_g\ m)$.

The construction can be abstracted as in Listing 6. For each target type T_i in O, we define a rule r_i in R. The target metamodel has a finite set of types, so we need to define a finite set of rules. In the *from* section for r_i we just want to guarantee that the rule is always executed exactly once for each non-empty input model (the way we achieve it in Listing 6 is by matching the first element of the input model through the *hdElem* function, as in line 9). The *for* section of r_i calls f_g on the input model m and selects the output elements of type T_i from the produced output model. Thus, when the *for* section of each r_i iterates over the specified collection (line 11), each iterator *it* represents a pre-computed output element that needs to be created for this rule. The iterator *it* is passed to the *to* section, that has only the task of copying it to the transformation output.

By encoding the MT specification R in this way, we can prove our theorem by a case analysis on the input model m. When m is empty, none of the rules in R would be matched, thus $(execute\ R\ m)$ produces an empty target model as $(f_g\ m)$. When m is not empty, each rule in R would be matched exactly once, whose *for* section ensures that every element that is created by $(f_g\ m)$ will also be copied to the output by $(execute\ R\ m)$. □

Existing model-transformation languages like ATL can represent all computable model transformations. Instead Gallina is not Turing-complete, since it does not allow for nonterminating computation. However the class of computable functions in Gallina is very large (any function that can be shown to be total in ZFC with countably many inaccessibles can be defined in Gallina [39]), hence CoqTL can represent all transformations of any practical interest.

4 Practical Usage of CoqTL

In this section we consider two aspects of CoqTL: capacity of enabling practical verification for MTs (Section 4.1), and effectiveness of writing MTs (Section 4.2). Then, we conclude by a discussion of our case studies (Section 4.3). The artifacts to reproduce the experiments and the raw results are available in `https://github.com/atlanmod/CoqTL`.

```
1
2  Definition hdElem (m : InputModel) := (hd (allInstances m)).
3
4  Definition R := transformation from I to O with m as InputModel := [
5     r_i
6        from
7           e Class (Type (hdElem m))
8        when
9           e = (hdElem m)
10       for
11          it in (allInstancesOf (f_g m) T_i)
12       to
13           ["T_i" :
14              e' class T_i :=
15                 (instantiation e' T_i it)
16              with
17                 [ ( initialization e' T_i it) ] ] ;
18
19    ...
20 ]
```

Listing 6 Constructive proof template for Theorem 1

4.1 Proving theorems with CoqTL

To show that CoqTL can enable practical verification for MTs, we formulate 4 theorem proofs over the MT presented in Section 2. Some measures are shown in Table 1, to give the reader an idea of the complexity of the proofs: lines of code (LoC) and number of user-developed lemmas.

As a first theorem we prove that Class2Relational preserves id positivity, i.e. if all identifiers in the source model are positive, then they also are in the target model. In the first and second row we show two proofs for this theorem. In the second proof we obtain a reduction of about 60% LoC, thanks to the use of a generic lemma for transformation surjectivity, provided in the CoqTL library. This shows that CoqTL enables the design and proof of generic lemmas that make interactive verification more efficient and concise.

Transformation surjectivity states that for all elements contained in the output model there has to exist a rule and a matching input pattern that created them. Our design choices in CoqTL enable this kind of theorems: during the proof we can refer to syntactic elements of the transformation (e.g. *rules*, *input/output patterns*) by their type in the abstract syntax (e.g., *OutputPatternLinkDefinition* in Listing 4), and *quantify over them*. Moreover we use the reflective model API mentioned in Section 3.1 to reason on metamodel-agnostic properties.

The surjectivity lemma is also used in the third and fourth proof. In the third row we prove the name definedness property shown in Listing 2, separately for all

Table 1 Theorem proofs on *Class2Relational*

Theorem	**LoC**	**No. Lemmas**
positive_ids	180	4
positive_ids_surj	75	1
name_definedness	89	2
unreachability_preservation	1161	17

element types in source and target models. Finally by the fourth row, it is clear that the unreachability preservation theorem (Section 2) is difficult to prove, and shows the need of further work in proof engineering for MTs.

