Deep Specification and Proof Preservation for the CoqTL Transformation Language

Zheng Cheng, Massimo Tisi

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

Deep Specification and Proof Preservation for the CoqTL Transformation Language

Zheng Cheng · **Massimo Tisi**

the date of receipt and acceptance should be inserted later

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. 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 proofs that leverage this specification and are thus preserved through the updates. Finally, we discuss the evolution of the deep specification by an extension mechanism, we present an evolution that introduces trace links in the specification, and we show which user proofs are preserved through specification evolutions.

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

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

Keywords MDE, Model Transformation, Programming Language Implementation, Certification, Theorem Proving, Coq

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 [43], medical data processing [50], aviation [7]), this analysis is crucial to guarantee that the MT will not generate faulty models.

In previous work, we presented CoqTL, a RMT language implemented as an internal DSL in the Coq interactive theorem prover [46]. 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 [3,36,47,51], 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 [2], 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`

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 is 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 argue that our way of structuring specification and proofs for CoqTL can be adapted to other RMT languages as well (e.g. ATL [30], QVT [39], ETL [32]), with the purpose of interfacing them with theorem provers.

Of course, specifications, like any software-engineering artifact, are themselves subject to evolution. Unconstrained evolution of our specifications may easily break existing user proofs in the future. This paper extends [20] by the same authors, by addressing this issue in Section 6. We provide the following contributions: 1) We show that some user-proofs on the deep specification can be preserved through specification evolutions, by representing these evolutions as *specification extensions*, 2) We present a specification extension that introduces the concept of trace links in the specification, and we certify the engine against this extended specification.

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 implementation updates. Section 6 shows an evolution mechanism for the CoqTL deep specification, exemplified by an evolution that adds the concept of trace links to the specification. We show which user proofs are preserved through the specification evolution. Section 7 identifies limitations and perspectives of our approach. Section 8 compares our work with related research, and Section 9 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, Class2Relational. The example is intentionally very small, so that it can be completely illustrated within this paper. In the evaluation section (Section 5) we will use a more realistic example, i.e. the transformation of hierarchical state machines into flat state machines, where CoqTL has been used to prove interesting properties in previous work [19].

The structure of the involved metamodels for Class2Relational is shown in Fig. 1.

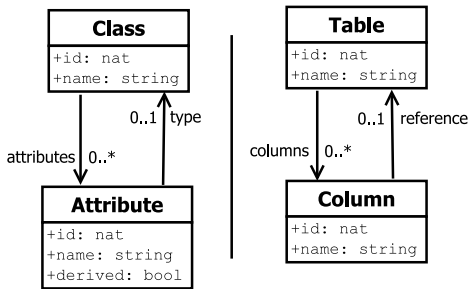

Fig. 1 Minimal metamodels for class diagrams (left), and relational schemas (right).

The left part of Fig. 1 shows the simplified structural metamodel of class diagrams. Each class diagram contains a list of named classes, 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, 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 [29] (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 `Class2Relational` is to transform a model conforming to the `Class` metamodel to a model conforming to the `Relational` metamodel, and we name the input model as `m` (lines 2- 3). Then, the transformation is defined via two rules in a mapping style: one maps `Classes` to `Tables`, another one maps `non-derived Attributes` to `Columns`. Each rule in CoqTL has a `from` section that specifies the input pattern to be matched in the source model. A boolean expression in Gallina can be added as guard, and a rule is applicable only if the guard evaluates to `true` for a certain assignment of the input pattern elements. Each rule has a `to` section which specifies elements and links to be created in the target model (output pattern) when a rule is fired. The `to` section is formed by a list of labeled `outputs`, each one including an `element` and a list of `links` to create. The `element` section includes standard Gallina code to instantiate the new element specifying the value of its attributes (line 11). The `links` section contains standard Gallina code to instantiate links 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

```
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

(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).

[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`

```
1  Theorem tables_name_defined :
2    ∀ (cm : ClassModel) (rm : RelationalModel),
3      (* transformation *)
4      rm = execute Class2Relational cm  →
5      (* precondition *)
6      (∀ (c : Class),  In c (allModelElements cm)→
7        length (getClassName c)>0) →
8      (* postcondition *)
9      (∀ (t : Table),  In t (allModelElements rm)→
10       length (getTableName t)>0).
```

Listing 2 Name definedness theorem for the `Class2Relational` transformation

In the `Attribute2Column` rule we can notice the presence of a guard. When the `Attribute` is not `derived`, a `Column` is constructed with the same name 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.

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

For example, Listing 2 shows 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 [11,14]). 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 [46].

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 9) 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).

```
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    simpl in H1.
9      (* H1: In t (flat_map (...) (...) *)
10   apply in_flat_map in H1.
11     (* H1: ∃ sp : list ClassMetamodel_EObject,
12            In sp (allTuples Class2Relational cm) ∧
13            In t (toList (instantiatePattern
14            Class2Relational cm sp)) *)
15   ...
```

Listing 3 First steps of a proof for tables_name_defined

– 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 [41]. 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 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 [19]. There, the type of models is defined by a type class in Coq , and includes a list of `ModelElements` and a list of `ModelLinks`:

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

```
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

```
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`).

```
(* 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

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 to navigate the syntactic structure of the transformation.

```
Inductive Rule : Type :=
  buildRule :
    (* name *) string
    (* types *) → (list SourceModelClass)
    (* from *) → Guard
    (* to *) → (list OutputPatternElement)
    → Rule.

Definition Rule_getOutputPatternElements: Rule → list OutputPatternElement := ...
```

Listing 6 Abstract syntax for the syntactic type `Rule`

For example, the type of `Rule` and one of its accessors `Rule_getOutputPatternElements` are defined as in Listing 6. Here a rule contains in order: 1) the rule name, 2) the types of the pattern elements to match (where `SourceModelClass` is the type of metaclasses in the source metamodel), 3) a match function defined by the syntactic type `Guard`, 4) a list of elements to generate for each match, defined by the syntactic type `OutputPatternElement`[4].

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 of semantic functions is shown in Listing 7. 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[5]. 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.

[4] CoqTL supports also a syntax for iterative rules that is not shown for brevity.

