JOT
JOURNAL OF
OBJECT TECHNOLOGY

Automated Proof Tactics for Model Transformation

Julien Cohen[*], **Massimo Tisi**[†], **and Remi Douence**[‡]
[*]Nantes Université, LS2N (UMR CNRS 6004), France
[†]IMT Atlantique, LS2N (UMR CNRS 6004), France
[‡]IMT Atlantique, Inria, LS2N (UMR CNRS 6004), France

ABSTRACT When model-driven engineering is applied to critical systems, proving the correctness of model transformations (MT) assumes great importance. For instance, users may want a guarantee that the transformation does not produce models that break a critical post-condition, or that it preserves the semantics of the transformed model. Methods for automatic proof of MT correctness have shown their effectiveness for simple transformations and/or correctness properties. However, arbitrarily complex transformations and properties may require interactive theorem proving, that is a very costly activity.
In this paper we aim at simplifying the development of interactive proofs for MTs written in the CoqTL transformation language, by providing a set of automated proof tactics for MT certification. The tactics encode reasoning patterns that depend only on the semantics of the CoqTL engine, hence they are generally applicable to proofs on any CoqTL transformation. Each tactic is associated to a direction (forward if it derives facts about the target model given facts on the source model, backward if it derives facts on the source given facts on the target) and a subject (element or link, respectively if the tactic derives facts on elements or links). They are implemented as a set of lemmas and tactics for the Coq interactive theorem prover. In our experimentation, we show that these tactics allows us to write shorter and easier proofs. The source code for CoqTL and our tactics is provided at https://zenodo.org/records/11119867.

KEYWORDS Model Driven Engineering, Model Transformation, ATL, Formal Methods, Proof, Coq, CoqTL, Tactics.

1. Introduction

Model-driven engineering (MDE), i.e. software engineering centered on software models, metamodels and model transformations (MTs), is being used in critical systems, e.g. in the automotive industry (Selim et al. 2012), medical data processing (Wagelaar 2014), aviation (Berry 2008). MTs need to be formally verified, to guarantee that they will not produce faulty models in any case (e.g., models that break a critical post-condition of the transformation, or do not preserve the semantics of the original model).

Among the main formal methods for software verification, the application of theorem proving to transformation verification has been very limited. Automated theorem proving for MTs has shown its effectiveness for simple transformations and/or correctness properties (Büttner, Egea, Cabot, & Gogolla 2012; Büttner, Egea, & Cabot 2012; Cheng et al. 2015; Oakes et al. 2015). On the other hand, arbitrarily complex transformations and properties may require interactive theorem proving, that is generally very time consuming, and necessitates an expertise that typically transformation experts do not have.

Recent efforts try to simplify the application of interactive theorem proving to MTs. Among them, CoqTL (Tisi & Cheng 2018) has been proposed as a domain-specific language to simplify the definition of transformations in the Coq interactive theorem prover. However, proofs of non-trivial properties on CoqTL transformations remain long and complex. For instance, proving that a simple transformation of a graph of UML classes to a graph of relational tables preserves node reachability[1] has taken the CoqTL authors 1267 lines of proof code in previous work (Cheng & Tisi 2018b).

JOT reference format:
Julien Cohen, Massimo Tisi, and Remi Douence. *Automated Proof Tactics for Model Transformation*. Journal of Object Technology. Vol. 23, No. 3, 2024. http://dx.doi.org/10.5381/jot.2024.23.3.a1

[1] https://en.wikipedia.org/wiki/Reachability

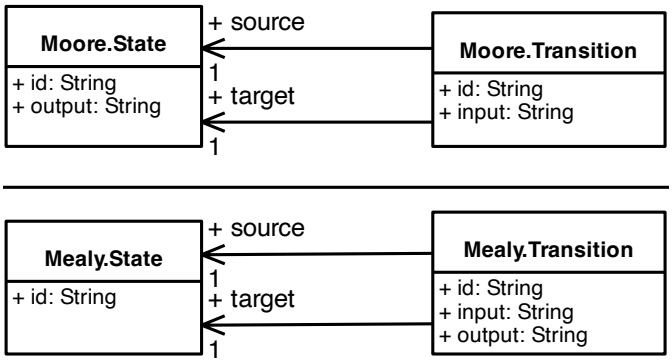

Figure 1 Simplified metamodels for state machines in Moore style (top) and in Mealy style (bottom).

Interactive theorem provers have mechanisms for the partial automation of proofs. For instance, Coq includes the Ltac language for the definition of automated proof tactics.[2] Users developing proofs for CoqTL transformations may develop specific tactics to reduce their proof size and effort. While the Coq tactic language is imperative, it has some fundamentally distinctive features, e.g. backtracking for proof search and unification. So engineering reusable code is much more complex for Coq proof scripts than imperative programs, and less covered in literature. In particular, the existence and structure of general automated proof tactics that can be reused across different MTs have not yet been studied.

This paper is a first step in the definition of reusable tactics for the partial automation of proofs on MTs. We define two general dimensions for model-transformation tactics: direction (forward or backward) and subject (element or link). We propose tactics for the CoqTL language, for each direction and target. The tactics are dependent only on the semantics of the CoqTL engine, and thus generally applicable. We implement these tactics in Ltac and discuss their application to existing transformations and theorems.

Section 2 introduces a running case and uses it for giving some necessary background on CoqTL. Section 3 describes the proposed tactics and their application to the motivating example. Section 4 discusses the application of the tactics on other proofs. Section 5 outlines related work and Section 6 concludes the paper.

2. Running Case

We start by introducing a simple running case, to motivate the paper and illustrate the proof tactics. As a sample transformation, we consider a simplified version of the transformation of UML state machines from the Moore style to the Mealy style. The transformation is intentionally very simple, so that proofs in Coq can be completely illustrated within this paper.

The metamodels in Fig. 1 are very simplified versions of the metamodel of UML state machines (for a full metamodel see the *Precise Semantics of UML State Machines* OMG specification).[3] The top and bottom metamodels represent respectively a machine in Moore and Mealy style, and their metaclasses have been renamed to highlight the style. A `Moore.State` has an `id` and an `output` that is produced when the state is entered. A `Moore.Transition` is always connected to a single `source` and `target` state, and is associated with an `id` and an `input` that triggers the transition. A `Mealy.State` has only an `id`. A `Mealy.Transition` is always connected to a single `source` and `target` state, and is associated with an `id`, an `input` that triggers the transition, and an `output` that is produced when the transition is triggered.