One road we want to follow is providing a complete library of generic lemmas for CoqTL such as transformation surjectivity, to shorten proofs on CoqTL. Some recurring proof patterns could be factorized into *domain-specific automatic proof tactics*, aware of the CoqTL representation and properties. Another line could be investigating a set of domain specific guidelines to construct proofs for MT verification. For example, to prove that if two *Tables* are reachable, the *Classes* that generated them are reachable too; we induct on the definition of reachablity. However other induction strategies, e.g. on the structure of the model, may be more efficient.

4.2 Writing and executing transformations in CoqTL

To have an idea of the effectiveness of CoqTL coding w.r.t. pure Gallina, we asked an expert functional programmer to develop the *Class2Relational* transformation as a function in Gallina and to prove the name definedness theorem. To ensure a fair comparison, the independent expert is asked to work on the same data representation of models/metamodels as presented in this work (Section 3.1), but is free to chose any means to encode the MT and prove the theorem.

The final MT in standard Gallina is composed by a total of 3 recursive functions encoding the transformation logic: one function that recursively transform class model elements to relational model elements, plus two functions that transform the bidirectional class model links to relational model links (one function for each direction). The equivalent CoqTL code is more concise (1194 vs. 2015 characters).

However the most interesting finding comes from the analysis of the proof. To prove the theorem, the Coq developer independently created and proved a version of the surjectivity lemma, specific to Class2Relation. By using CoqTL the developer would have simply been able to reuse the generic lemma provided in the CoqTL library. This experiment exemplifies the proof reuse potential of CoqTL.

To show the effectiveness of CoqTL in writing MTs as a DSL, we specify 4 transformations using ATL and CoqTL: 1) the *Class2Relational* transformation given in Section 2; 2) an extended version of the same transformation that has specific treatment of attributes based on whether they are multi-valued; 3) the transformation from hierarchical state machine to flattened state machine (HSM2FSM) given in [2,6]; 4) the Graph2Tree transformation given in Section 3.3. The first two transformations naturally encode mapping rules, while the difference being more complex usage of resolve function in the second case study. The third one demonstrates how to specify transformation in presence of inheritance in the metamodels. The last transformation illustrates the power of iterative mapping rules. Therefore, the chosen transformations demonstrate a complete coverage of the current CoqTL language.

We first implement these 4 transformations using ATL and CoqTL respectively, and ensure the correspondent ones are semantically equivalent. Table 2 shows meta information in this process. The 2nd and 3rd columns give the size of source and target metamodels respectively, in terms of number of classifiers, attributes and references. The last two columns show the information on specified transformation

in ATL and CoqTL respectively, in terms of number of rules, helpers and lines of code in the transformation.

Table 2 Specifying transformations using ATL and CoqTL

	Source Metamodel	Target metamodel	ATL	CoqTL
	(No. Class/Attr/Ref)		(No. Rule/Helper/Line)	
Class2Relational	2/5/2	2/4/2	2/0/16	2/0/36
Class2RelationalMV	2/5/2	2/4/2	3/0/35	3/0/79
Graph2Node	1/2/1	1/2/1	2/0/15	1/4/59
HSM2FSM	6/3/8	5/3/6	7/0/42	7/4/206

The main result from Table 2 is that ATL is still more concise than CoqTL in specifying MTs. One reason for this conciseness is that ATL delegates several low level model management operations to its underlying engine (e.g. bidirectional reference bindings, resolution), whereas CoqTL needs to manage these operations explicitly in the transformation logic. Moreover, ATL provides OCL libraries that facilitates user in specifying MTs, whereas CoqTL developers need to build them from scratch (by introducing CoqTL helpers). Work on a library for factorizing functions typically used during specification is ongoing.