[5] 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 → TargetModelElement →
                                 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 7 Hierarchy of semantic functions

`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 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 [19]), `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 [30].

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 `nat`ural.

`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 `trace links`, that is very common in engines for RMT languages. Indeed, for a minimal specification of the input-output semantics of the engine, it is not necessary to introduce the concept of trace links. However, in Section 6 we will show that extending this specification to explicitly consider trace-links may still be useful, since it allows users to predicate on trace links in their theorems and proofs.

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 the following, 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 7 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 7 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 a few examples. We refer the reader to our online repository for the full specification[6].

[6] `https://github.com/atlanmod/CoqTL/tree/2a8cea5`

Membership lemmas. As shown in Listing 7 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.

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:

```
1 Lemma execute_In_elements :
2   ∀ (tr : Transformation) (sm : SourceModel) (te : TargetModelElement),
3     In te (allModelElements (execute tr sm)) ↔
4     (∃ (sp : list SourceModelElement) (tp : list TargetModelElement),
5         incl sp (allModelElements sm) ∧
6         instantiatePattern tr sm sp = Some tp ∧
7         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 7 .

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[7]:

```
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);
```

[7] 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

```
Class TransformationEngine :=
{

  (** Metamodels *)
  SourceMetaModel := Metamodel SourceModelElement SourceModelLink
    SourceModelClass SourceModelReference;
  TargetMetaModel := Metamodel TargetModelElement TargetModelLink
    TargetModelClass TargetModelReference;

  (** Models *)
  SourceModel := Model SourceModelElement SourceModelLink;
  TargetModel := Model TargetModelElement TargetModelLink;

  (** Abstract Syntax *)
  Rule : Type;
  ...

  (** Semantic Functions *)
  execute : Transformation → SourceModel → TargetModel;
  ...

  (** Semantic Lemmas *)
  tr_execute_in_elements : ...
  ...
}
```

Listing 8 The deep-specification type class

Deep Specification. Finally, specifications of Model and Metamodel, signatures of the semantic functions and semantic lemmas are gathered in the main type class of the deep specification, `TransformationEngine` (Listing 8). To certify against the specification, engine developers have to demonstrate that their engine is an instance of this type class, by showing that it contains an implementation of the semantic functions and proving that the semantic lemmas hold.

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.

4.1 The CoqTL Engine

In general, 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.

```
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 9 Three versions of the execute function

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[8]. In the following, implementations of the CoqTL engine are denoted as CoqTL$_x$, where x indicates the version of the implementation. We refer to these three versions as CoqTL$_a$, CoqTL$_b$ and CoqTL$_c$.

Lines 1 - 4 in Listing 9 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 9). 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$_a$.

[8] `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)

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 `instantiate-Pattern`$_c$ 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.

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 7.

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:

```
Proof.
  intros.
    (* H: rm = execute Class2Relational cm
       H1: In t (allModelElements rm) *)
  rewrite H in H1.
    (* H1: In t (allModelElements (execute
            Class2Relational cm)) *)
  apply execute_In_elements in H1.
    (* H1: ∃ (sp : list ClassMetamodel_EObject)
           (tp : list RelationalMetamodel_EObject),
           incl sp (allModelElements cm) ∧
           instantiatePattern Class2Relational cm sp = return tp ∧
           In t tp *)
   ...
```

Listing 10 First steps of a proof for tables_name_defined that uses the specification lemmas

```
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 $\texttt{execute}_a$[9].

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 Proof Preservation

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 10 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

[9] Only one version of the engine is included in each repository snapshot. That engine is always certified against the type class, and not against another version of the engine. The *execute_preserv* lemma is one of the effective means to achieve this certification. Hence, we put it, together with a copy of all the necessary semantic functions of a previous version, into the "Certification" module of the new version.

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 10.

Also, the final proof state in Listing 10 (lines 9 - 13) is equivalent to the one in Listing 3, but more abstract. In particular, the formula `incl sp (allModelElements cm)` in Listing 10 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), but 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 impact 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 10.

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)

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 [4, 11]. `HSM2FSM` is a transformation that performs a flattening algorithm for hierarchical state machines. The transformation requires 7 rules, for a total 205 LoC.

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.

Out of the 12 theorems shown in Table 2, 4 proofs based on implementation are expected to be impacted by the update of the `execute` function to CoqTL$_b$ or CoqTL$_c$. The other proofs are broken by similar updates to the `match` or `instantiate` function. As we expected, all 12 proofs using the specification are instead preserved through the engine updates, 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 that 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.

```
Theorem table_name_defined_on_trace :
  ∀ (cm : ClassModel) (c : Class) (t : Table) (tl : TraceLink),
    (* trace link *)
      (In tl (trace Class2Relational cm) ∧
       In c (TraceLink_getSourcePattern tl) ∧
       t = (TraceLink_getTargetElement tl)) →
    (* precondition *)
      length (getClassName c)>0) →
    (* postcondition *)
      length (getTableName t)>0).
```

Listing 11 Name definedness theorem for the `Class2Relational` transformation

6 Specification Evolution by Extension

By relying on a deep specification for CoqTL, user proofs are robust w.r.t. evolution of the transformation engine implementation. However, as any kind of software artifact, specifications are also prone to evolution in time. They may be updated with the purpose of fixing specification errors, refactoring for elegance or conciseness, or extending the specification scope. Of course, user proofs that depend on the deep specification are not robust w.r.t. general evolution of the specification itself.

This section illustrates this problem using an example, by describing an important evolution of the deep specification for CoqTL. The evolution introduces the concept of *trace link* to the specification. The concept of trace link is common in transformation languages like ATL and ETL. In general, it is a link connecting: 1) some source elements, 2) the corresponding target elements, and 3) the syntactic transformation elements that are responsible for the correspondence.

Listing 11 provides an example of user theorem that predicates on trace links, as enabled by the updated specification. The theorem `table_name_defined_on_trace` predicates on the `Class2Relational` transformation shown in Listing 1. For any source class c and target table t, which are linked by a trace link tl generated during transformation execution, the theorem states that, if c has a non-empty name, then t has a non-empty name too. While the theorem has its own interest, it is also a potentially useful intermediate result in the proof of theorem `tables_name_defined` in Listing 2.

Adding the concept of trace links to the deep specification would not change the input/output behavior of the engine, since trace links are only used internally during the transformation execution. In the deep specification, the signature of several semantic functions would need to be updated, to pass the computed trace links in function calls. Semantic lemmas would need to be updated accordingly.

An in-place update of the specification in Listing 8 would give the result outlined in Listing 12. This update contains:

- semantics functions and lemmas whose semantics is unchanged from the previous specification, e.g. `execute`.
- new and updated semantic functions that consider the set of trace links, e.g. `tracePattern`.
- new and updated semantic lemmas that complete the hierarchical structure of the evolved deep specification for the new and updated semantic functions. For

```
Class TransformationEngine :=
  {
    (** Unchanged semantic functions and lemmas *)
    execute: Transformation → SourceModel → TargetModel;
    ...