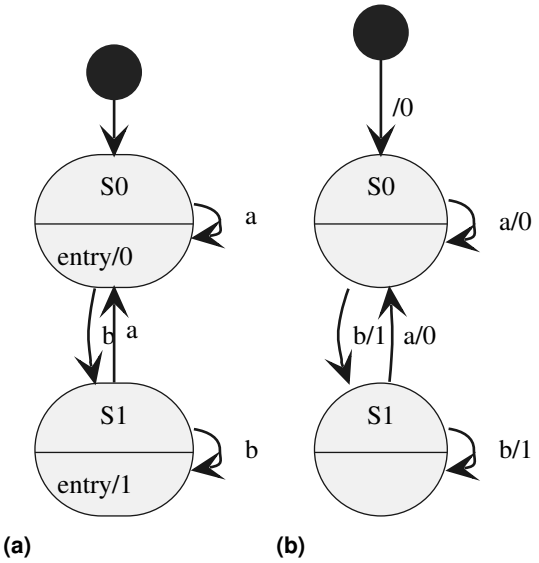

Figure 2 Sample statechart in Moore style (a) and in Mealy style (b)

As an example, Fig. 2 shows two state machines for a switch, respectively in Moore style (Fig. 2a) and Mealy style (Fig. 2b). Note that the two machines are equivalent, i.e. they produce the same output sequences for the same input sequences.

In MDE, MT languages are used to manipulate or translate models conforming to any metamodel. The user writes a declarative transformation logic, and the language engine applies it to standardized interfaces for object-oriented models and metamodels. In our running example we want to translate Moore machines into equivalent Mealy machines. For instance, we want to transform the model in Fig. 2a to the one in Fig. 2b. Note that this transformation is particularly simple for the source and target state machines have a very similar structure.

Listing 1 presents the transformation code in CoqTL (Tisi & Cheng 2018), a MT language implemented as an internal domain-specific language in the Coq theorem prover.[4] The transformation is a Coq `Definition`. It is made by two rules in a mapping style: one maps Moore `States` to Mealy `States` (lines 3-8), another one maps Moore `Transitions` to Mealy `Transitions` (lines 10-33). Each rule in CoqTL has a `from`

[2] Several other languages for defining automated proof tactics have been proposed by the Coq community: https://coq.inria.fr/refman/proofs/creating-tactics/index.html. Here we use Ltac because it was the main one until quite recently.

[3] https://www.omg.org/spec/PSSM/

[4] A mapping between each listing of this paper and the corresponding file is given on the web page of the artifact https://zenodo.org/records/11119867.

```
1  Definition Moore2Mealy :=
2    transformation [
3      rule "state"
4      from [Moore.State]
5      to [
6        ELEM "ms" ::= Mealy.State
7          fun _ _ s ⇒ return {| Mealy.State_id := s.(State_id) |}
8      ];
9
10     rule "transition"
11     from [Moore.Transition]
12     to [
13       ELEM "mt" ::= Mealy.Transition
14         fun _ m t ⇒
15           s ← Moore.getTransition_target m t;
16           return {|
17             Mealy.Transition_id := t.(Transition_id);
18             Mealy.Transition_input := t.(Transition_input);
19             Mealy.Transition_output := s.(State_output)
20           |}
21
22       LINK ::= Mealy.Transition_source
23         fun tls _ m moore_t mealy_t ⇒
24           t_source ← Transition_getSourceObject moore_t m;
25           res ← resolve tls "ms" Mealy.State (singleton t_source);
26           return {| src := mealy_t; trg := res |};
27
28       LINK ::= Mealy.Transition_target
29         fun tls _ m moore_t mealy_t ⇒
30           t_target ← Transition_getTargetObject moore_t m;
31           res ← resolve tls "ms" Mealy.State (singleton t_target);
32           return {| src := mealy_t; trg := res |}
33   ]
34 ]
```

Listing 1 Moore2Mealy transformation in CoqTL

section that specifies the input pattern (i.e. a tuple of model elements) to be matched in the source model. A boolean expression in Gallina (i.e. the functional language included in Coq) can be added as guard. CoqTL searches the source model for tuples that satisfy the guard, and fires the rule for all of them. 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 `ELEM`s and `LINK`s to create. The `ELEM` section includes standard Gallina code to instantiate the new element specifying the value of its attributes. The `LINK` section contains standard Gallina code to instantiate links connected to the previously instantiated element.

In Listing 1, the `state` rule (lines 3-8) matches any Moore state (line 4) and for each one of them, it constructs a corresponding Mealy state, labeled `ms` (line 6). The matched Moore state is referred to as `s`, and the new Mealy state is assigned the same `id` as the state `s` (line 7).

The `transition` rule (lines 10-33) transforms Moore transitions into Mealy transitions. The rule matches any transition (line 11), and for each match it creates a corresponding Mealy transition `mt` (line 13). The new transition is initialized by the following standard Gallina code (lines 15-20), in monadic (*a.k.a.* imperative) style. The source model and matched transition are referred respectively as `m` and `t` (line 14). We first retrieve the target state of the Moore transition, and name it `s` (line 15). The instantiated transition will have the `id` and `input` symbol of the matched transition `t` and the same `output` symbol of the state `s` (lines 16-20).

In order to connect the generated transition `mt` to its source and target Mealy states, two `LINK`s are created. The first `LINK` defines the source of the transition (line 22). In the computation of CoqTL `LINK`s we can navigate the list of *trace links* for that transformation execution, i.e. pairs of corresponding source and target elements instantiated by any rule of the transformation. We refer to the list of trace links with `tls`, to the source model with `m`, to the Moore transition matched by the rule with `moore_t` and to the generated Mealy transition as `mealy_t` (line 23). We first find the source state of the matched Moore transition (line 24). Then we *resolve* it, i.e. we find in the trace links the corresponding Mealy state (line 25). Finally, we return the pair of the Mealy transition and this resolved Mealy state (line 26). The target of the Mealy transition is similarly computed (lines 28-32).

The two simple rules in Moore2Mealy both have arity one,

```
1 Lemma state_element_fw : ∀ sm tm,
2     tm = execute Moore2Mealy sm →
3   ∀ (s:Moore.State),
4     In (Moore.State s) (modelElements sm) →
5     In (Mealy.State {|Mealy.State_id := s.(State_id)|})
6       tm.(modelElements).
```

Listing 2 Example of lemma in forward direction on Moore2Mealy

```
1 Lemma state_element_bw : ∀ sm,
2   ∀ (s:Mealy.State),
3     In (Mealy.State s) (Model.modelElements
4       (Semantics.execute Moore2Mealy sm)) →
5         exists s0, In (Moore.State s0)
6           (Model.modelElements sm) ∧
7             s = {| Mealy.State_id := s0.(State_id) |}.
```

Listing 3 Example of lemma in backward direction on the state rule

i.e. they both match only singleton tuples. CoqTL transformations however have often rules with higher arities, in order to match subgraphs of the source model. The arity of the transformation is defined as the maximum arity of the included rules.