Although CoqTL is currently not designed to be a high performance language for large scale MTs, we conduct a small experiment to give an idea of execution time of CoqTL compared to ATL on the chosen 4 transformations. The experiment is performed on a 4-core computer with 16 GB RAM running on a Windows 64bit operating system. We use ATL v3.8 on the standard EMFTVM engine, and CoqTL is implemented on top of Coq v8.8.1. Each transformations is run against two kinds of models: The first model has roughly 50 model elements, and 50 model links. The second model has roughly 250 model elements, and 250 model links. To measure execution time, we use the native profiler of ATL, and Coq native "Time" command. Time is measured in milliseconds. To make the result more accurate, we run the transformations 10 times, and calculate the average.

Table 3 Model transformation execution time in ATL and CoqTL (milliseconds)

	Model1 (50 els)		Model2 (250 els)	
	ATL	CoqTL	ATL	CoqTL
Class2Relational	1	83	4	566
Class2RelationalMV	3	363	16	2177
Graph2Node	N/A	816	N/A	16625
HSM2FSM	720	>600000	69227	>600000

The result of performance evaluation is shown in Table 3. We can see that ATL is generally more than one hundred times faster than CoqTL to execute the given transformations. CoqTL enumerates all the possible input pattern instances and then it matches them against transformation rules, which significantly impacts execution performance. Pattern matching in *HSM2FSM* is more complex than the other case studies, hence enumerating all the possible input pattern instances causes the transformation to exceed the imposed timeout (10 minutes). The *Graph2Tree*

transformation cannot be executed on the standard ATL engine, whereas CoqTL has no trouble to execute it on reasonably-sized models.

4.3 Discussion

In our long experience in developing model transformations, it is not trivial to guarantee the respect of given correctness criteria in all cases, even for simple transformation tasks. We believe that the lack of popularity of traditional approaches to theorem proving for model transformation is mainly caused by two issues that are solved by CoqTL:

- Theorem proving on transformations requires switching between the level of abstraction of models (e.g., in ATL, OCL, etc..) and the level of abstraction of the theorem prover (e.g., in Coq, Isabelle). CoqTL avoids this yo-yo problem by enabling the use of the same model-level primitives in the transformation language and in proofs. For instance, during the proof users can directly manipulate modeling elements (e.g. of type `RelationalModel`, `Table`, `Column` from Listing 3) and transformation syntactic elements (e.g., of type `Transformation`, `Rule`, `OutputPatternElement` from Grammar 1). Traditional approaches translate modeling concepts in functional counterparts, with completely separated type systems.
- The compilation of OCL expressions to Gallina is particularly problematic (given the termination requirements of Gallina). CoqTL avoids the translation issues by using directly Gallina as an expression language. This also allows users to exploit the full set of existing Coq libraries for mathematical reasoning during transformation.

While the concrete potential of CoqTL will only be demonstrated by an extensive user evaluation, the aim of this paper is showing that CoqTL is a solution to these two problems. In the evaluation we observe the practical applicability of CoqTL in proving MTs, by using it to specify transformations, prove non-trivial contracts over them and automatically extract certified implementations. We show that CoqTL allows for concise specifications of MTs, when compared with pure Gallina, while not reaching the compactness of ATL. We recognize the importance of a traceability relation in proving properties of MTs, that can be naturally encoded in CoqTL as generic reusable lemmas (transformation surjectivity).

The following aspects need particular attention:

Generality. More case studies are underway to further explore the full potential of CoqTL in building certified MTs. The roadmap for future CoqTL development includes also organizing more transformation language properties (e.g. additivity [20]) into an interface, which aims to make CoqTL proofs more accessible.

Usability. Currently, CoqTL depends on the experience of users in debugging MTs/proofs in Coq. This can be time-consuming, also given the state of the software engineering tools in the Coq ecosystem. We plan to produce tools that would make debugging more user-friendly for CoqTL, like more meaningful error report (based on our work on fault localization [11]), and counter-example generation (e.g. by model finding [13] and random testing [3]).