    (** New/Updated semantic functions *)
    tracePattern: tracePattern: Transformation → SourceModel →
                list SourceModelElement → list TraceLink;
    ...

    (** New/Updated semantic lemmas *)
    tr_execute_in_elements :
      ∀ (tr : Transformation) (sm : SourceModel) (te : TargetModelElement),
        In te (allModelElements (execute tr sm)) ↔
        (∃ (tl : TraceLink) (sp : list SourceModelElement),
          In sp (allTuples tr sm) ∧
          In tl (tracePattern tr sm sp) ∧
          te = TraceLink_getTargetElement tl)
    ...
  }
```

Listing 12 In-place update of the deep specification of CoqTL, to include trace links

```
Class TraceAwareTransformationEngine (base : TransformationEngine) :=
  {
    (** Semantic functions for the update *)
    tracePattern: tracePattern: Transformation → SourceModel →
                list SourceModelElement → list TraceLink;
    ...

    (** Semantic lemmas entailed by the update *)
    tr_execute_in_elements_traces :
      ∀ (tr : Transformation) (sm : SourceModel) (te : TargetModelElement),
        In te (allModelElements (execute tr sm)) ↔
        (∃ (tl : TraceLink) (sp : list SourceModelElement),
          In sp (allTuples tr sm) ∧
          In tl (tracePattern tr sm sp) ∧
          te = TraceLink_getTargetElement tl)
    ...
  }
```

Listing 13 Representation of the specification update as a specification extension

example, now the `tr_execute_in_elements` lemma characterizes the `flat_map` relation between `execute` and `tracePattern`: each target element in the result of `execute` is connected by a trace link computed by the `tracePattern` function.

Of course such update of the specification (assuming that it is possible to certify the engine against it), would in general invalidate user proofs. This is not surprising for user theorems that predicate on updated semantic functions, since such user theorems may even be falsified by the update in the function semantics. More interestingly, the update may generally break also user proofs on functions that are unchanged in signature and input/output behavior. This is due to the nature

of the proof steps, that perform syntactical manipulations of the proof state. For example, consider the user proof we already shown in Listing 10, assuming now that we have updated the base specification to Listing 12. Trying to execute this user proof would produce an error. Line 8 of Listing 10 would apply the new version of `execute_In_elements`. The resulting proof state would be syntactically different than the one shown at lines 9 - 13 of Listing 10 (even if semantically equivalent), e.g. by not including any reference to the function `instantiatePattern`. Hence, the application of lemmas on `instantiatePattern` would raise an error in the rest of the proof.

We want to preserve user theorems that predicate on functions whose input/output semantics is not changed by the update. To this purposes, we represent the specification evolution as a *specification extension*. We first identify which semantic functions in the updated version are semantically unchanged by the update (they always produce the same outputs given the same inputs). We call them *preserved semantic functions*. We call *preserved semantic lemmas* all the lemmas that only predicate about preserved semantic functions. Then the updated specification is represented as a type class inheriting from the base specification (exemplified by Listing 13). Preserved semantic functions and lemmas are simply inherited from the base specification (e.g. `execute`). Only updated functions are specified in the extension (e.g. `tracePattern`). Renaming is needed if updated function/lemma names clash with the ones that are already in the base specification (e.g. `tr_execute_in_elements` of Listing 12 is renamed to `tr_execute_in_elements_traces`).

It is important to highlight that only one version of the preserved semantic functions exists, the one imported by the base specification. Both base and updated semantic lemmas refer to the same preserved semantic functions (e.g. `tr_execute_in_elements` and `tr_execute_in_elements_traces` both refer to the same imported version of `execute`). This guarantees for preserved semantic functions that the update does not replace the base specification but adds lemmas for new properties, typically as relations to the new/updated semantic functions. [10]

The main result obtained by this approach is that: *existing proofs for user theorems that predicate exclusively on preserved semantic functions are still valid after the specification update, with no need for any adaptation.* First of all such user proofs are indeed validated by Coq, since all the functions and lemmas of the base specification that may be used in proof steps are inherited by the updated specification. It is important to notice that these proof steps would typically rely on outdated functions and lemmas, but this is irrelevant to the validity of the proof for the new engine: since all the preserved semantic functions are exactly the same used by the updated specification, the statement of the theorem is not impacted by the update, hence any proof of that statement is equivalent.

The base version of non-preserved semantic functions is deprecated in the update. Existing proofs for user theorems that predicate on these functions are still formally valid after the specification update, but they have limited usefulness, since

[10] Notice that it is technically possible to contradict in new lemmas the base specification of a preserved semantic function. However this would contradict the assumption on the semantic preservation of the function and make the specification impossible to implement.

they predicate on functions that are not implemented anymore in new versions of the engine.

To illustrate the specification evolution mechanism, and validate a trace-link-aware specification w.r.t. our base deep specification for CoqTL, we consider a fourth point of CoqTL in its development history, that we name CoqTL$_d$. We prove that CoqTL$_d$ is certified w.r.t. the trace-link-aware specification. In addition, we show that, as a consequence of the extension mechanism, the proofs of all user theorems mentioned in the previous sections hold for CoqTL$_d$ too.

6.1 Deep Specification for CoqTL with Trace Links

To illustrate evolution by extension using an example, we define the extended specification with trace links as a Coq type class that extends the type class of the base specification (from Listing 8), using the Coq type class inheritance mechanism. We add to the new type class a representation for trace links, new semantic functions, and new lemmas, as described in the following.

Trace link representation. A trace link in the new CoqTL engine captures the fact that each target model element is generated by instantiating an output pattern element of a transformation rule on a certain iteration over a source pattern. To represent it, we introduce a new `TraceLink` type, with a set of accessors:

```
Inductive TraceLink : Type.
Definition TraceLink_getTargetElement (tl: TraceLink): TargetElement.
Definition TraceLink_getSourcePattern (tl: TraceLink): list SourceModelElement.
Definition TraceLink_getOutputPatternElementName (tl: TraceLink): string.
Definition TraceLink_getRuleName (tl: TraceLink): string.
Definition TraceLink_getIteration (tl: TraceLink): nat.
```

As shown by the accessors, a `TraceLink` is associated to a single target model element, and for that model elements it records: 1) the set of source model elements (i.e., source pattern) that triggered its generation, 2) the name of output pattern element, 3) transformation rule and 4) iteration number that produced it.

Semantic functions. We identify that the semantics of `execute`, `match-` and `resolve-` functions is not impacted by the update.

While the full hierarchy of semantic functions from the base specification (shown in Listing 7) is preserved, we show the new semantic functions included in the extension in Listing 14. The extension maintains the hierarchical structure of the specification: the new semantic functions are organized in a tree, where a composition pattern connects parent and children. The full specification update can be obtained by merging the trees in Listing 7 and 14. We include the preserved semantic function `execute` in both listings, to indicate the exact merging point.

It can be noticed that the functions `tracePattern`, `traceRuleOnPattern`, `traceIterationOnPattern` and `traceElementOnPattern` have a one-to-one correspondence to `instantiatePattern`, `instantiateRuleOnPattern`, `instantiateIterationOnPattern` and `instantiateElementOnPattern` from Listing 7, respectively. Corresponding *trace-* and *instantiate-* functions essentially perform the same computation. However, as shown by their signatures, the *trace-* functions in the extended specification do not produce target model element(s), but trace link(s).

```
(* the preserved semantic function *)
execute: Transformation → SourceModel → TargetModel;
  (* flat_map *)
  tracePattern: Transformation → SourceModel → list SourceModelElement →
                list TraceLink;
    (* flat_map *)
    traceRuleOnPattern: Rule → SourceModel → list SourceModelElement →
                        list TraceLink;
      (* flat_map *)
      traceIterationOnPattern: Rule → SourceModel →
                               list SourceModelElement → nat → list TraceLink;
        (* flat_map *)
        traceElementOnPattern: OutputPatternElement → SourceModel →
                               list SourceModelElement → nat →
                               option TraceLink;
  (* flat_map *)
  applyPatternTraces: Transformation → SourceModel →
                      list SourceModelElement →
                      list TraceLink → list TargetModelLink;
    (* flat_map *)
    applyRuleOnPatternTraces: Rule → SourceModel →
                              list SourceModelElement → list TraceLink →
                              list TargetModelLink;
      (* flat_map *)
      applyIterationOnPatternTraces: Rule → SourceModel →
                                     list SourceModelElement → nat →
                                     list TraceLink → list TargetModelLink;
        (* flat_map *)
        applyElementOnPatternTraces: OutputPatternElement →
                                     SourceModel → list SourceModelElement →
                                     nat → list TraceLink →
                                     list TargetModelLink;
          (* flat_map *)
          applyReferenceOnPatternTraces: OutputPatternElementReference →
                                         SourceModel →
                                         list SourceModelElement → nat →
                                         TargetModelElement →
                                         list TraceLink →
                                         option TargetModelLink;
```

Listing 14 Hierarchy of extended semantic functions

The functions `applyPatternTraces`, `applyRuleOnPatternTraces`, `applyIterationOnPatternTraces`, `applyElementOnPatternTraces`, and `applyReferenceOnPatternTraces` are analogous to the corresponding `applyPattern`, `applyRuleOnPattern`, `applyIterationOnPattern`, `applyElementOnPattern` and `applyReferenceOnPattern` from Listing 7, respectively. Corresponding *apply-* functions have the same signatures, except that functions in the extended specification have a set of trace links as an extra argument[11]. Hence *apply...Traces* functions are able to make use of a precomputed set of trace links, when needed.

[11] In the Coq code, an extra parameter of type `Transformation` is added to all apply functions for helping type inference, without contributing to their functionality. We omit such technical aspect in Listing 14 for conciseness.

Lemmas overview. The formal semantics of the extended semantic functions is given by additional lemmas that the extended engine will need to certify against.

The trace-aware specification extension contains 11 lemmas defining the semantic functions' new behavior. The additional lemmas follow the same categorization we followed for the base deep specification of CoqTL[12].

In the following we describe a few exemplary lemmas. We refer the reader to our online repository for the full list[13].

Membership lemmas. We define 9 membership lemmas in the extended deep specification of CoqTL. They are characterized by the `flat_map` relation. We therefore instantiate the template lemma `in_flat_map`, shown already in Listing 4. For instance, to characterize the `flat_map` relation between `execute` and `tracePattern`, we specialize `in_flat_map` and produce the following lemma:

```
Lemma execute_In_elements_traces :
  ∀ (tr : Transformation) (sm : SourceModel) (te : TargetModelElement),
    In te (allModelElements (execute tr sm)) ↔
    (∃ (sp : list SourceModelElement) (trl : TraceLink),
        In sp (allTuples tr sm) ∧
        In trl (tracePattern tr sm sp)) ∧
        te = TraceLink_getTargetElement trl).
```

The lemma states that an element `te` is included in the target model elements of the transformation if and only if, by applying the function `tracePattern` to a source pattern `sp` chosen in the list of all possible tuples from the source model `sm`, we obtain a trace link whose target element is `te`.

This lemma introduces a new property for the preserved semantic function `execute`, by specifying that any target element needs to be connected by a trace link to a source pattern. Analogous lemmas for `traceRuleOnPattern`, `traceIterationOnPattern` and `traceElementOnPattern` specify the connection of the trace link with the rule name, the iteration number and the output pattern element name, respectively.

The following lemma `execute_In_links_traces` defines the relation between `execute` and `applyPatternTraces`:

```
Lemma execute_In_links_traces :
  ∀ (tr : Transformation) (sm : SourceModel) (tl : TargetModelLink),
    In tl (allModelLinks (execute tr sm)) ↔
      (∃ (sp : list SourceModelElement),
         In sp (allTuples tr sm) ∧
         In tl (applyPatternTraces tr sm sp (trace tr sm)));
```

The lemma states that a link `tl` is included in the output of the transformation execution if and only if it is included in the result of the function `applyPatternTraces` applied to some source pattern `sp` (chosen in the list of all possible tuples from the source model `sm`) and receiving in argument the full list of trace links of the transformation execution `(trace tr sm)`.

Intuitively, the theorem states that every link in the transformation result has to be produced by an application of the `applyPatternTraces` function, where

[12] We do not define error lemmas since the characterization of error cases is: a) suppressed in the extended specification for non-leaf functions (i.e. they should always return non-none values); b) delegated to the generic expression evaluation functions for leaf functions.

[13] `https://github.com/atlanmod/CoqTL/tree/948eb94`

`applyPatternTraces` can access the traces produced by the transformation execution. Membership lemmas for `applyRuleOnPatternTraces`, `applyIterationOnPatternTraces`, `applyElementOnPatternTraces`, and `applyReferenceOnPatternTraces` define the analogous necessity for target links to be produced by an execution of each of these functions, if these functions can access the trace links of the transformation.

Leaf lemmas. There are 2 leaf semantic functions in the extended deep specification, therefore producing 2 leaf lemmas.

The semantics of the leaf function `traceElementOnPatternTraces` is given by the following lemma:

```
Lemma tr_traceElementOnPatternTraces_leaf :
  ∀ (o : OutputPatternElement) (sm : SourceModel)
    (sp : list SourceModelElement) (i : nat) (tl : TraceLink),
    Some tl = (traceElementOnPattern o sm sp i) ↔
    (∃ (te : TargetModelElement),
       Some te = (instantiateElementOnPattern o sm sp i) ∧
       tl = (buildTraceLink sp (OutPatternElement_getName o)
              (OutPatternElement_getRule o) i te)).
```

This lemma is particularly important because it specifies the relation between a trace link and element instantiation. The lemma states that if a trace link `tl` is produced (by `traceElementOnPattern`) then this means that a target element `te` is instantiated (by `instantiateElementOnPattern`). The lemma also describes the structure of the trace link `tl`, as a record of: 1) the source pattern `sp` that triggered its generation, 2) the name of output pattern element (`OutPatternElement_getName o`), 3) transformation rule (`OutPatternElement_getRule o`), 4) iteration number (`i`) that produced its generation, and 5) the target element (`te`) that has been instantiated as a result.

In the `apply-` functions, like the `applyReferenceOnPatternTraces` leaf, trace links may be used to improve performance, but they do not change the final result of the function (i.e., the one that would be computed by their base versions). This is evident by noticing that the leaf lemma for `applyReferenceOnPatternTraces` is almost identical to the leaf lemma for `applyReferenceOnPattern` in the base specification. Indeed, we trivially prove the following derived lemma that states the equivalence of `applyReferenceOnPatternTraces` and `applyReferenceOnPattern`, *when the full set of traces is available* (`trace tr sm`).

```
Lemma tr_applyReferenceOnPatternTraces_equiv :
  ∀ (tr : Transformation) (oper : OutputPatternElementReference)
    (sm : SourceModel) (sp : list SourceModelElement) (iter : nat)
    (te : TargetModelElement),
  applyReferenceOnPatternTraces oper sm sp iter te (trace tr sm) =
    applyReferenceOnPattern oper sm sp iter te.
```

6.2 Engine Certification

In this section we validate the specification by certifying a version of CoqTL against it. Therefore, we isolate a version of CoqTL (commit `[da3560d]`), which we refer to as CoqTL$_d$. Internally, CoqTL$_d$ makes use of trace links, by computing them in a first phase, and using them to initialize reference features (i.e., a two-step semantics inspired from ATL).

```
Definition execute_d (tr : Transformation) (sm : SourceModel) :=
  let tls :=
    (flat_map (λ sp ⇒ tracePattern tr sm) (allTuples tr sm)) in
  Build_Model
    (map TraceLink_getTargetElement tls)
    (flat_map (λ sp ⇒ applyPatternTraces tr sm sp tls) (allTuples tr sm)).
```

Listing 15 Execute function of CoqTL$_d$

There are only a few differences between the implementation of CoqTL$_d$ and the already described CoqTL$_c$. The entry function for CoqTL$_d$, `execute`$_d$, is shown in Listing 15. First, `execute`$_d$ pre-computes trace links by `flat_mapping` `tracePattern` on all candidate source patterns generated by the `allTuples` function. Then, the target model is built (`BuildModel`) with two arguments:

- The list of target model elements is simply retrieved from the trace links, by calling their `TraceLink_getTargetElement` accessor.
- The list of target model links is computed by executing `applyPatternTraces` on all the possible tuples, and concatenating the result. The list of trace links is passed as argument to each call to `applyPatternTraces`.

The implementation of the `tracePattern` function and its auxiliary functions is adapted from the instantiation ones in CoqTL$_c$. Besides the new `execute`$_d$, CoqTL$_d$ introduces an efficient implementation for `applyPatternTraces` and its auxiliary functions. This implementation leverages the fact that the full list of trace links has been computed before the first call to `applyPatternTraces`.

We report on the certification effort for CoqTL$_d$. This certification also ensures that preserved semantic functions are implemented to satisfy the lemmas from the base specification, and the trace-aware lemmas. We exemplify on the `execute` function.

We start by proving that the `execute`$_d$ has the same input-output behavior than `execute`$_c$. This is formalized by a preservation lemma, similar to how we proceeded in Section 4.2:

```
Lemma execute_preserv : ∀ (tr : Transformation) (sm : SourceModel),
  execute_d tr sm = execute_c tr sm.
```

This equivalence allows us to immediately prove the top-level lemmas for the `execute` of the base specification `execute_In_elements`, `execute_In_links`, `execute_Some` and `execute_None`, simply by substituting `execute`$_c$ with `execute`$_d$ in the proofs of CoqTL$_c$. Then, we certify `execute`$_d$ against the trace-aware lemmas. While doing this, we benefit again from our hierarchical organization of lemmas: by noting that proofs of lemmas in the same category have a similar structure, we reuse/adapt similar induction principles and patterns to reduce the proof effort.

The 2nd part of the certification is to ensure non-preserved semantic functions are still presented in the certifying implementation to satisfy the lemmas from the base specification. For that, we import implemented non-preserved functions (i.e. `instantiate-` and `apply-` functions) and their corresponding certifications from CoqTL$_c$ with no change.

Finally, we certify trace-aware semantic functions (i.e. proof for lemmas related to `trace-` and `apply...Traces` functions). All certification proofs are manually

developed, mechanically checked by Coq, and are publicly available in the CoqTL repository.

6.3 Proof Preservation

We report that as expected, all the user proofs discussed in Section 5 validate over CoqTL$_d$. Indeed, these proofs depend exclusively on lemmas and functions of the base specification. The certification of CoqTL$_d$ against the extended semantics ensures that the base lemmas continue to hold, and that the semantic functions have the same input/output semantics required by the base specification.

However, in this section we are not only interested in evolution scenarios that change the implementation (e.g. CoqTL$_a$ to CoqTL$_d$) but also in evolutions that change the specification and corresponding certification.

To analyze these evolutions, we consider an initial state containing a base specification (BS) and a child specification (CS), that extends the base specification. We consider to have two engines that implement these specifications, certified respectively by a base-specification certification (BSC) and a child-specification certification (CSC).

Note that this general initial state includes also the case where only a base specification exists, as for CoqTL$_{a-c}$. In this case, we consider an initial CS as empty, but still inheriting from BS, and an initial CSC as identical to BSC.

Artifact	Operation	User proofs on BS	User proofs on CS	BS Certif.	CS Certif.
CS	**Add**	preserved	preserved	preserved	not preserved
CS	**Remove/Modify**	preserved	not preserved	preserved	not preserved
BSC	**Add**	preserved	preserved	preserved	preserved
BSC	**Remove/Modify**	preserved	preserved	not preserved	not preserved
CSC	**Add**	preserved	preserved	preserved	preserved
CSC	**Remove/Modify**	preserved	preserved	preserved	not preserved

Table 3 Proof preservation for different engine evolution scenarios. Operations on CS add/remove/modify function signatures and lemmas. Operations on BSC and CSC can add/remove/modify also function bodies and proofs.

Here we analyze which proof artifacts in CoqTL are preserved through different evolutions of the base or child specification/certification. The result is summarized in Table 3. We enumerate the possible elementary evolution scenarios, through two dimensions:

- evolving artifact: CS, BSC or CSC
- evolution operation: addition/removal/modification of a definition (e.g. function, lemma) to the artifact.

Note that we do not allow changes to the base specification. Such changes, according to the specification extension mechanism, should be always applied to (possibly new) child specifications.

Each column represents a proof artifact that may be preserved or not: user proofs that depend on BS, user proofs that depend on CS, certification proofs

against BS, or certification proofs against CS. Each cell shows whether the complete proof artifact still holds (i.e., the Coq compiler validates the proof with no error) when the chosen evolution scenario occurs.

The first two columns show the preservation of user proofs. Changes to certification artifacts (BSC and CSC) do not impact user proofs. This is a consequence of the proof preservation property shown in Section 5. Moreover user proofs depending on BS are not impacted by changes to CS, as expected by the specification extension mechanism. Finally, user proofs depending on CS are preserved by additions of lemmas or semantic functions to CS, since they are not used by the proof.

The third column shows that proofs in BSC are impacted only by removals or modifications within the same artifact. Indeed such updates could break existing certification proofs. Additions to the BSC artifact are again harmless, since they are not used by the proofs. Finally updates to child artifacts have no effect on BSC.

The last column shows that certification proofs against the child specification are generally not preserved, when certification or specification artifacts evolve. Only adding certification functions or theorems is a harmless evolution, that guarantees proof preservation for CSC.

As an experimental validation of Table 3, we perform evolutions starting from commit [254bd71] in the history of CoqTL. We first ensure that this point in history is accepted by the Coq prover, by executing the built-in compiler of Coq (*coqc*), that validates proofs during compilation. We perform six concrete evolutions of the specification and certification, one for each line in Table 3. For additions we add a trivial tautology definition (for CS) or its proof (for BSC and CSC). For removals/modifications, we delete a randomly chosen proof (for BSC and CSC), and we modify a randomly chosen definition (for CS). After these evolutions, we report that *coqc* returns errors only for the cases marked as *non preserved* in Table 3. The full artifacts related to the analysis in this section can be found at our online repository: `https://github.com/atlanmod/CoqTL/tree/948eb94/fr.inria.atlanmod.coqtl.coq/preservation`.

7 Limitations and Perspectives

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

From a practitioner point of view, the current approach has the following limitations:

- User proofs that exclusively rely on the type class are more complex and verbose. 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 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.
- 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.
- The actual signatures of the semantic functions in Coq contain some technical arguments that are omitted in Listing 7. 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 a hypothetical update to these functions would break the user proofs.

In this work, we enable proof persistence by proposing a stable transformation engine interface. Other RMT languages (e.g. ATL, ETL) may benefit from our experience. In order to interface with Coq, existing RMT languages inevitably need to encode their executable semantics in Coq. This encoding may use a similar hierarchical organization as in CoqTL, by replacing our semantic functions with specific ones for the given RMT language. Moreover, in order to guarantee user proof preservation, other RMT languages may define semantic lemmas, as we do, and certify the executable semantics against them.

