Certifying a Rule-Based Model Transformation Engine for Proof Preservation

Zheng Cheng, Massimo Tisi, Joachim Hotonnier

Certifying a Rule-Based Model Transformation Engine for Proof Preservation

Zheng Cheng
LORIA (CNRS, INRIA)
Université de Lorraine
Nancy, France
zheng.cheng@inria.fr

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

Joachim Hotonnier
LS2N (UMR CNRS 6004)
IMT Atlantique
Nantes, France
joachim.hotonnier@imt-atlantique.fr

ABSTRACT

Executable engines for relational model-transformation languages evolve continuously because of language extension, performance improvement and bug fixes. While new versions generally change the engine semantics, end-users expect to get backward-compatibility guarantees, so that existing transformations do not need to be adapted at every engine update.

The CoqTL model-transformation language allows users to define model transformations, theorems on their behavior and machine-checked proofs of these theorems in Coq. Backward-compatibility for CoqTL involves also the preservation of these proofs. However, proof preservation is challenging, as proofs are easily broken even by small refactorings of the code they verify.

In this paper we present the solution we designed for the evolution of CoqTL, and by extension, of rule-based transformation engines. We provide a deep specification of the transformation engine, including a set of theorems that must hold against the engine implementation. Then, at each milestone in the engine development, we certify the new version of the engine against this specification, by providing proofs of the impacted theorems. The certification formally guarantees end-users that all the proofs they write using the provided theorems will be preserved through engine updates.

We illustrate the structure of the deep specification theorems, we produce a machine-checked certification of three versions of CoqTL against it, and we show examples of user theorems that leverage this specification and are thus preserved through the updates.

CCS CONCEPTS

• **Theory of computation → Logic**; • **Software and its engineering → Software notations and tools**.

ACM Reference Format:
Zheng Cheng, Massimo Tisi, and Joachim Hotonnier. 2020. Certifying a Rule-Based Model Transformation Engine for Proof Preservation . In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410949

1 INTRODUCTION

Model-driven engineering (MDE), i.e. software engineering centered on software models and model transformations (MTs), is widely recognized as an effective way to manage the complexity of software development. While MTs are often written in general-purpose programming languages, rule-based MT (RMT) languages are characterized by a compact execution semantics, that simplifies reasoning and analysis on the transformation properties. When RMTs are used in critical scenarios (e.g. in the automotive industry [31], medical data processing [37], aviation [6]), this analysis is crucial to guarantee that the MT will not generate faulty models.

In previous work, Tisi and Cheng presented CoqTL, a RMT language implemented as an internal DSL in the Coq interactive theorem prover [33]. CoqTL allows users to define model transformations, theorems on their behavior and machine-checked proofs of these theorems in Coq. CoqTL is designed to be used for highly-critical transformations, since developing formal proofs typically demands a considerable effort by users.

As any other piece of software, the CoqTL execution engine is subject to unpredictable changes because of bug fixes, performance improvements or the addition of new features. As we witnessed during the lifetime of other transformation engines [2, 25, 34, 38], this can also lead to several forks of the engine with significant differences in semantics[1]. Subsequent versions of transformation engines typically provide some guarantees of backward compatibility, so that users do not have to rewrite their MTs to exploit the features of the new version. This is typically achieved by defining a (more or less formal) behavioral interface that the engine developers commit to respect through the updates.

While this mechanism has been effective for transformation code, preserving proof code through transformation engine updates is a more challenging task. When proving the properties of a given function, proof steps depend on the exact instructions of that function. For example, a refactoring of the function code, that preserves its global semantics, may easily break the proof.

The importance of proof preservation is also amplified by the elevated cost of proof adaptation. Proofs in theorem provers like Coq can be seen as imperative programs that manipulate a complex state, i.e. the proof state. Any update to the underlying definitions can change the proof state at some point, and this change generally propagates to the rest of the proof from that point on. The result is that in general proof adaptations are not localized to easily identifiable steps.

To address these issues we propose a *deep specification* of the CoqTL engine that consists of a set of signatures for internal functions of the CoqTL engine, plus a set of lemmas that must hold on the implementation of these functions. A deep specification, as recently defined by Pierce [1], is a specification that is 1) formal, 2) rich (describing all the behavior of interest), 3) live (connected via machine-checked proofs to the implementation), and 4) two-sided (connected to both implementations and clients). The two-sided aspect is key:

[1]For instance, differences between ATL versions are documented at https://wiki.eclipse.org/ATL/VM_Comparison

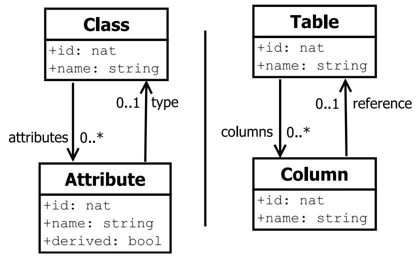

Figure 1: Minimal metamodels for class diagrams (left), and relational schemas (right).

(1) Engine developers certify every new major version of CoqTL against the deep specification by assigning concrete implementations of the required types and functions, and formally proving that they satisfy the required lemmas.
(2) Users leverage the deep specification as a library for proving theorems over their transformations. Any user-written proof that relies on the specification lemmas (instead of directly on the engine implementation), holds for any CoqTL version that is certified against the specification.

The structure of functions and lemmas that compose the deep specification of CoqTL are the central contribution of this paper. While our main motivation has been proof preservation, the lemmas are a useful artifact for documenting the engine behavior, and the certification process guarantees the absence of regression bugs during the engine lifetime. Moreover, we believe that our way of structuring specification and proofs for CoqTL can be adapted to other RMT languages as well (e.g. ATL [21], QVT [27], ETL [22]), with the purpose of interfacing them with theorem provers.

The paper is structured as follows. In Section 2 we define a minimal transformation and theorem that we will use to exemplify the rest of the paper. Section 3 illustrates our deep specification for CoqTL. The rest of the paper both illustrates the application of the specification and validates the feasibility of the approach. In Section 4 we isolate two updates of the CoqTL engine, and we show how subsequent versions are certified against the specification. In Section 5 we show how user proofs can be written relying on the specification, and how this guarantees their preservation through updates. Section 6 compares our work with related research, and Section 7 draws conclusions and lines for future work.

2 RUNNING EXAMPLE

As a sample transformation, we consider a very simplified version of the transformation from class diagrams to relational schemas. 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.

The left part of Fig. 1 shows the simplified structural metamodel of class diagrams. Each class diagram contains a list of named classes, each class contains a list of named and typed attributes. Classes and attributes have 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,

```
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 (getClassId c) (getClassName c)
12         with [
13           ref TableColumns :=
14             attrs ← getClassAttributes c m;
15             cols ← resolveAll Class2Relational m “col” Column
16               (singletons attrs);
17             return BuildTableColumns t cols
18         ]
19     ];
20
21   rule Attribute2Column
22     from
23       a class Attribute
24         when (negb (getAttributeDerived a))
25     to [
26       “col” :
27         c class Column :=
28           BuildColumn (getAttributeId a) (getAttributeName a)
29         with [
30           ref ColumnReference :=
31             cl ← getAttributeType a m;
32             tb ← resolve Class2Relational m “tab” Table [cl];
33             return BuildColumnReference c tb
34         ]
35     ]
36 ].
```

Listing 1: Simplified Class2Relational in CoqTL

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.

