CoqTL: an Internal DSL for Model Transformation in Coq

Massimo Tisi[1] and Zheng Cheng[2]

[1] IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France
massimo.tisi@imt-atlantique.fr
[2] Research Center INRIA Rennes - Bretagne Atlantique, Rennes, France
zheng.cheng@inria.fr

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 deep 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 reuse of native Coq libraries in the transformation definition and proof. In this paper we introduce CoqTL, we evaluate its practical applicability on a case study, and identify its limitations.

Keywords: Model-Driven Engineering, Model Transformation, 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 [20], medical data processing [22], aviation [2]), 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 [4,3,6,16]. 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. [15,10]). While not strictly needed for understanding this paper, we refer the reader to [18] 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 [6,1].

Coq includes Gallina, a functional programming language with pattern matching and rich type system, well suited as a platform for embedding domain-specific programming languages (DSLs) (e.g. [7]). 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 [12] (hence it should be familiar also to users of other rule-based MT languages, like

ETL [13], or RubyTL [8]). Thus, CoqTL aims to lighten the cognitive load of MT developers trying to build certified MTs in Coq.
– We design and implement a transformation engine in Coq that interprets programs written in CoqTL to transform models (Section 3.3). 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 an extensive use of the Coq Notation mechanism) so that any Coq development environment is able to support the domain-specific CoqTL syntax without requiring ad-hoc modifications.
– We show the practical applicability of CoqTL, by using it to specify a sample transformation, prove non-trivial contracts over it and automatically extract a certified implementation.

We make CoqTL publicly available as open source[3]. The repository contains also the example and proofs described in this paper.
Paper organization. We motivate our work by a sample transformation in Section 2. Section 3 illustrates the design of CoqTL in detail. In Section 4 we prove theorems on a CoqTL specification. Section 5 compares our work with related research, and Section 6 draws conclusions and lines for future work.

2 Class to Relational in CoqTL

We consider a very simplified version of the transformation from class diagrams to relational schemas (arguably, the Hello World transformation in the MT community). The example is intentionally very small, 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.

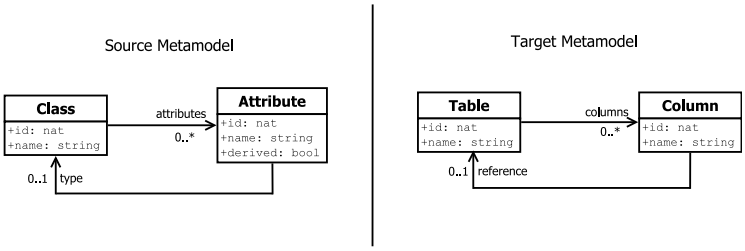

Fig. 1. A simplified structural metamodel for class diagrams (left), and relational schemas (right)

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.

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

```
1  Definition Class2Relational :=
2    transformation from ClassMetamodel to RelationalMetamodel
3      with m as ClassModel := [
4
5      rule Class2Table
6        from
7          element c class Class from ClassMetamodel
8            when true
9        to
10         [
11           output "tab"
12             element t class Table from RelationalMetamodel :=
13               BuildTable (getClassId c) (getClassName c)
14             links
15               [
16                reference TableColumns from RelationalMetamodel :=
17                  attrs ← getClassAttributes c m;
18                  cols ← resolveAll (match Class2Relational m) "col" Column (singletons attrs);
19                  return BuildTableColumns t cols
20               ]
21         ];
22
23     rule Attribute2Column
24       from
25         element a class Attribute from ClassMetamodel
26           when (negb (getAttributeDerived a))
27       to
28         [
29           output "col"
30             element c class Column from RelationalMetamodel :=
31               BuildColumn (getAttributeId a) (getAttributeName a)
32             links
33               [
34                 reference ColumnReference from RelationalMetamodel :=
35                   cl ← getAttributeType a m;
36                   tb ← resolve (match Class2Relational m) "tab" Table [cl];
37                   return BuildColumnReference c tb
38               ]
39         ]
40   ].
```

Listing 1.1. Class2Relational model transformation in CoqTL

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 Figure 1. *Tables* contain *Columns*, *Columns* can *refer* to other *Tables* in case of foreign keys.

In Listing 1.1 we use the CoqTL language to specify how to transform class diagrams to relational schemas. 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 13). The *links* section contains standard Gallina code to instantiate links related to the previous element (lines 17-19).