Listing 2 shows a first simple theorem that predicates on the structure of the state machine produced by the transformation. The theorem states that for all the state machines `sm` in Moore style and `tm` in Mealy style (line 1), where `tm` is generated by applying the `Moore2Mealy` transformation to `sm` (line 2), if a state `s` is in the elements of `sm` (line 3-4), then a state with the same identifier will be in the elements of `tm` (lines 5-6). We say that this theorem is in *forward direction* since it states that a property of the source model entails a property of the target model. Intuitively, we know this theorem to hold because of the structure of the `state` rule, that for each Moore state produces a Mealy state with an identifier copied from the Moore state (line 8 in Listing 1). The proof is short but difficult to write because it requires familiarity with the full semantics of the transformation engine. While several proofs on CoqTL share similarities with it, they need to be written from scratch. We want to write reusable tactics that partially automate this kind of proofs.

Listing 3 shows another lemma on `Moore2Mealy`. For all the Moore state machines `sm` and Mealy states `s` (lines 1-2) that are elements of the result of the execution of Moore2Mealy on `sm` (lines 3-4), there exists a Moore state `s0` that is included in the elements of `sm` (lines 5-6), and has the same id as `s` (line 7). Intuitively we know this theorem to hold, for a similar argument of the previous theorem: Mealy states are only produced by the `state` rule, with identifiers that are always copied from identifiers of Moore states. However, this theorem is significantly different from the previous one, since now we are trying to prove a property of the source model by looking at the outcome in the target model. We say that the theorem is in *backward direction*. We aim at designing automated tactics that can address this kind of reasoning too.

The two previous lemmas are simple since they only predicate on the structure of the state machines (and only on their states). In Listing 4 we show a more complex theorem, that involves the semantics of state machines. We want to prove that the transformation produces Mealy machines that are semantically equivalent to the original Moore machines. Concretely, for all input sequences `i` and Moore machines `m` (line 2), the output of the execution of the Mealy machine (line 2) that is generated by the transformation (line 3) of `m`, is equal to the output of `m` on the same input sequence (line 4). While the proof

```
1 Theorem SemanticsPreservation :
2   ∀ i m, MealySemantics.execute
3     (Semantics.execute Moore2Mealy m) i
4       = MooreSemantics.execute m i.
```

Listing 4 Semantic preservation theorem for Moore2Mealy

of this theorem has been developed by us, it has analogies to the already mentioned proof of preservation of node reachability that took 1267 lines of proof in related work (Cheng & Tisi 2018a). We aim at designing automated tactics that can reduce this proof effort.

3. Proof tactics for CoqTL

In this paper we propose four automated tactics for CoqTL, encapsulating four key reasoning patterns for proofs on CoqTL transformations. We define two dimensions for tactics, *direction* and *subject*, and we organize the tactics along them:

We identify two *directions* for reasoning in CoqTL proofs:

- *Forward* direction when starting from assumptions on the source model we want to derive conclusions on the target model, i.e. the output of the transformation.
- *Backward* direction when starting from assumptions on the target model we want to derive conclusions on the source model.

Listings 2 and 3 are examples of lemmas in forward and backward direction. In a general case, even if a property is purely forward or purely backward, complex proofs may mix steps in forward and backward direction in different parts of the proof.

We also identify two *subjects* for reasoning in CoqTL proofs:

- *Element* reasoning aims at proving the existence (or not) of elements with certain properties, either in the source or target model.
- *Link* reasoning aims at proving the existence (or not) of links with certain properties, either in the source or target model.

While both Listings 2 and 3 are examples of element reasoning, the code repository joint to this article contains analogous examples of link reasoning. Again a given proof can mix element reasoning and link reasoning at different points.

3.1. Forward direction on elements

In this section we isolate a reasoning pattern for proofs on elements in the forward direction of a CoqTL transformation. We first illustrate the pattern by example, describing a manual proof that follows it. While the proof may seem complicated, the fact that it follows a general pattern allows us to automate a significant part of it. We encode the automation of the reasoning pattern as a proof tactic.

In Listing 5 we show a proof for `state_element_fw` that uses our reasoning pattern.[5] The proof starts (line 38) with the user applying the `in_modelElements_inv_m2m` lemma, that he previously defined and proved (lines 1-11, the proof is not shown). The lemma states that, in order to prove that a Mealy state `e` is included in the resulting elements of the Moore2Mealy transformation execution (line 10), it is sufficient to prove that the list of trace links of the Moore2Mealy execution (line 9) includes a trace link whose produced target state is `e` (line 6). In practice, by applying this lemma in the first step of the proof, the proof goal passes from a property about target elements to a property about the set of transformation trace links.

In a second proof step (line 39), the user applies a second lemma that he has previously proved, `in_compute_trace_inv_m2m` (lines 13-34). The lemma gives a sufficient condition to prove the inclusion of a certain trace link, connecting `s` in the source model to `res` in the target model, in the set of trace links produced by an application of Moore2Mealy (lines 27-33). The lemma states that to prove this inclusion, it is sufficient to prove 7 simpler properties:

1. Moore2Mealy has to include a rule `r` (line 15),
2. that rule `r` has to include an output pattern element `opu_el` (lines 16-21),
3. the source model has to include a set of Moore states `s` (line 22),
4. the guard of the rule `r` has to evaluate to true for `s` (line 23),
5. the length of `s` has to be 1 (line 24),
6. (step 6 predicates on the rule iterator, a feature of CoqTL that we do not discuss for brevity),
7. evaluating the output pattern element `opu_el` for `s` has to produce a target element `res` (line 26).

Finally in the last step of the reasoning pattern, the user needs to prove these 7 subgoals over the transformation and the source model (lines 40-46). We can not automate two of these proofs in general, since they depend on the specific transformation (steps 4 and 7). We can instead automate two other proofs (steps 5 and 6), and three others can be automated with some help from the user, who has to give as arguments to the tactic which rule, which pattern of that rule, or which hypothesis have to be used[6] (steps 1, 2 and 3):

1. the rule `r` can be directly selected by the user in Moore2Mealy as the rule named `state` (line 40),
2. the output pattern element `s` can be directly selected in Moore2Mealy as the element named `s` (line 41),
3. the theorem statement was assuming that a certain state was in the source model (line 4 in Listing 2), hence it is trivial to prove that a singleton set containing such a state is included in the source model (line 42),
4. since the guard of the `state` rule evaluates to true for all singletons containing a Moore state, this subgoal is automatically proved by the standard `reflexivity` Coq tactic (line 43),
5. it is obvious to prove that the length of a singleton is 1, after a few simplifications, by the standard `auto` Coq tactic (line 44),
6. the property on rule iterators is trivially proved too, since the `state` rule does not use iterators (line 45),
7. the target Mealy state generated from `s` is computed and checked for equality with the Mealy state from the theorem statement (line 5 in Listing 2), this is done automatically by the standard `reflexivity` Coq tactic (line 46).