Our usage of specification extension can be replicated for specification of other languages as well. Often MT engine lifelines fork into different implementations with significant differences in semantics. For instance, the ATL community produced three independent virtual machines with different features[14], and also a set of forks implementing specific execution strategies (incremental, lazy, reactive, share-memory parallel, distributed). In CoqTL, we plan to manage engine branching as a double hierarchy of implementations and specifications. Engine forks and rewritings are organized in a hierarchy, where each branch is focused on a particular application scenario. When a fork is associated with specific concepts that may be useful in user theorems and proofs, we produce an associated specification that extends the specification of the parent engine. User proofs that depend on a specification extension, hold for all the engines that certify against that extension. Such user proofs are robust to implementation evolutions and further branching of the specifications.

Users can make proofs more robust to engine updates by various alternative means, e.g. by a finer modularization of their user theorems in lemmas or by the development of ad-hoc automatic tactics. While our proposal assigns the user-

[14] RegularVM, EMFVM, EMFTVM (`https://wiki.eclipse.org/ATL/VM_Comparison`)

proof preservation concern to the responsibility of engine developers, it can be complemented by these good practices in proof engineering.

Besides enabling user-proof preservation, certifying against a stable deep specification guarantees the absence of regression bugs (on the part of the semantics that is included in the specification) in the engine development process. Because of the significant effort required by the certification, we perform this certification for CoqTL only for publicly released major milestones.

Specification extensions may be composable. In this case the composed specification can be obtained by performing one extension on top of the other. Unfortunately certification proofs are not generally composable. Certifying a composed specification typically requires to manually replicate a significant part of the certification effort of each component certification. We plan to investigate this issue in future work.

8 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 [42] where rewriting rules are used to define executable language semantics.

Within the Coq community, the DeepSpec project [2] 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 deep specifications are being developed in the project. The list includes Leroy's CompCert [34], i.e. a Coq specification and verified optimizing compiler for a large subset of the C programming language, the DataCert project working towards a certified SQL engine [5, 6], Haskell CoreSpec [52] formalizing a core subset of Haskell, and Kami [22] with a deep specification of the BlueSpec language.

In the same spirit, the JSCert project [8] provides a formal specification of Javascript in Coq and a reference certified interpreter.

Furthermore, Chlipala et al. [21] 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 [39]. 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 [48]. The goal is to produce an

alternative implementation of ATL in Maude. Cheng et al. develop an operational semantics for the ATL bytecode [14, 15], which is used for translation validating axiomatic semantics of ATL transformations. Boronat also proposes an operational (big-step) semantics for ATL [9]. It differs from the work of Troya and Vallecillo in: 1) abstracting models (represented as nested objects), and 2) using a dedicated DSL to separate implementation for side effects of model transformations from encoded semantics.

Varró et al. implement a graph transformation engine in relational databases [49]. 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).

Ko et al. develop a general-purpose bidirectional model transformation language based on put-back-based semantics [31]. The language is formalized by primitives in a functional style. The formalization enables reasoning of typical bidirectional language properties (i.e. round-trip laws). In the same vein, He and Hu develop a logical framework to address the ambiguity and the shared node issues in bidirectional model transformation engines [27].

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 [28]. 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.

There are many approaches of model transformation language engineering based on graph grammars [1, 37]. We think that these approaches are different from our proposal in how the transformation is formalized: graph grammars define inductive relations to establish a correspondence between source and target models, whereas we define the transformation as a computable function. This difference results in distinct proof styles when reasoning about language properties (e.g. how to define the inductive principle).

Correctness for Model Transformations. Automatic theorem proving is one of the most popular approaches to ensure the correctness of MTs.

Black-box generation criteria like meta-model [25, 44] or requirements [26] coverage have been proposed to test the correctness of model transformations. However, the detection of type errors is not addressed. Cuadrado et al. present a static approach to uncover various type errors in ATL MTs, and use the USE constraint solver to compute an input model as a witness for each error [23].

Büttner et al. use Z3 SMT solver to verify the functional correctness of a declarative subset of the ATL [11]. 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 [12].

Oakes et al. statically verify ATL MTs by symbolic execution using DSLTrans [38]. 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.

Burgueño et al. syntactically calculating the intersection constructs used by the rules and contracts for fault localization of ATL MTs [10]. W.r.t. their approach, Cheng and Tisi aim at improving the localization precision by considering also semantic relations between rules and contracts [16]. This allows to produce smaller slices by semantically eliminating unrelated rules for the given bug, and to provide debugging clues that help understand the fault. Based on the same idea that explores semantic relations between rules and contracts, Cheng and Tisi also propose an incremental way to speed up the verification of ATL MTs [17].

One of the problems that prevents MT verification to scale is that complex MTs generate complex proof obligations that are not suitable for the state-of-the-art automatic theorem provers. Cheng and Tisi improve the scalability of the verification process by generating proof obligations using only transformation rules that could impact a postcondition, instead of the full transformation [18].

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 [13].

In [45], 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 [33]. 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 [40], 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 [24].

None of these works addresses explicitly the modularity of the specification for certifying a transformation engine, and for proof preservation through engine updates.

W.r.t. theorem proving, test-based approaches are more lightweight and popular with practitioners. For instance, specification-based test generation helps testing the correctness of MTs [26, 35]. Typically these approaches refer to a fixed semantics for the transformation engine, hence the test generator is in principle impacted by the transformation engine evolution. Studying MT test preservation w.r.t. to engine evolution is an interesting line for future work.

9 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. We exemplify an evolution mechanism for the specification by describing an extended specification for CoqTL that includes trace links. We show which user proofs are preserved in all these versions of CoqTL.