Performance. As discussed in Section 3.4, the main goal of CoqTL design is simplifying proofs, and this comes with a significant tradeoff on pure execution performance. However, to make it more appealing to practitioners, the performance of CoqTL needs to be improved. In Section 4.2, we identify some performance problems. In current work we are investigating solutions to improve performance without complicating proofs: our strategy is decoupling proofs from implementation optimization by providing a stable specification of CoqTL in the form of a set of theorems.

5 Related Work

There are many automatic theorem proving approaches for MTs (e.g. [7, 6, 10, 27]). However, interactive theorem proving is inevitable for more serious verification tasks. In this section, we focus on recent advancements of MT verification based on interactive theorem proving. To our knowledge, none of the existing works designs and implements DSLs for MT within interactive theorem provers.

Yang et al. interactively verify that a particular MT, i.e. from AADL to TASM language, is semantic preserving [41]. The approach is based on providing a translational semantics of both languages as timed transition systems in Coq and then reasoning on their equivalence. CoqTL could be used to simplify this kind of work.

Most previous works focus on giving a translational semantics of a MT language towards the target theorem prover. Generally they do not investigate a way to formally ensure that the semantics of the MT language has been axiomatized correctly in the back-end theorem prover. Calegari et al. encode ATL MTs and OCL contracts into Coq to interactively verify that the MT is able to produce target models that satisfy the given contracts [8]. In [32], a Hoare-style calculus is developed by Stenzel et al. in the KIV prover to analyze transformations expressed in (a subset of) QVT Operational. UML-RSDS is a tool-set for developing correct MTs by construction [23]. It chooses well-accepted concepts in MDE to make their approach more accessible by developers to specify MTs. Then, the MTs are verified against contracts by translating both into interactive theorem provers.

Kezadri et al. defines the Coq4MDE framework to formally embed some key aspects of MDE in Coq [17]. We have a similar abstraction of metamodels as graphs. While our understanding is that Coq4MDE is capable of embedding MT languages and enabling MT verification, no specific work has been proposed. We expect an evaluation in the future to compare the complexity of MT verification between the two works.

Poernomo and Terrell follow the classical approach in type theory to formally specify MTs as $\forall\exists$ types in interactive theorem provers [30]. Their approach does not target any specific MT languages. In addition, although their work does not propose a generic MT engine as we presented here, a corresponding executable MT program can be extracted once the MT is proved. The approach is further extended by Fernández and Terrell on using co-inductive types to encode bi-directional or circular references [15]. We also plan to investigate how co-inductive types can cooperate with our encoding and proofs (e.g. guardedness issues of co-recursive functions might arise because the syntactic criterion applied by the Coq system is too rigid [28]).

Tisi et al. use mapping rules with lazy rules in ATL to generate potentially infinite number of elements, and combine with a lazy execution strategy, driven by external model-consumption events to determine when to stop generation [35]. However, the main problem with recursion to support conditional repetitions is that it is difficult to establish traceability, i.e. difficult to track each target pattern back to a source pattern and a transformation state that generate it. Consequently, proofs that depend on traceability would become more challenging than our iterative design.

We particularly focus on DSL design for mapping style rule-based MT languages. There are many other MT languages with well-founded semantics, but targeting different theoretical background / paradigm than our concern in this work, e.g. graph transformation languages [24,36,33], or bi-directional MT languages [18, 19]. However, we are interested in whether our approach can be systematically migrated to these languages for their DSL design in Coq.

6 Conclusion

In conclusion, we present CoqTL, to our knowledge the first DSL in Coq for MTs and their verification, that strengthens the safety of MDE. CoqTL is both functional and declarative in style, providing a familiar environment for transformation developers in Coq. Its main primitives to express MTs are iterative mapping rules that specify conditional repetitive generation of target pattern from source pattern. CoqTL has the same expressive power as Gallina in specifying MTs, but with a user-friendly rule-based concrete syntax to lighten the cognitive load of MT developers. Inside rules, the Gallina specification language can be used to express complex transformation logic, which also increases the possibilities to reuse native Coq libraries in the transformation definition and proof. The underlining transformation engine of CoqTL is implemented in Coq, allowing the execution of transformations directly within Coq. We show the practical applicability of CoqTL in proving MTs, by using it to specify a sample transformation, prove non-trivial contracts over it and automatically extract a certified implementation. We also show the effectiveness of CoqTL in specifying MTs, by comparing the implementation of a sample transformation using CoqTL and pure Gallina.