To automate this reasoning pattern, we now define the `in_modelElements_singleton_fw_tac` tactic at the element level in forward direction. Listing 6 shows that the full proof in Listing 5 can be replaced by a simple call to the automatic tactic. In this case the tactic is particularly effective since the theorem can be proved by reasoning on the elements produced by a single transformation rule (i.e. the `state` rule), disregarding the rest of the model. In more complex cases the call to the tactic would only partially automate the proof, leaving some subgoals for the user to manually prove. This is a reason why dedicated tactics are hard to define: they must do enough work to progress, but not too much to avoid producing complex forms that may hamper the following steps.

Listing 7 shows the main part of the tactic. Figure 3 represents the application of the tactic to `state_element_fw` as a UML activity diagram. Actions are statements of the tactic. Object-flow links pass the global state of the proof (or subproof) at that point, as object tokens. For brevity we do not represent the full global state of the proof in each object token, but only the part that was updated from the action that generated the token.

The `in_modelElements_singleton_fw_tac` tactic proceeds as follows. It first checks that the statement to prove is in a specific form (line 7). The tactic is applicable to prove that an element (with certain properties) is included in the result of the execution of a transformation. Note that while the tactic assumes that the full transformation code is available, it does not assume anything about the element, and tries to compute what is possible based on the available knowledge.

If the statement to prove matches the supported pattern, the `in_modelElements_inv` lemma can be applied (line 8). That

[5] The reader which is not familiar with Coq builtin tactics can refer to the following *cheatsheet* : https://julesjacobs.com/notes/coq-cheatsheet/coq-cheatsheet.pdf

[6] We explain in Appendix A why the user has to give some information here.

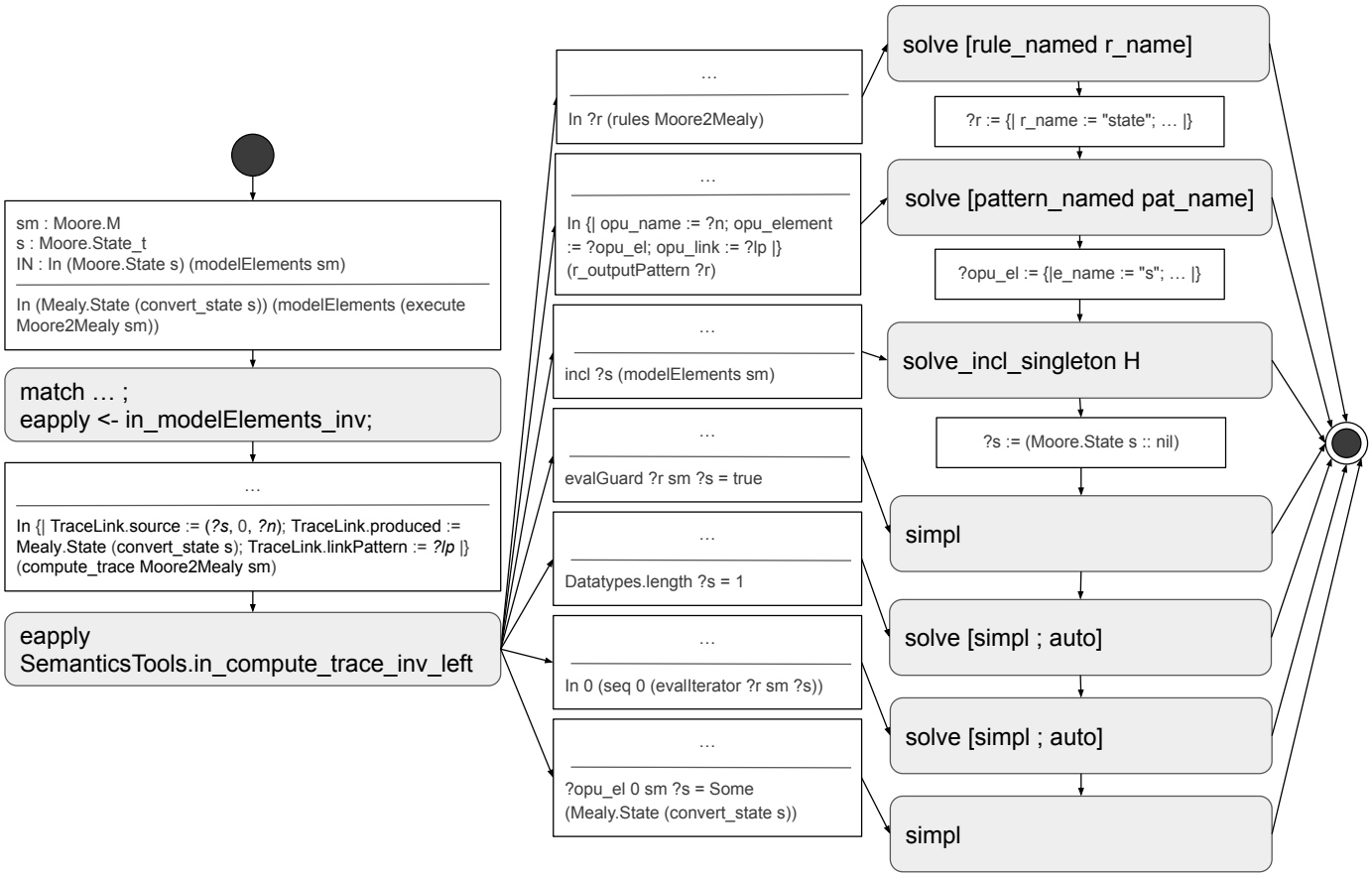

Figure 3 Execution of the `transform_element_fw_tac` tactic on `state_element_fw` (we omit unchanged contexts).

lemma[7] is a generalization of `in_modelElements_inv_m2m` from Listing 5, that we proved for any transformation (not only Moore2Mealy). We state that all the components of the trace link identified by `in_modelElements_inv` exist (line 9). As shown in Figure 3 this step changes the proof goal into an inclusion of a trace link in the computed set of transformation traces.

We then apply the `in_compute_trace_inv_left` tactic (line 11), a generalization of the `in_compute_trace_inv_m2m` from Listing 5, that we proved for any transformation. This lemma generates 7 subgoals to prove as in the Moore2Mealy example. Figure 3 shows each generated subgoal. The first and second subgoal require an explicit choice from the user of a specific rule and pattern element, by the `r_name` and `pat_name` parameters of the tactic (lines 18 and 22). These hints from the user allow us to avoid some over-automation that may worsen the proof state (when several assumptions with the correct form are available in the context, automatic tactics such as `eassumption` can select the wrong assumption, and backtracking cannot be performed in the general case, see Appendix A).