Listing 1 demonstrates how to encode the transformation in CoqTL. CoqTL is an internal DSL for RMT within the Coq theorem prover. The transformation primitives are newly-defined keywords (by the notation definition mechanism of Coq), while all expressions are written in Gallina, the functional language used in Coq. The CoqTL semantics is heavily influenced by ATL [20] (notably in the distinction between a match/instantiate and an apply function), and its original design choices are due to its focus on simplifying proof development.

In Listing 1, we declare that a transformation named `Class2-Relational` 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

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

Listing 2: Name definedness theorem for the `Class2Relational` transformation

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 outgoing from the new element (lines 14-17).

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 the same `id` and `name` of `c` (line 11). While the body of the `element` section (line 11) can contain any Gallina code, it is type-checked against the `element` signature (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 16), and construct new set of links connecting the table and these columns (line 17). This is standard Gallina code, where we use an imperative style with a monadic notation (⟵, similar to the do-notation in Haskell) that makes the code more clear in this case[2]. 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 is type-checked against the `link` signature (i.e. in this case the generated links must have type `TableColumns`, as specified at line 13).

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.

One major benefit of the CoqTL language is that it naturally enables deductive verification for the RMT under development. Users can write Coq theorems that apply pre/postconditions (correctness conditions) to the model transformation.

[2]the intuitive semantics of ⟵ 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`

For example, Listing 2 defines a theorem stating that if all elements contained by the input model have not-empty `names`, by executing the `Class2Relational` MT (by the function `execute`), all generated elements in the output model will also have not-empty `names`. Interactively proving this simple theorem in Coq takes 91 lines of routine proof code (this short proof can be even automated by using modern automatic theorem provers [8, 11]). Proofs on CoqTL may be much more demanding. For instance, proving that a complex CoqTL transformation preserves node unreachability needed more than a thousand lines of proof code in [33].

To give an idea on how a proof proceeds in Coq we present in Listing 3 the first steps of one of the possible proofs of Theorem `tables_name_defined` (each step is followed by the crucial part of the resulting proof state in comment):

- The `intros` tactic (line 2) extracts universal quantifiers and premises of implications from the proof goal (the theorem's property), and transforms each of them into new hypotheses. In the following proof state (lines 3 - 4) we show only two new hypotheses, **`H`** and **`H1`**, transformed from two premises. **`H`** says that the output model `rm` is the result of executing the `Class2Relational` transformation on the input model `cm`. **`H1`** says that the output table `t` is one of the elements in `rm`.
- The `rewrite` tactic (line 5) replaces `rm` in **`H1`** with the other side of the equality from **`H`**.
- The `simpl` tactic (line 8) tries to simplify **`H1`**. In order to search for simplifications, it implicitly replaces the call to the `execute` function with its body, and tries to simplify subexpressions. The implementation of `execute` (Listing 7) uses the `flat_map` function (Listing 4), that simply concatenates the results of the application of a given function to a list of elements. We can see that after the simplification, the call to `flat_map` appears in the resulting version of **`H1`** (line 9).
- The `apply` tactic (line 10) continues the proof by exploiting a property of the `flat_map` function, represented by the lemma `in_flat_map` (Listing 4). To apply the lemma, the tactic syntactically matches **`H1`** with the left-hand side of `in_flat_map`. If a match is found, **`H1`** is replaced with the right-hand side of `in_flat_map` (with the necessary substitutions). Then, the user analyzes the resulting state and continues the proof.

As any RMT engine, the CoqTL engine is bound to evolve, due to bug fixing, performance improvement or the addition of new features. Changes in the engine implementation may impact the semantics of the language primitives, and thus invalidate some proof steps [29]. The `apply` step in Listing 3 already shows a strong dependency on the implementation of the `execute` function, e.g. on the fact that it uses `flat_map`. Even trivial refactorings on the engine implementation can impact this dependency and break this step. For instance, if we update the implementation by replacing the call to `flat_map` with a completely equivalent code (e.g. a call to `concat(map ...)`), then the simplification of **`H1`** will not contain a call to `flat_map` anymore, the `in_flat_map` theorem will not match with **`H1`**, and line 10 would fail with error[3].

[3]In this particular case the proof could be simply adapted by the application of the lemma `flat_map_concat_map` in Listing 4

```
Proof.
  intros.
    (* H: rm = execute Class2Relational cm
       H1: In t (allModelElements rm) *)
  rewrite H in H1.
    (* H1: In t (allModelElements (execute
           Class2Relational cm)) *)
  simpl in H1.
    (* H1: In t (flat_map (...) (...) *)
  apply in_flat_map in H1.
    (* H1: ∃ sp : list ClassMetamodel_EObject,
           In sp (allTuples Class2Relational cm) ∧
           In t (toList (instantiatePattern
           Class2Relational cm sp)) *)
  ...
```

Listing 3: First steps of a proof for tables_name_defined

```
Fixpoint flat_map (f: A → list B) (l: list A) :=
  match l with
    | nil ⇒ nil
    | cons x t ⇒ app (f x) (flat_map t)
  end.

Lemma in_flat_map :
  ∀ (A B : Type) (f : A → list B) (l : list A) (y : B),
    In y (flat_map f l) ↔ (∃ x : A, In x l ∧ In y (f x)).

Lemma flat_map_concat_map :
  ∀ (A B : Type) (f : A → list B) (l : list A),
    flat_map f l = concat (map f l).
```

Listing 4: The `flat_map` function in the Coq standard library

3 DEEP SPECIFICATION FOR COQTL

We describe a deep specification for CoqTL, that we use on two sides: for engine certification in the next section, and as an interface for robust user proofs in Section 5.

Models and Metamodels. The deep specification that we introduce for CoqTL reuses the definition of models and metamodels in Coq from [15]. There, a model is defined by a Coq typeclass as a list of ModelElements and a list of ModelLinks.

```
Class Model := {
    modelElements : list ModelElement;
    modelLinks : list ModelLink;
}.
```

The concrete types for ModelElements and ModelLinks are defined in a metamodel specification, that is generated automatically from an Ecore metamodel. For instance, the relational metamodel is translated into the types shown in Listing 5. For each metamodel, `ModelElement` and `ModelLink` are respectively the sum-types of classes (e.g. `Table`, `Column`) and references (e.g. `TableColumns`, `ColumnReference`).