Our future work will focus on the issues we identified in different points of our discussion. We plan to improve the performance of the CoqTL engine (e.g. by advanced model index / query methods as in [37,9]) without hampering its capability of enabling simple proofs. We want to develop a theorem library on top of CoqTL to facilitate MT verification, including transformation-agnostic lemmas such as transformation surjectivity, and domain-specific proof tactics to automatize recurring proof steps. We aim to investigate whether there are domain-specific guidelines to construct proofs for MT verification. Last, we want to improve interoperability between CoqTL and common MDE tools such as EMF, for industry readiness.

References

1. Ab.Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software & Systems Modeling 14(2), 1003–1028 (2015)

2. Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.M.: Barriers to systematic model transformation testing. Communications of the ACM 53(6), 139–143 (2010)
3. Berghofer, S., Nipkow, T.: Random Testing in Isabelle/HOL. In: 2nd International Conference on Software Engineering and Formal Methods. pp. 230–239. IEEE, Beijing, China (2004)
4. Berry, G.: Synchronous design and verification of critical embedded systems using SCADE and Esterel. In: 12th International Workshop on Formal Methods for Industrial Critical Systems, pp. 2–2. Springer, Berlin, Germany (2008)
5. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. IEEE Transactions on Software Engineering 41(5), 490–506 (2015)
6. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: 15th International Conference on Model Driven Engineering Languages and Systems. pp. 198–213. Springer, Innsbruck, Austria (2012)
7. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: 14th International Conference on Formal Engineering Methods. pp. 198–213. Springer, Kyoto, Japan (2012)
8. Calegari, D., Luna, C., Szasz, N., Tasistro, Á.: A type-theoretic framework for certified model transformations. In: 13th Brazilian Symposium on Formal Methods. pp. 112–127. Springer, Natal, Brazil (2011)
9. Cheng, L., Kotoulas, S.: Scale-Out Processing of Large RDF Datasets. IEEE Transactions on Big Data 1(4), 138–150 (2015)
10. Cheng, Z., Monahan, R., Power, J.F.: A sound execution semantics for ATL via translation validation. In: 8th International Conference on Model Transformation. pp. 133–148. Springer, L'Aquila, Italy (2015)
11. Cheng, Z., Tisi, M.: Slicing ATL model transformations for scalable deductive verification and fault localization. International Journal on Software Tools for Technology Transfer 20(6), 645–663 (2018)
12. Chlipala, A.: The Bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. In: 18th ACM SIGPLAN International Conference on Functional Programming. pp. 391–402. ICFP '13, ACM, Boston, Massachusetts, USA (2013)
13. Cuadrado, J.S., Guerra, E., de Lara, J.: Uncovering errors in ATL model transformations using static analysis and constraint solving. In: 25th IEEE International Symposium on Software Reliability Engineering. pp. 34–44. IEEE, Naples, Italy (2014)
14. Cuadrado, J.S., Molina, J.G., Tortosa, M.M.: RubyTL: A practical, extensible transformation language. In: 2nd European Conference on Model Driven Architecture: Foundations and Applications. pp. 158–172. Springer, Bilbao, Spain (2006)
15. Fernández, M., Terrell, J.: Assembling the proofs of ordered model transformations. In: 10th International Workshop on Formal Engineering approaches to Software Components and Architectures. pp. 63–77. EPTCS, Rome, Italy (2013)
16. Gu, R., Shao, Z., Chen, H., Wu, X., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent os kernels. In: 12th USENIX Conference on Operating Systems Design and Implementation. pp. 653–669. USENIX Association, Berkeley, CA, USA (2016)
17. Hamiaz, M.K., Pantel, M., Combemale, B., Thirioux, X.: A formal framework to prove the correctness of model driven engineering composition operators. In: International Conference on Formal Engineering Methods (2014)
18. He, X., Hu, Z.: Putback-based bidirectional model transformations. In: 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 434–444. ACM, FL, USA (2018)
19. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K.: GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In: 26th IEEE/ACM International Conference on Automated Software Engineering. pp. 480–483. ACM, KS, USA (2011)
20. Hidaka, S., Jouault, F., Tisi, M.: On additivity in transformation languages. In: 20th International Conference on Model Driven Engineering Languages and Systems. pp. 23–33. ACM/IEEE, Austin, TX, USA (2017)
21. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)

22. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon transformation language. In: 1st International Conference on Model Transformations, pp. 46–60. Springer, Zürich, Switzerland (2008)
23. Lano, K., Clark, T., Kolahdouz-Rahimi, S.: A framework for model transformation verification. Formal Aspects of Computing 27(1), 193–235 (2014)
24. de Lara, J., Vangheluwe, H.: AToM^3: A Tool for Multi-formalism and Meta-modelling. In: 5th International Conference on Fundamental Approaches to Software Engineering. pp. 174–188. Springer, Grenoble, France (2002)
25. Leroy, X.: Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. SIGPLAN Notices 41(1), 42–54 (2006)
26. Meyer, B.: Applying design by contract. Computer 25(10), 40–51 (1992)
27. Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M.: Fully verifying transformation contracts for declarative ATL. In: 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 256–265. IEEE, Ottawa, ON (2015)
28. Picard, C., Matthes, R.: Coinductive graph representation: the problem of embedded lists. Electronic Communications of the EASST 39 (2011)
29. Pierce, B.C., de Amorim, A.A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Software Foundations. Electronic textbook (2017)
30. Poernomo, I., Terrell, J.: Correct-by-construction model transformations from partially ordered specifications in Coq. In: 12th International Conference on Formal Engineering Methods. pp. 56–73. Springer, Shanghai, China (2010)
31. Selim, G., Wang, S., Cordy, J., Dingel, J.: Model transformations for migrating legacy models: An industrial case study. In: 8th European Conference on Modelling Foundations and Applications. pp. 90–101. Springer, Lyngby, Denmark (2012)
32. Stenzel, K., Moebius, N., Reif, W.: Formal verification of QVT transformations for code generation. Software & Systems Modeling 14, 981–1002 (2015)
33. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance. pp. 446–453. VA, USA (2003)
34. Tisi, M., Cheng, Z.: Coqtl: An internal DSL for model transformation in coq. In: 11th International Conference on Model Transformation. pp. 142–156. Springer, Toulouse, France (2018)
35. Tisi, M., Perez, S.M., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: 14th International Conference on Model Driven Engineering Languages and Systems. pp. 32–46. ACM/IEEE, Wellington, New Zealand (2011)
36. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming 68(3), 214–234 (2007)
37. Varró, G., Varró, D., Friedl, K.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. In: 1st International Workshop on Graph and Model Transformations. pp. 191–205. Elsevier, Brighton, United Kingdom (2006)
38. Wagelaar, D.: Using ATL/EMFTVM for import/export of medical data. In: 2nd Software Development Automation Conference. Amsterdam, Netherlands (2014)
39. Werner, B.: Sets in types, types in sets. In: Proceedings of TACS'97. pp. 530–546. Springer-Verlag (1997)
40. Willink, E., Hoyos, H., Kolovos, D.: Yet another three QVT languages. In: 6th International Conference of Model Transformations. pp. 58–59. Springer, Budapest, Hungary (2013)
41. Yang, Z., Hu, K., Ma, D., Bodeveix, J.P., Pi, L., Talpin, J.P.: From AADL to timed abstract state machines: A verified model transformation. Journal of Systems and Software 93, 42 – 68 (2014)