For instance in the *Class2Table* rule, once a class c is matched (lines 6 to 8), we specify that a table should be constructed by the constructor *BuildTable*, with the same *id* and *name* of c (line 13). While the body of the *element* section (line 13) can contain any Gallina code, it is type-checked against the *element* signature (line 12), 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 17), resolve them to their corresponding *Columns*, generated by any other rule (line 18), and construct new set of links connecting the table and these columns (line 19). While this is standard Gallina code, we use for this example an imperative style with a monadic notation ($\leftarrow$, similar to the do-notation in Haskell) that makes the code more clear in this case[4]. 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 *link* signature (i.e. in this case they must have type *TableColumns*, as specified at line 16).

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 and identifier of the *Attribute*. 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.

CoqTL naturally enables deductive verification of model transformations. Users can write Coq theorems that apply pre/postconditions (correctness conditions) to the model transformation. For example, Listing 1.2 defines a theorem stating that if all elements contained by 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 theo-

[4] the intuitive semantics of $\leftarrow$ 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 i)>0).
```

Listing 1.2. Name definedness theorem for the *Class2Relational* transformation

rem in Coq takes 56 lines of routine proof code (this short proof can be even automated by using modern automatic theorem provers [3,6]).

To illustrate more complex theorems we want to 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 addressed 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 model transformation in Coq. In this section we will describe the three 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) A transformation engine interprets transformation specifications against input models.

3.1 Metamodels and Models

Our encoding of metamodels in Coq is similar to analogous encodings in related work, based on inductive data types. As an example, Listing 1.3 shows the basic

definitions for encoding the *Relational* metamodel of Fig. 1. Since this interface is the main means to access source and target models, we aim at providing the simplest native representation.

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 multivalued 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 a 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* (for simplicity here we omit the definition of sum types, that relies on dependent types). The *RelationalModel* type is obtained by parametrizing *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 37-44), reference accessors need to pass through the list of links to find the ones connected to the element in parameter. Thus, reference accessors need to have the whole model as extra parameter (e.g., *getTableColumns* in the listing).

Listing 1.3 shows 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 casting a generic element to/from a specific metaclass. Similar functions are available for links.

While our representation allows us to encode any model instance, in our current prototype we do not directly implement several features that are found in modeling frameworks like EMF. Bidirectional references currently have no special treatment: both sides are encoded as separate references, that need to be separately assigned in the transformation code. No direct 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. Constraints for reference multiplicity or strong containment can only be encoded via extra pre/postconditions. Finally, differently from EMF, identifiers are considered as normal attributes and elements are considered equal when all their attributes are.

Automatic translators to/from EMF are still under development, and only partial implementations are provided on the CoqTL website.

```
(*** 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 1.3. Some basic definitions for the *Relational* models in Coq

3.2 Transformation Specification

Grammar 1.1 describes the concrete syntax of CoqTL. With respect to what we already discussed in Section 2, the grammar shows that CoqTL supports patterns with multiple input and output pattern elements. As indicated by the *header* production rule, CoqTL currently supports only transformations from a single source model to a single target model.

⟨*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*⟩ '`to`' ⟨*output-pattern*⟩

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

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

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

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

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

⟨*output-elem*⟩ ::= '`output`' ⟨*string*⟩ '`element`' ⟨*elem-def*⟩ '`links`' '`[`' ⟨*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*⟩ ::= '`reference`' ⟨*id*⟩ '`from`' ⟨*id*⟩

Grammar 1.1. The concrete syntax of the CoqTL language

The way we implement the concrete syntax of CoqTL relies on the Notation facility of Coq. A notation is a symbolic abbreviation to denote some expressions, 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 1.4 implements the production rules *link-def* and *link-decl* in Grammar 1.1. After the declaration of this notation, when the expression on the left-hand-side is matched, it is expanded in memory 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).

```
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 1.4. A few notations for CoqTL

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 1.4, 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 input (line 9).

The constructors used in our notations, like *BuildOutputPatternLinkDefinition* in Listing 1.4 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).

Gallina has several characteristics that make it suitable as an expression language for CoqTL. It is:

- Expressive. Gallina is based on a formal language called the Calculus of Inductive Constructions, combining a higher-order logic and a richly-typed functional programming language.
- Easy to learn. In our experience, the learning curve of the language is low if the user had some exposure to functional languages.
- Accompanied by sophisticated libraries. Reusing functions in those libraries during the MT specification is also important for the proof phase, that can exploit the theorems and lemmas provided by the library for those functions.

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 multivalued version, *resolveAll*) for element resolution. As illustrated at lines 18 and 36 in Listing 1.1, its signature requires the following arguments: 1) the result of the matching phase of the current transformation (*match Class2Relational m*), 2) the label associated to the required output element, useful for rules with multiple output elements (*"col"*), 3) the type of the expected result, useful for type checking (*Column*), 4) the source pattern to resolve (or the list of source patterns in case of *resolveAll*). Notice

that the matching phase is provided as a new application of the transformation in a specific *match* mode. While this choice affects the global efficiency of the transformation, it simplifies the development of proofs, because it avoids having a concept of transformation traces as side effects of the transformation execution.

While the expressiveness of CoqTL has important limitations, we are currently able to manually translate to CoqTL a significant subset of ATL transformations: 1-to-1 model transformations, in standard (non-refining) mode, written in the declarative subset of ATL, without lazy rules.

3.3 Transformation Engine

Algorithm 1 illustrates in pseudocode how the transformation specifications are interpreted by our transformation engine. This algorithm has been influenced by the execution algorithm of ATL [12] (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 1.2) 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.

Next, the engine iterates on each potential pattern instance p, and seeks for a rule r in R that matches it (i.e. if model elements in the pattern instance have the types defined in the input pattern of the rule) and satisfies the guard of that rule. If a rule r is found for the pattern instance p, then the *instantiation* phase of r will be invoked to construct the corresponding output elements of p and add them to the output model. Finally the *apply* phase is invoked, i.e. to construct the corresponding output links and add them to the output model.

Algorithm 1 Algorithm of the *execute* function