Abstract Syntax. The specification requires the CoqTL engine to define syntactic types for the elements of the CoqTL abstract syntax such as `Transformation`, `Rule`, `OutputPatternElement`, and `OutputPatternElementReference`. Engine developers need also to provide accessors that allow to navigate the syntactic structure of the transformation, e.g.:

```
getRules: Transformation → list Rule;
```

```
(* Concrete Types for ModelElements for Relational Model *)
Inductive Table : Set :=
  BuildTable :
    (* id *) nat →
    (* name *) string → Table.

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

(* Concrete Types for ModelLinks for Relational Model *)
Inductive TableColumns : Set :=
  BuildTableColumns:
    Table → list Columns → TableColumns.

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

Listing 5: Concrete types generated from the Relational Schema metamodel

Semantic functions. To enable reasoning on the behavior of CoqTL we propose a fine-grained decomposition of its semantics into a hierarchy of (pure and total) functions. CoqTL engines implement these functions. Users reference these functions in their theorems and proofs, to predicate on the desired behavior of their transformation. However they do not have access to the implementation of these functions in their proofs, but only to a set of lemmas defining the CoqTL behavior (discussed in the next subsection).

The full hierarchy is shown in Listing 6. Each function in the hierarchy can be obtained by composing its direct children functions according to the pattern noted between parent and child. This hierarchical way of structuring the specification of the MT engine is the first key point of our solution. In the following we briefly illustrate each function, while the exact semantics of the composition patterns will be discussed in the next subsection.

The first two arguments of each function represent the syntactic element that the function is executing (`Transformation`, `Rule`, etc.) and the source model that is being transformed.

The `execute` function, given a transformation, transforms the whole source model into a target model[4]. In the specification, `execute` is obtained by considering all the possible tuples of model elements in the source model, `filter`ing them using the `matchPattern` function, and concatenating (`flat_map`) the result of `instantiatePattern` and `applyPattern` on each tuple.

`matchPattern` returns the rules in the given transformation that match the given source pattern (list of `SourceModelElement`). The result is obtained by iterating on all the rules and `filter`ing the ones that match the given pattern, by the function `matchRuleOnPattern`. `matchRuleOnPattern` checks that the guard evaluates to `true` for the given pattern.

`instantiatePattern` generates target elements by transforming only the given source pattern. The `option` type (here and in the following) represents the possibility to return an error value, in case, e.g., that the source pattern does not match the rule in

[4]CoqTL transformations have one source and one target model. Multiple source and target models can still be transformed by pre-computing union models.

```
execute: Transformation → SourceModel → TargetModel;
∟ (* filter *)
  matchPattern: Transformation → SourceModel → list SourceModelElement → list Rule;
  ∟ (* filter *)
    matchRuleOnPattern: Rule → SourceModel → list SourceModelElement → option bool;
∟ (* flat_map *)
  instantiatePattern: Transformation → SourceModel → list SourceModelElement → option (list TargetModelElement);
  ∟ (* flat_map *)
    instantiateRuleOnPattern: Rule → SourceModel → list SourceModelElement → option (list TargetModelElement);
    ∟ (* flat_map *)
      instantiateIterationOnPattern: Rule → SourceModel → list SourceModelElement → nat → option (list TargetModelElement);
      ∟ (* map *)
        instantiateElementOnPattern: OutputPatternElement → SourceModel → list SourceModelElement → nat → option TargetModelElement;
∟ (* flat_map *)
  applyPattern: Transformation → SourceModel → list SourceModelElement → option (list TargetModelLink);
  ∟ (* flat_map *)
    applyRuleOnPattern: Rule → SourceModel → list SourceModelElement → option (list TargetModelLink);
    ∟ (* flat_map *)
      applyIterationOnPattern: Rule → SourceModel → list SourceModelElement → nat → option (list TargetModelLink);
      ∟ (* flat_map *)
        applyElementOnPattern: OutputPatternElement → SourceModel → list SourceModelElement → nat → option (list TargetModelLink);
        ∟ (* map *)
          applyReferenceOnPattern: OutputPatternElementReference → SourceModel → list SourceModelElement → nat → option TargetModelLink;

resolveAll: Transformation → SourceModel → string → list (list SourceModelElement)) → nat → option (list TargetModelElement);
∟ (* flat_map *)
  resolve: Transformation → SourceModel → string → list SourceModelElement → nat → option TargetModelElement;
```

Listing 6: Hierarchy of semantic functions

the first place. instantiatePattern is obtained iterating on the matching rules and concatenating (flat_map) the result of instantiateRuleOnPattern on the rules.

instantiateRuleOnPattern generates target elements by transforming the source pattern using only the given rule. Since CoqTL supports iterative rules (executed once for every value of a given iterator [15]), instantiateRuleOnPattern is obtained by concatenating the results of instantiateIterationOnPattern for every iteration. In the same way instantiateIterationOnPattern is obtained by concatenating the result of the creation of each output element in the rule, by instantiateElementOnPattern.

applyPattern, applyRuleOnPattern, applyIterationOnPattern and applyElementOnPattern are analogous to the corresponding instantiation functions, but they generate the target links connecting target elements. applyReferenceOnPattern generates a single link in the output pattern. The separation between functions that generate elements and functions that generate links is inspired by ATL [21].

The resolveAll function is not used directly by the engine, but is provided to the user to resolve a given list of patterns (list (list SourceModelElement)). It requires the users to specify also the label of the OutputPatternElement they are referring to, as a string, and the iteration number for iterative rules, as a natural.

resolveAll is the simple composition by flat_map of calls to the resolve function, that given a single source element returns the corresponding target element. Note that the functional specification does not mention any concept of transformation traces, that is very common in engines for RMT languages. While traces can indeed be used to improve performance of the actual implementation of the specification, they are not necessary to provide a simple definition of its input-output semantics.

Lemmas overview. The signatures introduced in the previous subsection define the structure of the specification and the types used by each function. Implicitly they also state, for each function, that same argument values must always return the same output values (functionality).

In this subsection we define the formal semantics of these functions as lemmas that the engine will need to certify against. We distinguish three kinds of lemmas:

- **Membership lemmas.** For each function we define lemmas that characterize the necessary and sufficient conditions for an element to belong to the output of that function. We define these conditions by *predicating on the relation of the function with its children functions*. Hence for each relation between functions in Listing 6 we define a lemma capturing the meaning of the relation. This is the second key point of the specification's structure.
- **Leaf lemmas.** For leaf functions of the trees in Listing 6 we define specific lemmas defining their intended behavior.
- **Error lemmas.** For each function we define lemmas that list reasons for error states of the function.

The full specification contains 48 lemmas defining the semantic functions behavior. Because of space constraints we here show only few examples. We refer the reader to our online repository for the full specification[5].

Membership lemmas. As shown in Listing 6 relationships between parent and child function fit in three categories, filter, flat_map and map. To formally capture the meaning of the relationship, we define for each one of these categories a template lemma and we instantiate it for each function pair.

[5]https://github.com/atlanmod/CoqTL/tree/flat-syntax-parametric

For instance, for lemmas in the `filter` category we use as a template the `filter_In` lemma of the Coq standard library:

```
Lemma filter_In : ∀ (A : Type) (f : A → bool)(x : A) (l : list A),
  In x (filter f l) ↔ In x l ∧ f x = true.
```

The lemma states that an element is included in the result of a filter if and only if it was included in the initial list in the first place, and the filtering function evaluates to `true` for that element. We show how this template lemma is instantiated for characterizing the `matchPattern` function:

```
Lemma matchPattern_In :
  ∀ (tr: Transformation) (sm : SourceModel),
∀ (sp : list SourceModelElement)( r : Rule),
  In r (matchPattern tr sm sp) ↔
    In r (getRules tr) ∧
    matchRuleOnPattern r tr sm sp = Some true;
```

The lemma states that a rule appears in the result of a `matchPattern` if and only if the rule is included in the list of rules of the transformation, and the `matchRuleOnPattern` function returns true for that rule.

Instead, to characterize a `flat_map` relation, we instantiate the template lemma `in_flat_map`, shown already in Listing 4. For instance, to characterize the `flat_map` relation between `execute` and `instantiatePattern`, we specialize `in_flat_map` and produce the following lemma:

```
Lemma execute_In_elements :
  ∀ (tr: Transformation) (sm : SourceModel) (te : TargetModelElement),
    In te (allModelElements (execute tr sm)) ↔
    (∃ (sp : list SourceModelElement) (tp : list TargetModelElement),
        incl sp (allModelElements sm) ∧
        instantiatePattern tr sm sp = Some tp ∧
        In te tp);
```

The lemma states that an element `te` is included in the model elements of the result of `execute` if and only if we can find a source pattern `sp` in the source model and a target pattern `tp` that include `te`, and the application of `instantiatePattern` to `sp` returns `tp`. An analogous lemma `execute_In_links` is defined for the relation of `execute` and `applyPattern` and so on.

Lemmas in the map category are similarly produced, by specializing the lemma `in_map_iff` of the standard library to the `applyElementOnPattern` and `instantiateIterationOnPattern` functions:

```
Lemma in_map_iff : ∀ (A B : Type) (f : A → B)(l : list A) (y : B),
  In y (map f l) ↔ (∃ x : A, f x = y ∧ In x l).
```

Leaf lemmas We introduce specific lemmas to specify the semantics of leaf functions in Listing 6 .

The functions `matchRuleOnPattern`, `instantiateElementOnPattern`, `applyReferenceOnPattern` have similar semantics: they all simply consist of the evaluation of a Gallina expression (respectively the `guard`, the `OutputElement` definition and the `OutputLink` definition) embedded in the CoqTL transformation. Hence, their semantics is expressed by a simple lemma like the following:

```
Lemma tr_matchRuleOnPattern :
  ∀ (r: Rule) (sm : SourceModel) (sp: list SourceModelElement),
  matchRuleOnPattern r sm sp = evalExpression (getGuardExp r) sm sp.
```

The lemma states that execution of `matchRuleOnPattern` for a certain rule coincides with the evaluation of the guard expression for that rule. `evalExpression` is a generic function provided by CoqTL to execute a given Gallina expression, checking that types are correct.

Finally we specify the semantics of the `resolve` function with the following lemma[6]:

```
Lemma tr_resolve:
∀ (tr: Transformation) (sm : SourceModel) (name: string)
  (sp: list SourceModelElement) (iter: nat) (te: TargetModelElement),
  resolve tr sm name type sp iter = Some te →
   (∃ (r: Rule) (o: OutputPatternElement),
   In r (getRules tr) ∧ In o (getOutputPattern r) ∧ beq name (getName o)
    ∧ (instantiateElementOnPattern o sm sp iter = Some te)).
```

The lemma states that if `resolve` returns an element `te`, then it means that 1) it found a rule `r` and output pattern element `o`, whose name corresponds to the argument of the call to `resolve`, and 2) the instantiation of that output pattern element would produce `te`. As we anticipated in the previous subsection, resolution is not defined by traces, but (in a functional style) by a reference to the element instantiation function.

Error lemmas. Membership and leaf lemmas characterize completely the *presence* of elements or links in the result of the functions. In cases where the function does not return any element or link, we need to characterize if this is a normal behavior or it is the result of an error (represented by the value `None`). For each function we define lemmas that characterize the presence (`= None`) or the absence of errors (`<> None`).

For instance, the following lemma states that the `applyRuleOnPattern` function will return an error when the length of the source pattern is different than the number of input pattern elements expected by the rule:

```
Lemma applyRuleOnPattern_None :
    ∀ eng: TransformationEngine,
      ∀ (tr: Transformation) (sm : SourceModel) (r: Rule)
          (sp: list SourceModelElement),
        length sp <> length (getInTypes r) →
        applyRuleOnPattern r tr sm sp = None.
```

The following lemma states that the output of instantiatePattern is correct (`<> None`) if and only if it exists at least one rule that matches the pattern and does not return an error when instantiated on that pattern (`instantiateRuleOnPattern`).

```
Lemma instantiatePattern_Some :
∀ (tr: Transformation) (sm : SourceModel) (sp: list SourceModelElement),
 instantiatePattern tr sm sp <> None ↔
 (∃ (r: Rule),
     In r (matchPattern tr sm sp) ∧
     instantiateRuleOnPattern r tr sm sp <> None);
```

4 ENGINE CERTIFICATION

In this section we certify three versions of CoqTL against the specification described in the previous section. The machine-checked proofs aim at both validating the correctness of the specification and giving a qualitative measure of the certification effort.

[6]The lemma is modeled after `find_some` in the standard Coq library, since `resolve` essentially finds, for a given source pattern, the matching rule and corresponding target

```
Definition execute_a (tr: Transformation) (sm : SourceModel) :=
  Build_Model
  (flat_map (λ t ⇒ toList (instantiatePattern_a tr sm t)) (allTuples tr sm))
  (flat_map (λ t ⇒ toList (applyPattern_a tr sm t)) (allTuples tr sm)).

Definition execute_b (tr: Transformation) (sm : SourceModel) :=
  let matchedTuples := (filter (λ t ⇒ match (matchPattern tr sm t)
    with nil ⇒ false | _ ⇒ true end) (allTuples tr sm)) in
  Build_Model
  (flat_map (λ t ⇒ toList (instantiatePattern_b tr sm t)) matchedTuples)
  (flat_map (λ t ⇒ toList (applyPattern_b tr sm t)) matchedTuples).