The third subgoal automatically proves the inclusion of the source pattern instance in the source model, given a hypothesis about the inclusion of an element in the source model, as the parameter `H` (line 25). Note that because of this, the `in_modelElements_singleton_fw_tac` tactic supports only rules of arity 1. We provide other tactics for rules of arity up to 5 (see the code repository). Analogous versions of the same tactic can be produced to support rules of higher arity.

Finally, the last 4 subgoals are handled (lines 30, 33, 36, 41) by calling standard Coq tactics that perform evaluation (`simpl`) and automatic proof search (`auto`). We expect automated proofs of steps 5 and 6 to succeed, given the simplicity of the subgoals in the cases considered by the tactics. Instead for steps 4 and 7 we only perform simplifications, in order to leave a manageable proof state for manually continuing the proof. For instance, in Listing 6 it is sufficient for the user to finalize the proof with a call to the `reflexivity` tactic (line 4), but proof continuations are case-dependent in general.

3.2. Forward direction on links

Reasoning on links is more complex than reasoning on single elements. Indeed, in order to create a link in CoqTL we need to provide the source element and target element for the link. Such elements are typically generated by different rules. This is the key capability that makes graph manipulation by MT languages effective. For instance, in Listing 1, lines 22-26 link a new transition to its source state. These lines do not state which

[7] See in `SemanticsTools.v` in the artefact.

```
Lemma in_modelElements_inv_m2m :
  ∀ (sm:Moore.M) e s n lp,
    In
      {|
        TraceLink.source := (s, 0,  n);
        TraceLink.produced := e ;
        TraceLink.linkPattern := lp
      |}
      ( compute_trace Moore2Mealy sm) →
    In e (execute Moore2Mealy sm).(modelElements).
Proof. ...  Qed.

Lemma in_compute_trace_inv_m2m (sm:Moore.M) :
  ∀ s n res l r opu_el,
      (*1*) In r Moore2Mealy.(rules) →
      (*2*) In {|
          opu_name := n ;
          opu_element := opu_el ;
          opu_link := l
        |}
        r.( r_outputPattern) →
      (*3*) incl s (modelElements sm) →
      (*4*) evalGuard r sm s = true →
      (*5*) length s = 1 →
      (*6*) In 0 ( seq 0 (evalIterator r sm s)) →
      (*7*) opu_el 0 sm s = Some res →
        In
          {|
            TraceLink.source := (s, 0,  n);
            TraceLink.produced := res ;
            TraceLink.linkPattern := l
          |}
          ( compute_trace Moore2Mealy sm).
Proof. ...  Qed.

Lemma state_element_fw:
   (** statement given above **)
Proof.
  eapply in_modelElements_inv_m2m.
  eapply in_compute_trace_inv_m2m.
  - (*1*) ChoiceTools.rule_named "state".
  - (*2*) ChoiceTools.pattern_named "s".
  - (*3*) apply ListUtils.incl_singleton; exact IN.
  - (*4*) reflexivity.
  - (*5*) simpl ; auto.
  - (*6*) simpl ; auto.
  - (*7*) reflexivity.
 Qed.
```

Listing 5 Proof for the `state_element_fw` lemma (Listing 2) (proofs of accessory lemmas are omitted).

rule will compute this state (we know it to be the `state` rule). The `resolve` function is responsible to retrieve this element by looking at all rule applications. This mechanism is generally called *implicit resolution* in MT languages. Concretely the `resolve` function analyzes a set of trace links connecting each generated tuple with its originating one.

Listing 9 shows the code of our `transform_link_fw` tactic for forward direction on links:

```
Lemma state_element_fw:
   (** statement given above **)
Proof.
  in_modelElements_singleton_fw_tac
                                  "state" "s" 0 IN ;
  reflexivity.
Qed.
```

Listing 6 Proof of `state_element_fw` (Listing 2) by tactic

```
Ltac in_modelElements_singleton_fw_tac
       r_name
       pat_name
       i
       H
  :=
  match type of H with ...  (** typecheck **) ;

  apply ← SemanticsTools.in_modelElements_inv ;
  eexists; exists i; eexists; eexists;

  eapply
     SemanticsTools.in_compute_trace_inv_left ;

  (* 7 goals *)
  [ | | | | | | ] ;

  [ (* 1. Fix the rule under concern following
       user hint *)
    solve [ChoiceTools.rule_named r_name]

  | (* 2. Fix the output pattern in the rule
       following user hint *)
    solve [ChoiceTools.pattern_named pat_name]

  | (* 3. Fix the source pattern instance *)
    ListUtils.solve_incl_singleton H

  | (* 4. The guard goal may rely on user
       expressions and can be arbitrary
       difficult to prove *)
    simpl

  | (* 5. arity *)
    solve [simpl ; auto]

  | (* 6. iteration counter *)
    solve [simpl ; auto ]

  | (* 7. The make_element goal relies on user
       expressions and can be arbitrary
       difficult to prove *)
    simpl
  ].
```

Listing 7 General tactic for FW results on elements

– The tactic is applicable only when we want to prove that a link (with certain properties) is included in the result of

```
1  Lemma in_modelLinks_inv
2   {tc:TransformationConfiguration} tr sm :
3    ∀ l, In l (execute tr sm).(modelLinks) ↔
4               exists s i n res lp,
5                 In
6                   {|
7                     source := (s, i, n);
8                     produced := res ;
9                     linkPattern := lp
10                  |}
11                  (compute_trace tr sm)
12                ∧ In
13                     l
14                     (apply_link_pattern
15                        (compute_trace tr sm)
16                        sm
17                        {|
18                          source := (s, i, n);
19                          produced := res ;
20                          linkPattern := lp
21                        |}).
```

Listing 8 Lemma `in_modelLinks_inv`

the transformation (checked at line 4).

- The `in_modelLinks_inv` lemma is applied (line 5). That lemma (shown in Listing 8) gives sufficient and necessary conditions for a link `l` to be included in the output of the transformation execution: there has to exist a suitable trace link in the result of the computation of the trace links (lines 4-11), and the link `l` has to be produced when the transformation processes that trace link (lines 12-21).
- We state that the suitable elements composing the trace link needed by `in_modelLinks_inv` actually exist and we ask Coq to find them automatically through the rest of the proof (line 6). This is done by declaring placeholders that will be filled (mainly by an automatic unification process) in the rest of the proof.
- The two sufficient and necessary conditions of `in_modelLinks_inv` are divided into two subgoals by the `split` step at line 8.
- (1st subgoal) We apply the `in_compute_trace_inv_left` lemma (line 10), that gives sufficient conditions for a trace link to be included in the list of trace links. The lemma is already used in Listing 7 and is a generalisation of the lemma `in_compute_trace_inv_m2m` (Listing 2). This yields 7 branches that are solved as in Listing 7 (lines 13-43).
- (2nd subgoal) The second subgoal introduced before refers to the user code which typically involves to `resolve` as discussed above, and is left to the user because it will generally involve business logic more than CoqTL machinery (empty branch line 45).