Our experiments show that using this interface often makes the user proofs longer. However all proofs written exclusively using the specification have the cru-

cial 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. Ab.Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software & Systems Modeling 14(2), 1003–1028 (2015)
2. Appel, A.W., Beringer, L., Chlipala, A., Pierce, B.C., Shao, Z., Weirich, S., Zdancewic, S.: Position paper: the science of deep specification. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 375(2104) (2017)
3. ATLAS Group: Specification of the ATL virtual machine. Tech. rep., Lina & INRIA Nantes (2005)
4. Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.M.: Barriers to systematic model transformation testing. Communications of the ACM 53(6), 139–143 (2010)
5. Benzaken, V., Contejean, E.: 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. pp. 249–261. ACM, Cascais, Portugal (2019)
6. Benzaken, V., Contejean, E., Keller, C., Martins, E.: A Coq formalisation of SQL's execution engines. In: 9th International Conference on Interactive Theorem Proving. pp. 88–107. Springer, Oxford, UK (2018)
7. 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)
8. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A Trusted Mechanised JavaScript Specification. In: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 87–100. ACM, San Diego, California, USA (2014)
9. Boronat, A.: Experimentation with a big-step semantics for ATL model transformations. In: 10th International Conference on Theory and Practice of Model Transformations. pp. 3–18. Springer, Marburg, Germany (2017)
10. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. IEEE Transactions on Software Engineering 41(5), 490–506 (2014)
11. 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)
12. 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)
13. 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)
14. 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)
15. Cheng, Z., Monahan, R., Power, J.F.: Formalised EMFTVM bytecode language for sound verification of model transformations. Software & Systems Modeling 17(4), 1197–1225 (2018)

16. Cheng, Z., Tisi, M.: A deductive approach for fault localization in ATL model transformations. In: 20th International Conference on Fundamental Approaches to Software Engineering. pp. 300–317. Springer, Uppsala, Sweden (2017)
17. Cheng, Z., Tisi, M.: Incremental deductive verification for relational model transformations. In: 10th IEEE International Conference on Software Testing, Verification and Validation. No. 379-389, IEEE, Tokyo, Japan (2017)
18. Cheng, Z., Tisi, M.: Slicing ATL model transformations for scalable deductive verification and fault localization. International Journal on Software Tools for Technology Transfer 20(6), 645–663 (2018)
19. Cheng, Z., Tisi, M., Douence, R.: CoqTL: a Coq DSL for rule-based model transformation. Software & Systems Modeling 19(2), 425–439 (2020)
20. Cheng, Z., Tisi, M., Hotonnier, J.: Certifying a rule-based model transformation engine for proof preservation. In: 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 297–307. ACM, Montreal, Canada (2020)
21. Chlipala, A., Delaware, B., Duchovni, S., Gross, J., Pit-Claudel, C., Suriyakarn, S., Wang, P., Ye, K.: The end of history? using a proof assistant to replace language design with library design. In: 2nd Summit on Advances in Programming Languages. pp. 1–15. No. 3, Asilomar, CA, USA (2017)
22. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A.: Kami: a platform for high-level parametric hardware specification and its modular verification. Proceedings of the ACM on Programming Languages 1(ICFP), 1–30 (2017)
23. Cuadrado, J.S., Guerra, E., de Lara, J.: Static analysis of model transformations. IEEE Transactions on Software Engineering 43(9), 868–897 (2016)
24. 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)
25. Fleurey, F., Baudry, B., Muller, P., Traon, Y.L.: Qualifying input test data for model transformations. Software & Systems Modeling 8(2), 185–203 (2009)
26. Guerra, E., Soeken, M.: Specification-driven model transformation testing. Software & Systems Modeling 14(2), 623–644 (2015)
27. He, X., Hu, Z.: Putback-based bidirectional model transformations. In: 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 434–444. ACM, FL, USA (2018)
28. Hidaka, S., Jouault, F., Tisi, M.: On additivity in transformation languages. In: 20th International Conference on Model Driven Engineering Languages and Systems. pp. 23–33. IEEE, Austin, TX, USA (2017)
29. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
30. Jouault, F., Bézivin, J.: KM3: A DSL for metamodel specification. In: 8th International Conference on Formal Methods for Open Object-Based Distributed Systems. pp. 171–185. Springer, Bologna, Italy (2006)
31. Ko, H.S., Zan, T., Hu, Z.: BiGUL: A Formally Verified Core Language for Putback-Based Bidirectional Programming. In: ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 61–72. ACM, St. Petersburg, FL, USA (2016)
32. 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)
33. Lano, K., Clark, T., Kolahdouz-Rahimi, S.: A framework for model transformation verification. Formal Aspects of Computing 27(1), 193–235 (2014)
34. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (Jul 2009)
35. López-Fernández, J.J., Guerra, E., de Lara, J.: Combining unit and specification-based testing for meta-model validation and verification. Information Systems 62(C), 104–135 (2016)
36. Martínez, S., Tisi, M., Douence, R.: Reactive model transformation with ATL. Science of Computer Programming 136, 1–16 (2017)
37. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic notes in theoretical computer science 152, 125–142 (2006)
38. 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)

39. Object Management Group: Query/view/transformation specification (ver. 1.3). http://www.omg.org/spec/QVT/1.3/ (2016)
40. 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)
41. Ringer, T., Yazdani, N., Leo, J., Grossman, D.: Adapting proof automation to adapt proofs. In: 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 115–129. ACM, Los Angeles, CA, USA (2018)
42. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
43. 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)
44. Sen, S., Baudry, B., Mottu, J.M.: Automatic model generation strategies for model transformation testing. In: 2nd International Conference on Model Transformations. pp. 148–164. Springer, Zurich, Switzerland (2009)
45. Stenzel, K., Moebius, N., Reif, W.: Formal verification of QVT transformations for code generation. Software & Systems Modeling 14, 981–1002 (2015)
46. Tisi, M., Cheng, Z.: CoqTL: An internal DSL for model transformation in Coq. In: 11th International Conference on Model Transformations. pp. 142–156. Springer, Springer, Uppsala, Sweden (2018)
47. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: 14th International Conference on Model Driven Engineering Languages and Systems. pp. 32–46. Springer, Springer, Wellington, New Zealand (2011)
48. Troya, J., Vallecillo, A.: A rewriting logic semantics for ATL. Journal of Object Technology 10(5), 1–29 (2011)
49. Varró, G., Friedl, K., Varró, D.: Implementing a graph transformation engine in relational databases. Software & Systems Modeling 5(3), 313–341 (2006)
50. Wagelaar, D.: Using ATL/EMFTVM for import/export of medical data. In: 2nd Software Development Automation Conference. Amsterdam, Netherlands (2014)
51. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: 14th International Conference on Model Driven Engineering Languages and Systems. pp. 623–637. Springer, Wellington, New Zealand (2011)
52. Weirich, S., Voizard, A., de Amorim, P.H.A., Eisenberg, R.A.: A specification for dependent types in Haskell. Proceedings of the ACM on Programming Languages 1(ICFP) (2017)