Definition execute_c (tr: Transformation) (sm : SourceModel) :=
  let matchedTuples' :=
    map (λ t ⇒ (t, matchPattern tr sm t)) (allTuples tr sm) in
  Build_Model
    (flat_map (λ t ⇒ toList (instantiatePattern_c tr sm t)) matchedTuples')
    (flat_map (λ t ⇒ toList (applyPattern_c tr sm t)) matchedTuples').
```

Listing 7: Three versions of the execute function

4.1 The CoqTL Engine

The implementation of a RMT engine is much more complex than the specification presented in the previous section. To reach acceptable performance, engines usually employ optimized transformation algorithms, tracing mechanisms, lazy computation, caching and indexes. The specification also omits cross-cutting concerns, e.g. related to logging and error-handling. Technical aspects, like the generation of unique identifiers are not considered either.

To exemplify these implementation choices we isolate two small updates in the development history of CoqTL. We discuss the version of CoqTL before these updates and the two following versions. The respective git commits are marked as [41875ed], [118eefa] and [c7f6526] in the CoqTL repository[7]. For brevity, in the following we will refer to these version as $CoqTL_a$, $CoqTL_b$ and $CoqTL_c$.

Lines 1 - 4 in Listing 7 show the implementation of the execute function of $CoqTL_a$ (i.e., execute$_a$). The allTuples function computes all tuples of n elements from the source model sm, with n less or equal to the maximum length of input patterns among all the rules in the given transformation tr. Then, the instantiatePattern function is applied on each tuple t to produce output elements, and the applyPattern function is applied on each tuple to produce output links. The elements and links of the resulting model are the concatenation of the results of each instantiatePattern and applyPattern, respectively.

execute$_a$ is very simple but it has some evident inefficiencies. For instance, both instantiatePattern$_a$ and applyPattern$_a$ are applied to the whole list of possible tuples of input elements. This list has size $T = (1 - |sm|^{(ar+1)})/(1 - |sm|)$, with $|sm|$ number of elements of the source model, and ar maximum number of input elements of a rule in tr (maximum arity). Before generating anything, both instantiatePattern$_a$ and applyPattern$_a$ determine if the input tuple matches any rule. Hence, this check is performed $2 * T$ times.

The $CoqTL_b$ version improves on this point by replacing the function execute$_a$ with execute$_b$ (lines 6 - 11 in Listing 7). A new matching step filters the list of all tuples, to determine the list of tuples that match at least one rule (matchedTuples). Now, instantiatePattern$_b$ and applyPattern$_b$ can be applied only to the much smaller list of matchedTuples. Hence, $CoqTL_b$ matches a tuple $T + 2|matchedTuples|$ times, much less than $CoqTL_a$.

Finally, $CoqTL_c$ further improves on $CoqTL_b$ by: 1) storing the matched rules for every tuple in a map, during the matching step, and 2) passing this information to the functions instantiatePattern$_c$ and applyPattern$_c$ so they do not need to compute any more matches. This way, $CoqTL_c$ matches a tuple T times, further reducing over $CoqTL_b$.

The first row of Table 1 summarizes the size of the three versions of the CoqTL implementation, in terms of lines of Gallina code (LoC). We show (between parentheses) also the size of the updated code w.r.t. to the previous version. As we can see, the three implementations have similar size, since the updates only touch few lines in the execute, instantiatePattern and applyPattern functions.

[7]https://github.com/atlanmod/CoqTL

	$CoqTL_a$	$CoqTL_b$	$CoqTL_c$
Impl. (LoC)	436	437 (+5,-4)	448 (+34,-22)
Cert. (LoC)	2055	2095 (+41,-1)	2170 (+118,-3)
Cert./Impl. Ratio	4.71	4.79	4.84

Table 1: Size of the implementation and certification of the semantic functions (measurement based on Coq's built-in tool (coqwc), excluding comments, model/metamodel framework, generators)

4.2 Certifying CoqTL

The second row in Table 1 shows the certification effort across the three versions of CoqTL, in terms of number of proof steps. Between parenthesis we show the number of updated proof steps w.r.t. the certification of the previous version. The third row shows the ratio between certification and implementation.

Certifying $CoqTL_a$ against its deep specification takes 2055 LoC. This includes: a) providing witnesses for the required semantic functions defined in the deep specification of CoqTL and b) certifying the semantic functions against their membership, leaf and error lemmas. All certification proofs are manually developed, mechanically checked by Coq, and are publicly available on the paper website. The global size of the proofs denotes the significant effort required by the certification activity: proofs of specification lemmas need 4.71 times the LoC required for implementing the semantic functions. However, we have to note that proofs of lemmas in the same category have some similarities with each other. For example, all membership lemmas follows a similar induction principle and proof pattern. Also, proofs for membership lemmas based on the same template lemma (e.g. in_flat_map) usually leverage the template lemma to some extent.

Although the certification proofs for $CoqTL_b$ have similar size to $CoqTL_a$, the difference between their certification code is very small (41 new lines of new proof, one line removed). Indeed the purely functional nature of the specification, and our hierarchical organization of lemmas, induce a useful modularity property: if an update impacts a certain function, we need to rework the certification only of the lemmas related to that function and possibly its ancestors in Listing 6.

```
1   Proof.
2     intros.
3       (* H: rm = execute Class2Relational cm
4          H1: In t (allModelElements rm) *)
5     rewrite H in H1.
6       (* H1: In t (allModelElements (execute
7                Class2Relational cm)) *)
8     apply execute_In_elements in H1.
9       (* H1: ∃ (sp : list ClassMetamodel_EObject)
10               (tp : list RelationalMetamodel_EObject),
11               incl sp (allModelElements cm) ∧
12               instantiatePattern Class2Relational cm sp = return tp ∧
13               In t tp *)
14       ...
```

Listing 8: First steps of a proof for tables_name_defined that uses the specification lemmas

In particular, since $CoqTL_b$ updates only the `execute` function, then we need only to update the proof of the membership and error lemmas of `execute`, i.e. `execute_In_elements`, `execute_In_links`, `execute_Some` and `execute_None`. One way of performing this adaptation is proving a preservation lemma like:

```
Lemma execute_preserv : ∀ (tr: Transformation) (sm : SourceModel),
  execute_b tr sm = execute_a tr sm.
```

This lemma proves that the two versions of the `execute` function produce the same result for same transformation and source model. When this lemma is proved, we can prove any lemma on $\texttt{execute}_b$ by first rewriting $\texttt{execute}_b$ with $\texttt{execute}_a$ and then applying the lemma already proved for $\texttt{execute}_a$.

The third column of Table 1 demonstrates the proof engineering effort for certifying $CoqTL_c$. This update is larger, since it impacts three semantic functions, namely `execute`, `instantiatePattern` and `applyPattern`. Again one possible adaptation exploits preservation lemmas to prove that the three updated semantic functions did not change their global behavior. The resulting update to the proof code amounts to the addition of 118 LoC, and removal of 3 lines.

The experience shows that the adaptation effort for certification proofs is limited to the properties of the updated functions, and that it tends to grow with the size of the update.

5 USER PROOFS

5.1 Using the Specification in Proofs

Once the engine has been certified against the abstract specification, the lemmas of the specification become available in user proofs. This allows users to write more abstract proofs about transformations, without looking into a specific engine implementation.

To illustrate this, Listing 8 shows the first steps of the same proof shown in Listing 3, but adapted to use the specification lemmas. The first two steps, `intros` and `rewrite` are exactly the same. At this point, Listing 3 was letting Coq `simpl`ify **`H1`** by looking into the specific engine implementation in use. Then it was able to `apply` the standard lemma `in_flat_map`, a step dependent on that implementation. Instead, we continue the proof by relying on the abstract specification of `execute`. In particular, we directly `apply` the lemma `execute_In_elements` (line 8).

As we shown, `execute_In_elements` is a specialized version of `in_flat_map` for the `execute` function, so the global structure of the proof is not changed. However, the lemma `execute_In_elements` makes the proof more robust. Now the `apply` step is independent from the engine used so it does not need to be changed if we update the implementation of `execute`. For example, replacing `flat_map` with `concat(map ...)` in `execute` would break the proof in Listing 3, but it does not invalidate the proof in Listing 8.

Also, the final proof state in Listing 8 (lines 9 - 13) is equivalent to the one in Listing 3, but more abstract. In particular, the formula `incl sp (allModelElements cm)` in Listing 8 only mentions the accessor `allModelElements` of the model interface. The corresponding line `In sp (allTuples Class2Relational cm)` in Listing 3, despite being equivalent to the previous one for the current versions of CoqTL, depends on the concrete computation of all possible tuples for a particular transformation, encoded in the function `allTuples`.

5.2 Impact on Proof Effort

While relying on the RMT specification instead of the engine implementation does not change the global strategy of the proof, it may have an impact on the effort required from the user to correctly encode the proof in Coq.

The main drawback is that the specification is not computational (we mean executable), the implementation is. When referring to the engine implementation, users can simply ask Coq to compute the result of a sub-computation (e.g. the application of a single rule) during a proof step. Users can also apply standard Coq proof tactics that perform implicit computations during their processing, like `simpl` in Listing 3. When referring exclusively to the specification lemmas, users can not automatically compute parts of the transformation logic, and they need to explicitly `apply` a lemma for each sub-step of the transformation. Of course, this does not impacts the computability of Gallina expressions, so during the proof all the guard expressions, output pattern element expressions and output pattern link expressions, can be automatically computed from their inputs. However, this drawback has the global effect of increasing the size of the proof in general.

On the other hand, computation can only help in forward reasoning, i.e. it can produce the output of a function starting from its inputs. The specification lemmas instead are based on bi-conditionals, thus they can be used for forward or also backward reasoning, i.e. from knowledge on the output they can be used to derive knowledge on the input of the transformation step. Proofs that require this kind of reasoning are reduced by using specification lemmas.

Finally, proofs on the specification stay at the same level of abstraction, while proofs on the implementation may need to switch from an abstract view to a concrete one, and back. This can cause a reduction of proof steps, as it for instance can be seen by comparing Listing 3 with Listing 8.

To quantify the potential impact of the specification on the proof effort we perform an experimentation summarized in Table 2. In the experimentation we consider two transformations, `Class2Relational` from Listing 1 and `HSM2FSM` from [3, 8]. `HSM2FSM` is a transformation that performs a flattening algorithm for hierarchical state machines. The transformation requires 7 rules, for a total 205 LoC.

Theorem	Transf.	w/ Impl.	w/ Spec.	Var.
all_classes_match	C2R	19	26	+37%
all_classes_inst	C2R	19	25	+32%
concrete_attrs_inst	C2R	28	38	+36%
all_elems_inst	C2R	87	79	-9%
attr_info_preserv	C2R	88	125	+42%
rel_nm_def	C2R	117	127	+9%
rel_id_uniq	C2R	-	315	-
all_sm_match	HSM2FSM	19	26	+37%
all_sm_inst	HSM2FSM	19	25	+32%
regular_states_inst	HSM2FSM	42	38	-1%
all_states_inst	HSM2FSM	130	114	-12%
sm_nm_def	HSM2FSM	-	259	-

Table 2: Summary of the proof-effort experiment (measurement based on coqwc, excluding comments)

We consider theorems of different complexity on both transformations. For each theorem, we compare existing proofs using the engine implementation with new proofs that we produce using only the specification. Table 2 shows the name of the theorem, the transformation it predicates on, the size (number of proof steps) of the proofs that use the implementation, the size of the proofs that use the specification, and the percentage variation in size between the two types of proofs.

First of all we report that, as we expected, all proofs using the specification are preserved through the two updates. They are valid for all three certified CoqTL versions, with no adaptation required.

Moreover, the results show that using only the specification requires longer user proofs. In average, we see an increase of +19% LoC, but with large variability, and a maximum case that reaches +42%. This shows that the lack of computability has a major impact on proof size. In a few cases, when little automatic computation is used, we see a reduction of proof steps, up to -12%.

Note that an increase in proof size does not immediately translate to an increase in proof effort. The global proof strategy is the part the requires the most creativity and time from users, and is not impacted by the use of the specification. The extra steps show also a high degree of repetitiveness. We plan to exploit this observation for automation in future work.

Finally Table 2 includes also two user theorems that show the applicability of the specification in proofs for more complex theorems. They prove respectively the uniqueness of generated tables and columns (`rel_id_uniq`) and the definedness for all state names (`sm_nm_def`). We report that both proofs, respectively counting 315 and 259 LoC, preserve their validity through the CoqTL updates with no adaptation.

5.3 Discussion and Limitations

In this section we discuss several points, highlighted by the experimentation activity and result.

As an alternative to using our proposed stable interface, users can make proofs more robust to updates by other means, e.g. by a finer modularization of their user theorems in lemmas or by the development of ad-hoc automatic tactics. These techniques require a good degree of experience with the interactive theorem prover. Our proposal instead assigns the proof-preservation concern to the responsibility of engine developers.