3.3. Backward direction on elements

Our `exploit_element_in_result` tactic for backward direction on elements is shown in Listing 12. In Listing 10 we show

```
1  Ltac transform_link_fw_tac_singleton
2     r_name pat_name i H :=
3
4   match type of H with ...  (** typecheck **) ;
5   apply ← SemanticsTools.in_modelLinks_inv ;
6   eexists ; exists i ; eexists ; eexists ; eexists ;
7
8   split ;
9
10  [ eapply
11      SemanticsTools.in_compute_trace_inv_left ;
12
13    (* 7 goals *)
14    [ | | | | | | ] ;
15
16
17    [ (* 1. Fix the rule under concern
18             following user hint *)
19      solve [ChoiceTools.rule_named r_name]
20
21    | (* 2. Fix the output pattern in the rule
22             following user hint *)
23      solve [ChoiceTools.pattern_named pat_name]
24
25    | (* 3. Fix the source piece (any size) *)
26      ListUtils.solve_incl_singleton H
27
28    | (* 4. The guard goal may rely on user
29             expressions and can be arbitrary
30             difficult to prove *)
31      simpl
32
33    | (* 5. arity *)
34      solve [simpl ; auto]
35
36    | (* 6. iteration counter *)
37      solve [simpl ; auto ]
38
39    | (* 7.The make_element goal relies on user
40           expressions and can be arbitrary
41           difficult to prove *)
42      simpl
43
44    ]
45
46  | ].
```

Listing 9 Tactic for forward direction on links

that its application automatizes most of the proof for the sample theorem in backward direction that we showed in Listing 3.

We briefly illustrate the main points of the tactic code in Listing 12:

- The tactic checks that there exists a hypothesis that states the inclusion of a certain element in the result of the execution of the transformation (lines 2-3). This is a requirement for the applicability of the tactic. Indeed, our backward tactics act on such hypotheses only (they do not act on the goal, contrarily to forward tactics).

```
1 Lemma state_element_bw :
2     (** statement given above **)
3 Proof.
4   intros s H.
5   TacticsBW.exploit_element_in_result H.
6   exists t0.
7   split; auto.
8 Qed.
```

Listing 10 Proof by tactic of `state_element_bw`

– We apply the lemma `in_modelElements_inv` to this hypothesis (line 7). We don't show the lemma here[8] but it is the analogous lemma of `in_modelLinks_inv` for elements. Note that the lemma is applied in left to right direction while in forward tactics it is applied in right to left direction (Listing 7, Section 3.1). Its application generates as new hypothesis the consequences of an element being included in the output of the transformation execution. One of these consequences is the existence of a corresponding trace link.
– We separate and name the consequences generated by `in_modelElements_inv` (line 8).
– Finally we execute a separate tactic, i.e. `exploit_in_trace`, to exploit what we now know about the existence of a trace link for this element (line 11).

The code of `exploit_in_trace` is shown in Listing 11.

– The tactic is applicable on an hypothesis that states that a certain trace link is among the ones computed by the transformation engine (lines 2-3).
– The first step (lines 7-12) applies `in_compute_trace_inv`,[9] which is the reverse counterpart of the `in_compute_trace_inv_left` lemma used in forward tactics (Listings 7 and 9), to generate as hypothesis the consequences of the existence of the trace link.
– The trace link has to be generated by one of the rules of the transformation. We call a tactic that performs a case analysis on the transformation rules and generates one subgoal per rule (line 16).
– We call a tactic that evaluates for each subgoal the guard of the corresponding rule (line 19). This tactic automatically filters out the rules that can not match, because this would contradict the information we know about the trace link.
– Steps 4.a-b (lines 22-30) use dedicated tactics[10] to unify the variables introduced at step 1 (lines 9-11) against the code of the rule selected at step 2.
– Since the pattern instance is encoded by a list, we transform (line 33) a hypothesis of the form `incl [e_1,...,e_n] sm.(modelElements)` into n hypothesis of the form `In e_i sm.(modelElements)` (`incl` denotes the inclusion while `in` denotes membership).

[8] See in `SemanticsTools.v` in the artefact.
[9] See in `SemanticTools.v` in the artifact.
[10] See `TacticsBW.v` for the tactics.

```
1  Ltac exploit_in_trace H :=
2   match type of H with
3    | In _ (compute_trace _ _) ⇒
4       (...)   (** generate fresh names **)
5
6       (* 1: inversion *)
7       apply → in_compute_trace_inv in H;
8       autounfold with tracelink in H;
9       destruct H
10        as (IN_ELTS & _ & r & IN_RULE & MATCH_GUARD
11            & IN_IT & opu & IN_OUTPAT & EV);
12      (* The _ because there is no
13          information here*)
14
15      (* 2: case analysis on the rules
16             in the transformation *)
17      In_rules_inv_tac IN_RULE;
18
19      (* 3: get rid of the rules that
20             cannot match *)
21      evalGuard_inv_tac MATCH_GUARD;
22
23      (* 4.a: unify the iteration number *)
24      In_evalIterator_inv_tac IN_IT;
25
26      (* 4.b.1 : unify the out-pattern with
27                  those of the selected rule *)
28      In_outputPattern_inv_tac IN_OUTPAT;
29
30      (* 4.b.2 : unification with the
31               evaluation of the out-pattern *)
32      makeElement_inv_tac EV;
33
34      (* 4.c : destruct incl to In *)
35      repeat ListUtils.explicit_incl IN_ELTS
36
37      (* Remark:
38         4.a, 4.b.(1-2) and 4.c are
39         independent; they can be switched
40         (except 4.b.2 that must occur
41         after 4.b.1) *)
42
43   | (...)   (** transform the shape of the
44                       hypothesis if needed **)
45   end.
```

Listing 11 Pivot tactic for backward direction

3.4. Backward direction on links

Our `exploit_link_in_result` tactic for backward direction on links is shown in Listing 13. The tactics is structurally similar to `exploit_element_in_result`, and leverages the same pivot tactic `exploit_in_trace`.