```
1: m ← maxArity(R)
2: P ← allPatterns(I_e, m)
3: for each p ∈ P do
4:     r ← findRule(R, p)
5:     if r ≠ None then
6:         O_e ← O_e ∪ instantiate(r, p)
7:         O_l ← O_l ∪ apply(r, p)
8:     end if
9: end for
```

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* or *resolveAll* functions, whose evaluation requires the execution of the *instantiate* phase. As mentioned in the previous section, in our solution the user passes to *resolve* the result of the transformation execution in *match* mode (i.e. *match* function at (lines 18 and 36 in Listing 1.1). The algorithm implemented in *match* is identical to Algorithm 1, without line 7 (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.

Finally the application of the transformation by the *execute* function can be automatically *extracted* by Coq into a separate executable program in several languages (e.g., OCaml, Haskell).

4 Proving theorems with CoqTL

In this section we show that CoqTL can enable practical verification for MTs. We formulate 4 theorem proofs over the model transformation 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 theorems 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 1.4), and *quantify over them*. Moreover we use the reflective model API mentioned in Section 3.1 to reason on metamodel-agnostic properties.

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

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 1.2, separately for all 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.

5 Related Work

There are many automatic theorem proving approaches for MTs (e.g. [4,3,6,16]). 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 model transformation, i.e. from AADL to TASM language, is semantic preserving [23]. 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 [5]. In [21], 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 [14]. 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 [11]. 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 [19]. 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 [9]. 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 [17]).

6 Conclusion

In conclusion, we present CoqTL, to our knowledge the first DSL in Coq for MTs and their verification. CoqTL is both functional and declarative in style, providing a familiar environment for transformation developers in Coq. Its underlining transformation engine, implemented in Coq, allows CoqTL programs to be interpreted against input models to compute output models. We show the practical applicability of CoqTL, by proving non-trivial contracts over a sample transformation.

Our future work would focus on the issues we identified in different points of our discussion. We want to develop a theorem library on top of CoqTL to facilitate MT verification, including of 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. We want to improve interoperability between CoqTL and common MDE tools such as EMF, for industry readiness.

Acknowledgements. We thank Rémi Douence for his valuable help during the development of CoqTL.

References

1. Ab.Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software & Systems Modeling 14(2), 1003–1028 (2015)
2. 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)
3. 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)
4. 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)
5. 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)

6. 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)
7. 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)
8. 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)
9. 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)
10. 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)
11. 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)
12. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
13. 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)
14. Lano, K., Clark, T., Kolahdouz-Rahimi, S.: A framework for model transformation verification. Formal Aspects of Computing 27(1), 193–235 (2014)
15. Leroy, X.: Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. SIGPLAN Notices 41(1), 42–54 (2006)
16. 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)
17. Picard, C., Matthes, R.: Coinductive graph representation: the problem of embedded lists. Electronic Communications of the EASST 39 (2011)
18. Pierce, B.C., de Amorim, A.A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Software Foundations. Electronic textbook (2017)
19. 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)
20. 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)
21. Stenzel, K., Moebius, N., Reif, W.: Formal verification of QVT transformations for code generation. Software & Systems Modeling 14, 9811002 (2015)
22. Wagelaar, D.: Using ATL/EMFTVM for import/export of medical data. In: 2nd Software Development Automation Conference. Amsterdam, Netherlands (2014)
23. 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)