We are following several leads to address the increase in proof length highlighted by the previous section. First of all we are augmenting the lemma library with derived lemmas proved by composition of existing lemmas. Derived lemmas do not increase the certification effort: they are proved only once and the engine developer only needs to certify against the basic lemmas. We also plan to provide RMT-specific proof tactics. Tactics are procedural applications of several proof steps at once. We plan to use the powerful Coq tactics language to perform some transformation computation steps by tactics that apply several specification lemmas at once. Another option we are considering is adding a reference implementation of the semantic functions, that would be usable for computation in user proofs. Note that this would not eliminate the need for specification lemmas, that give users important tools for backward reasoning (as discussed in the previous section). Also the equivalence of new versions of the engine with the reference implementation will have to be separately verified, strongly impacting the cost of certification.

The semantics defined by the specification naturally abstracts away some aspects of the implementation. For instance, membership lemmas only characterize the membership of an element to the list of results, they do not predicate about ordering of the produced elements (i.e. input and outputs are considered as `sets`). On the other side, implementation functions work with lists, processed in a deterministic order. The consequence is that some theorems (e.g. about element ordering) can be proved using the implementation, but not using the specification. Such proofs are not recommended, since they are not guaranteed to hold across versions of the engine.

Because of the similarities between CoqTL and ATL, the approach is applicable with few modifications to the ATL flavors. Other RMT languages can apply the specification structure to their case by adapting the hierarchy of Listing 6 to the behavior of their engine.

The actual signatures of the semantic functions in Coq contain some technical arguments that are omitted in Listing 6. In particular, CoqTL uses dependent types for performing static type checking, e.g. for checking that elements used as input for the match and output-pattern computation are the same. Also the metamodel interface has more complex meta-types enabling reflection over model elements. Finally, leaf semantic functions, that evaluate Gallina expressions, are currently provided as a runtime library. User proofs are currently allowed to unfold these functions and the Gallina expressions within, but in this case an hypothetical update to these functions would break the user proofs.

Besides enabling proof persistence, a stable deep specification is also an important improvement on the engine development process, since certification can guarantee the absence of regression bugs (on the part of the semantics that is included in the specification). The approach allows for cross-engine proofs. Often the engine lifeline forks into several different implementations with significant differences in semantics. For example, ATL has many forks for lazy, incremental, parallel semantics. User proofs hold for all engine forks that certify against the specification.

Finally, besides RMT verification, CoqTL can also be used as a platform for reasoning about transformations, e.g. for comparison, optimization, computing pre/post conditions. The deep specification allows users to perform these kinds of reasoning by abstracting from implementation details.

6 RELATED WORK

This work contributes to the area of proof engineering for model transformations. In this section we start highlighting the recent work on certified language implementation in Coq that influenced this work. Then we focus on related work on model transformation, for specification and theorem proving.

Certified language implementation in Coq. Several frameworks in literature are dedicated to the formal specification of language semantics, e.g. the K framework [30] where rewriting rules are used to define executable language semantics. Within the Coq community, the DeepSpec project [1] is a recent effort to build a network of deep specifications in Coq. The objective is to build fully certified software stacks by sharing specifications at each interface between tools making up the stack. Several language specifications are considered in the project. The list includes Leroy's CompCert [24], i.e. a Coq specification and verified optimizing compiler for a large subset of the C programming language, and the DataCert project working towards a certified SQL engine [4, 5]. In the same spirit, the JSCert project [7] provides a formal specification of Javascript in Coq and a reference certified interpreter. Furthermore, Chlipala et al. [16] propose a formalism in Coq to specify languages as libraries in a modular way by separating functionality and performance. This separation allows to expose only logical properties to the user and hide the optimization phases that derive an efficient implementation. Our proposal is strongly influenced by these works, and applies their principles to the RMT language paradigm.

Formal specification for model transformation languages. Several RMT languages are provided with a formal semantics. The OMG group gives a specification for QVT [27]. It is described in a mixture of natural language and semi-formal set-theoretic notations, which provides guidance on how to implement MT engines. Troya and Vallecillo give a detailed operational semantics for the ATL language in terms of rewriting logic using the Maude system [35]. The goal is to produce an alternative implementation of ATL in Maude. Varró et al. implement a graph transformation engine in relational databases [36]. They use relational algebra to algorithmically describe how a graph manipulation operator (e.g. delete an edge) can be implemented w.r.t. database operator(s). He and Hu gives a formal semantics of their putback-based bidirectional model transformation engine [18]. Full or partial specifications of RMT languages have been used to study properties of those languages. For instance, Hidaka et al. formally summarize the additivity property for MT engines [19]. It systematically characterizes how the addition or removal of input results in a corresponding addition or removal of parts of the output. Differently from these efforts, our aim is improving the engineering of proofs on RMTs. This strongly influences the shape of the specification. We provide an original functional decomposition of the internal engine behavior, and a library of lemmas that aims at producing stable proofs, without an excessive impact on proof effort.

Theorem proving for Model Transformations. Automatic theorem proving has attracted more attention than interactive theorem proving in the RMT community. Büttner et al. use Z3 to verify a declarative subset of the ATL and OCL contracts [8]. Their result is novel for providing minimal axioms that can verify the given OCL contracts. To understand the root of the unverified contracts, they demonstrate the UML2Alloy tool that draws on the Alloy model finder to generate counter examples [9]. Oakes et al. statically verify ATL MTs by symbolic execution using DSLTrans [26]. This approach enumerates all the possible states of the ATL transformation. If a rule is the root of a fault, all the states that involve the rule are reported. Cheng and Tisi address usability aspects of automatic theorem proving for RMT, e.g. fault localization [12], scability [14] and incrementality [13]. However, interactive theorem proving has shown to be necessary for certifying RMTs for complex properties. 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 [10]. 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. In [28], Poernomo et al. use Coq to specify MTs as proofs and take advantage of the Curry-Howard isomorphism to synthesize provably correct MTs from those proofs. The approach is further extended by Fernández and Terrell on using co-inductive types to encode bi-directional or circular references [17]. None of these works addresses explicitly the modularity of the specification for certifying a transformation engine, and for proof preservation through engine updates.

7 CONCLUSION AND FUTURE WORK

The main contribution of this paper is the design of a deep specification for CoqTL. This specification is made of a hierarchy of semantic functions and several lemmas about their behaviors. We validate the specification by certifying three versions of CoqTL against it. Our experiments show that using this interface often makes the user proofs longer. However all proofs written exclusively using the specification have the crucial advantage to be preserved across engine updates. We believe that the structure of this specification can be adapted to other RMT engines, and used to organize their current or future interface with interactive theorem provers.