It contains an additional last step (line 14 of Listing 13) that calls the auxiliary tactic `exploit_In_apply_link` on a hypothesis introduced by `in_modelLinks_inv`. That hypothesis states that the application of the link part of the rule has succeeded on a given input. The tactic `exploit_In_apply_link` unfolds all the machinery of the transformation engine until we

```
Ltac exploit_element_in_result IN :=
  match type of IN with
  | In _ (execute _ _).(modelElements) ⇒
      (...)   (** generate fresh names **)

      (* 1: make the trace appear *)
      apply → in_modelElements_inv in IN;
      destruct IN as (s & i & n & p & IN);

      (* 2: exploit the trace *)
      exploit_in_trace IN
  end.
```

Listing 12 Tactic for backward direction on elements

```
Ltac exploit_link_in_result IN :=
  match type of IN with
  | In _ (execute _ _).(modelLinks) ⇒
      (...)   (** generate fresh names **)

      (* 1: make the trace appear *)
      apply → in_modelLinks_inv in IN;
      destruct IN as (? & ? & ? & ? & ? & IN & IN_L);

      (* 2: exploit the trace *)
      exploit_in_trace IN;

      (* 3: exploit link creation code *)
      exploit_In_apply_link IN_L
  end.
```

Listing 13 Tactics for backward direction on links

```
Lemma transform_class_fw :
  ∀ (cm : ClassModel)(rm : RelationalModel),
    rm = execute Class2Relational cm →
  ∀ name, In (ClassElement {| Class_name := name |})
    cm.(modelElements) →
  In (TableElement {| Table_name := name |})
    rm.(modelElements).
Proof.
  intros cm rm H ; subst.
  intros n H.
  TacticsFW.transform_element_fw_tac.
Qed.
```

Listing 14 Example of lemma with forward direction on Class2Relational

```
Lemma transform_class_bw :
  ∀ (cm : ClassModel)(rm : RelationalModel),
    rm = execute Class2Relational cm →
  ∀ id name,
   In (TableElement {| Table_name := name |})
    rm.(modelElements) →
   In (ClassElement {| Class_name := name |})
    cm.(modelElements).
Proof.
  intros cm rm H ; subst.
  intros nm H.
  TacticsBW.exploit_element_in_result H ; [].
  destruct t0 ; assumption.
Qed.
```

Listing 15 Example of lemma with backward direction on Class2Relational

can see the user code applied to all its parameters (the computed trace, the given pattern instance, the source model, and the produced element). After that, the user can use this information and focus on business code instead of machinery.

4. Experimentation

4.1. Using the tactics in proofs

We performed several proofs, to assess the applicability of the tactics to different theorems and transformations.

Structural theorems on Moore2Mealy. For the Moore2Mealy transformation we apply the tactic to 4 element-level and 4 link-level structural theorems, in forward and backward direction. The tactics were applied in the proof of each one of these theorems. In the upper part of Table 2 we show for each theorem the applied tactics, their direction and subject.

Structural theorems on Class2Relational. In order to assess if the tactics are effective for structural theorems on different transformations, we apply them to prove a set of theorems for an independent case study. We choose the Class2Relational transformation, that transforms class diagrams into relational schemas (e.g. to create tables that persist a set of domain classes).

In Listings 14 and 15 we show two lemmas at the element level on Class2Relational, in forward and backward direction, with their proofs. The proofs show that besides some manipulations for goals and hypothesis, the core of both proofs is automatized by the respective tactics.

For Class2Relational, we apply the tactic to 10 element-level theorems and 1 link-level theorem (lower part of Table 2). Three proofs required more than one tactic application. One proof required to mix a backward element-level tactic with a forward link-level tactic. For one element-level theorem we compare the proof with another that does not use the tactic, and we observe that the tactic reduces the proof steps from 21 to 3.

Semantic preservation on Moore2Mealy. To show the applicability of the tactics to complex proofs, we used them in the proof of semantic preservation on Moore2Mealy (Listing 4). The proof required 2461 lines of code (1198 lines of definitions of properties and lemmas, and 1263 lines of proofs) following the typical structure of bisimulation proofs. Forward tactics were called 5 times and backward tactics 8 times. The proof takes into account the full semantics of the state machine. Part of this semantics involves traversing the edges of the graph, as in the proof of reachability (Cheng & Tisi 2018b). Hence, while the two proofs are structurally similar, the first is much more complex.

	Library	Moore2Mealy	Class2Relational
Definition LOC	612	1353	997
Proof LOC	120	1263	323
# Calls to FW tactics	-	5	7
# Calls to BW tactics	-	8	12

Table 1 Development metrics

Other applications. To illustrate and validate further the use of our tactics, we provide some complementary tests either in simple situations (see `TestTacticsFW.v` and `TestTacticsBW.v`) or in variations of the main examples. For instance `Class2Relational_TUPLES` contains rules that match pairs of elements. The corresponding code is available at the URL given in the abstract.

4.2. Discussion and limitations

We start by giving some metrics to quantify the impact of our development (Table 1). In total our tactics library counts 612 lines of Coq definitions and 120 lines of proof. The Moore2Mealy use case counts 1353 lines of Coq definitions and 1263 lines of proof. In proofs on Moore2Mealy, forward tactics are called 5 times, backward tactics 8 times. The Class2Relational use case counts 997 lines of Coq definitions and 323 lines of proof. In proofs on Class2Relational, forward tactics are called 7 times, backward tactics 12 times.

During the implementation of the tactics, we also performed some refactoring of the CoqTL transformation engine to enable pivot lemmas and simplify tactic mechanisms. This refactoring may have contributed to shortening proofs in our experimentation, and may impact the comparison with previous work.

The main limitation of current tactics lies in the limited support to source-pattern instances of arbitrary sizes. While we currently support matching up to five elements per rule, we plan to implement some tactics of variable arity for higher counts.

Another limitation of our tactics is that when they fail, it is difficult for the user to understand why. We plan to improve error messages in future work by using a tactic language with static typing, such as Ltac2.

A more fundamental limitation is that, since the user is free to use any well-typed Coq/Gallina code in his rules, we cannot provide tactics that anticipate all the possibilities. This is a design choice of CoqTL. We plan to explore more powerful tactics by augmenting CoqTL with a more restricted expression language.

Finally, much of the tediousness in proving properties of MTs comes from the need to navigate the object-oriented graphs that constitute metamodels and models in MDE. For instance, proofs like Moore2Mealy are more concise if performed over an ad-hoc data structure for the state machine. The code repository contains an experimentation in such a sense (see the directory Moore2MealyALT). However, CoqTL is meant to be compatible with MDE specifications (e.g. from the OMG), and consequently directly applicable to software built with MDE tools.

5. Related Work

5.1. Simplifying the proof of transformation correctness

Several fully automated theorem proving approaches have been proposed for MTs (e.g. (Büttner, Egea, Cabot, & Gogolla 2012; Büttner, Egea, & Cabot 2012; Cheng et al. 2015; Oakes et al. 2015)). It would be possible to delegate the verification of simple goals/transformations from the interactive prover to one of these automated provers. However, this approach would be effective only when it is possible to identify simple subgoals that can be automatically proved. Our work differentiates in investigating tactics that can automate part of the reasoning for complex goals/transformation, leaving subgoals for manual proof.