In current work we are exploiting the regular structure of the specification, in order to design automatic proof tactics. By applying chains of lemmas in a single step, tactics could be an effective replacement for the RMT computation steps of engine-dependent proofs. This would reduce length and effort for stable proofs. In general, we believe that this line of work would enable RMT languages to become an effective tool to express and verify steps (e.g. of code generation, program transformation, compilation) within fully-certified stacks.

REFERENCES

[1] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017). https://doi.org/10.1098/rsta.2016.0331

[2] ATLAS Group. 2005. *Specification of the ATL Virtual Machine.* Technical Report. Lina & INRIA Nantes.

[3] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. 2010. Barriers to systematic model transformation testing. *Commun. ACM* 53, 6 (2010), 139–143. https://doi.org/10.1145/1743546.1743583

[4] Véronique Benzaken and Evelyne Contejean. 2019. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs.* ACM, Cascais, Portugal, 249–261. https://doi.org/10.1145/3293880.3294107

[5] Véronique Benzaken, Evelyne Contejean, Chantal Keller, and E. Martins. 2018. A Coq Formalisation of SQL's Execution Engines. In *9th International Conference on Interactive Theorem Proving.* Springer, Oxford, UK, 88–107. https://doi.org/10.1007/978-3-319-94821-8_6

[6] Gérard Berry. 2008. Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel. In *12th International Workshop on Formal Methods for Industrial Critical Systems.* Springer, Berlin, Germany, 2–2. https://doi.org/10.1007/978-3-540-79707-4_2

[7] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, San Diego, California, USA, 87–100. https://doi.org/10.1145/2535838.2535876

[8] Fabian Büttner, Marina Egea, and Jordi Cabot. 2012. On verifying ATL transformations using 'off-the-shelf' SMT solvers. In *15th International Conference on Model Driven Engineering Languages and Systems.* Springer, Innsbruck, Austria, 198–213. https://doi.org/10.1007/978-3-642-33666-9_28

[9] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. 2012. Verification of ATL Transformations Using Transformation Models and Model Finders. In *14th International Conference on Formal Engineering Methods.* Springer, Kyoto, Japan, 198–213. https://doi.org/10.1007/978-3-642-34281-3_16

[10] Daniel Calegari, Carlos Luna, Nora Szasz, and Álvaro Tasistro. 2011. A Type-Theoretic Framework for Certified Model Transformations. In *13th Brazilian Symposium on Formal Methods.* Springer, Natal, Brazil, 112–127. https://doi.org/10.1007/978-3-642-19829-8_8

[11] Zheng Cheng, Rosemary Monahan, and James F. Power. 2015. A Sound Execution Semantics for ATL via Translation Validation. In *8th International Conference on Model Transformation.* Springer, L'Aquila, Italy, 133–148. https://doi.org/10.1007/978-3-319-21155-8_11

[12] Zheng Cheng and Massimo Tisi. 2017. A Deductive Approach for Fault Localization in ATL Model Transformations. In *20th International Conference on Fundamental Approaches to Software Engineering.* Springer, Uppsala, Sweden, 300–317. https://doi.org/10.1007/978-3-662-54494-5_17

[13] Zheng Cheng and Massimo Tisi. 2017. Incremental Deductive Verification for Relational Model Transformations. In *10th IEEE International Conference on Software Testing, Verification and Validation.* IEEE, Tokyo, Japan. https://doi.org/10.1109/ICST.2017.41

[14] Zheng Cheng and Massimo Tisi. 2018. Slicing ATL model transformations for scalable deductive verification and fault localization. *International Journal on Software Tools for Technology Transfer* 20, 6 (2018), 645–663. https://doi.org/10.1007/s10009-018-0491-8

[15] Zheng Cheng, Massimo Tisi, and Rémi Douence. 2020. CoqTL: a Coq DSL for rule-based model transformation. *Software & Systems Modeling* 19, 2 (2020), 425–439. https://doi.org/10.1007/s10270-019-00765-6

[16] Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. 2017. The End of History? Using a Proof Assistant to Replace Language Design with Library Design. In *2nd Summit on Advances in Programming Languages.* Asilomar, CA, USA, 1–15. https://doi.org/10.4230/LIPIcs.SNAPL.2017.3

[17] Maribel Fernández and Jeffrey Terrell. 2013. Assembling the Proofs of Ordered Model Transformations. In *10th International Workshop on Formal Engineering approaches to Software Components and Architectures.* EPTCS, Rome, Italy, 63–77. https://doi.org/10.4204/EPTCS.108.5

[18] Xiao He and Zhenjiang Hu. 2018. Putback-based bidirectional model transformations. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, FL, USA, 434–444. https://doi.org/10.1145/3236024.3236070

[19] Soichiro Hidaka, Frédéric Jouault, and Massimo Tisi. 2017. On Additivity in Transformation Languages. In *20th International Conference on Model Driven Engineering Languages and Systems.* IEEE, Austin, TX, USA, 23–33. https://doi.org/10.1109/MODELS.2017.21

[20] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A Model Transformation Tool. *Science of Computer Programming* 72, 1-2 (2008), 31–39. https://doi.org/10.1016/j.scico.2007.08.002

[21] Frédéric Jouault and Jean Bézivin. 2006. KM3: A DSL for Metamodel Specification. In *8th International Conference on Formal Methods for Open Object-Based Distributed Systems.* Springer, Bologna, Italy, 171–185. https://doi.org/10.1007/11768869_14

[22] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2008. The Epsilon transformation language. In *1st International Conference on Model Transformations.* Springer, Zürich, Switzerland, 46–60. https://doi.org/10.1007/978-3-540-69927-9_4

[23] Kevin Lano, T. Clark, and S. Kolahdouz-Rahimi. 2014. A framework for model transformation verification. *Formal Aspects of Computing* 27, 1 (2014), 193–235. https://doi.org/10.1007/s00165-014-0313-z

[24] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[25] Salvador Martínez, Massimo Tisi, and Rémi Douence. 2017. Reactive model transformation with ATL. *Science of Computer Programming* 136 (2017), 1–16. https://doi.org/10.1016/j.scico.2016.08.006

[26] B. J. Oakes, J. Troya, L. Lúcio, and M. Wimmer. 2015. Fully verifying transformation contracts for declarative ATL. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems.* IEEE, Ottawa, ON, 256–265. https://doi.org/10.1109/MODELS.2015.7338256

[27] Object Management Group. 2016. Query/View/Transformation Specification (ver. 1.3). http://www.omg.org/spec/QVT/1.3/.

[28] Iman Poernomo and Jeffrey Terrell. 2010. Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq. In *12th International Conference on Formal Engineering Methods.* Springer, Shanghai, China, 56–73. https://doi.org/10.1007/978-3-642-16901-4_6

[29] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting proof automation to adapt proofs. In *7th ACM SIGPLAN International Conference on Certified Programs and Proofs.* ACM, Los Angeles, CA, USA, 115–129. https://doi.org/10.1145/3167094

[30] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[31] GehanM.K. Selim, Shige Wang, JamesR. Cordy, and Juergen Dingel. 2012. Model Transformations for Migrating Legacy Models: An Industrial Case Study. In *8th European Conference on Modelling Foundations and Applications.* Springer, Lyngby, Denmark, 90–101. https://doi.org/10.1007/978-3-642-31491-9_9

[32] Kurt Stenzel, Nina Moebius, and Wolfgang Reif. 2015. Formal verification of QVT transformations for code generation. *Software & Systems Modeling* 14 (2015), 981–1002. https://doi.org/10.1007/s10270-013-0351-7

[33] Massimo Tisi and Zheng Cheng. 2018. CoqTL: An internal DSL for model transformation in Coq. In *11th International Conference on Model Transformations.* Springer, Springer, Uppsala, Sweden, 142–156. https://doi.org/10.1007/978-3-319-93317-7_7

[34] Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. 2011. Lazy execution of model-to-model transformations. In *14th International Conference on Model Driven Engineering Languages and Systems.* Springer, Springer, Wellington, New Zealand, 32–46. https://doi.org/10.1007/978-3-642-24485-8_4

[35] Javier Troya and Antonio Vallecillo. 2011. A Rewriting Logic Semantics for ATL. *Journal of Object Technology* 10, 5 (2011), 1–29. https://doi.org/10.5381/jot.2011.10.1.a5

[36] Gergely Varró, Katalin Friedl, and Dániel Varró. 2006. Implementing a graph transformation engine in relational databases. *Software & Systems Modeling* 5, 3 (2006), 313–341. https://doi.org/10.1007/s10270-006-0015-y

[37] Dennis Wagelaar. 2014. Using ATL/EMFTVM for import/export of medical data. In *2nd Software Development Automation Conference.* Amsterdam, Netherlands.

[38] Dennis Wagelaar, Massimo Tisi, Jordi Cabot, and Frédéric Jouault. 2011. Towards a General Composition Semantics for Rule-Based Model Transformation. In *14th International Conference on Model Driven Engineering Languages and Systems.* Springer, Wellington, New Zealand, 623–637. https://doi.org/10.1007/978-3-642-24485-8_46