In the related area of graph transformation, some work investigated specific proof strategies for semantic preservation (Giese et al. 2006; Hülsbusch et al. 2010; Dyck et al. 2019), e.g. based on bi-simulation. Our tactics are more general, since they can be applied to any property, structural or semantic, that predicates on the transformed model elements. On the other hand, more specific proof strategies may increase automation in the cases they cover.

Validity of a transformation w.r.t. constraints in the source or target model can be checked by alternative methods to theorem proving, e.g. by static analysis (Hildebrandt et al. 2012; Cuadrado et al. 2017) or weakest pre-condition computation (Clarisó et al. 2016). Again the applicability of our tactics is more general, e.g. the properties we check are not limited to the expressive power of a constraint language (e.g. OCL). CoqTL and our tactics can be used to prove any property that can be expressed in Coq[11].

5.2. Interactive theorem proving for MT

We list the related work on interactive theorem proving for MTs. To our knowledge (partial) proof automation has not been studied in these approaches. Our proof tactics are specific to the CoqTL language, but we believe that the paper may inspire the creation of similar proof tactics for these tools.

Yang et al. interactively verify that a particular MT, i.e. from AADL to TASM language, is semantic preserving (Yang et al. 2014). The approach is based on providing a translational semantics of both languages as timed transition systems in Coq and then reasoning on their equivalence.

Most previous works focus on giving a translational semantics of a MT language towards the target theorem prover. Generally they do not investigate a way to formally ensure that the semantics of the MT language has been axiomatized correctly in the back-end theorem prover. Calegari et al. encode ATL MTs and OCL contracts into Coq to interactively verify that the MT is able to produce target models that satisfy the given contracts (Calegari et al. 2011). In (Stenzel et al. 2015), a Hoare-style calculus is developed by Stenzel et al. in the KIV prover to analyze transformations expressed in (a subset of) QVT Op-

[11] Gallina is not Turing-complete, since it does not allow for non-terminating computation. However the class of computable functions in Gallina is very large (any function that can be shown to be total in ZFC with countably many inaccessibles can be defined in Gallina (Werner 2006)).

Theorem	Transformation	Applied tactics	Direction	Subject
state_element_fw	M2M	in_modelElements_singleton_fw_tac	FW	Elements
state_element_bw	M2M	exploit_element_in_result	BW	Elements
transition_element_bw	M2M	exploit_element_in_result	BW	Elements
transition_element_fw	M2M	in_modelElements_singleton_fw_tac	FW	Elements
source_link_fw	M2M	transform_link_fw_tac_singleton	FW	Links
target_link_fw	M2M	transform_link_fw_tac_singleton	FW	Links
source_link_bw	M2M	exploit_link_in_result exploit_in_trace	BW	Links
target_link_bw	M2M	exploit_link_in_result exploit_in_trace	BW	Links
Attribute_name_preservation_fw	C2R	in_modelElements_singleton_fw_tac	FW	Elements
c2r_monotonicity	C2R	exploit_element_in_result in_modelElements_singleton_fw_tac (×2)	BW + FW	Elements
Column_name_uniqueness	C2R	exploit_element_in_result (×2)	BW	Elements
transform_attribute_fw	C2R	in_modelElements_singleton_fw_tac	FW	Elements
transform_class_fw	C2R	in_modelElements_singleton_fw_tac	FW	Elements
transform_attribute_bw	C2R	exploit_element_in_result	BW	Elements
transform_class_bw	C2R	exploit_element_in_result	BW	Elements
Relational_name_definedness	C2R	exploit_element_in_result	BW	Elements
wf_stable	C2R	exploit_link_in_result	BW	Links
Relational_Column_Reference_definedness_1	C2R	exploit_element_in_result transform_link_fw_tac_singleton	BW + FW	Elements + Links
Table_name_uniqueness	C2R	exploit_element_in_result (×2)	BW	Elements

Table 2 Tactics applied in the proof of structural theorems on Class2Relational (C2R) and Moore2Mealy (M2M).

erational. UML-RSDS is a tool-set for developing correct MTs by construction (Lano et al. 2014). 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.

Poernomo and Terrell follow the classical approach in type theory to formally specify MTs as $\forall\exists$ types in interactive theorem provers (Poernomo & Terrell 2010). Their approach does not target any specific MT languages. In addition, although their work does not propose a generic MT engine, a corresponding executable MT program can be extracted once the MT is proved. The approach is further extended by Fernández and Terrell on using co-inductive types to encode bi-directional or circular references (Fernández & Terrell 2013).

Kezadri et al. defines the Coq4MDE framework to formally embed some key aspects of MDE in Coq (Hamiaz et al. 2014). While our understanding is that Coq4MDE is capable of embedding MT languages and enabling MT verification, no specific work has been proposed.

6. Conclusion and Future Work

In this paper we presented four reusable tactics for partially automatizing interactive proofs over CoqTL transformations. We observed that reasoning patterns in CoqTL proofs can be classified by direction (forward/backward) and subject (element/link). We proposed tactics for each category, and we tested them on two transformations and several theorems on structural and semantic properties. Since the tactics are based only on the semantics of the CoqTL engine, they are applicable to all our theorems, and replace a significant number of proof steps.

We plan several lines for future work:

- We will further evaluate the impact of tactics by applying them to transformations and proofs in an industrial setting, linked to the verification of transformations for digital twins.
- This paper focuses on simplifying proofs on the MT, but also simplifying the definition of the MT itself is an important concern. We plan to explore the definition of a compiler of languages like ATL and OCL to CoqTL.
- Among the other possible means for reducing the effort of developing proofs in CoqTL, we want to develop a library of useful lemmas including general properties of CoqTL transformations, like confluence and additivity (Hidaka et al. 2017).
- CoqTL transformations are computational specifications that can be evaluated to produce a target model from a source model. This computationality is generally beneficial for forward reasoning, (when an input is known, some proof steps can be performed by simple evaluation). A non-computational axiomatic semantics for CoqTL may be more abstract, and allow for shorter proofs when computationality is not used (e.g. in backward reasoning). We plan to define such a semantics and prove its equivalence to the computational one.

Acknowledgments

We thank Zheng Cheng for his support in the early phases of this work.

References

Berry, G. (2008). Synchronous design and verification of critical embedded systems using SCADE and Esterel. In *12th international workshop on formal methods for industrial critical systems* (p. 2-2). Berlin, Germany: Springer. doi: 10.1007/978-3-540-79707-4_2

Büttner, F., Egea, M., & Cabot, J. (2012). On verifying ATL transformations using 'off-the-shelf' SMT solvers. In *15th international conference on model driven engineering languages and systems* (p. 198-213). Innsbruck, Austria: Springer. doi: 10.1007/978-3-642-33666-9_28

Büttner, F., Egea, M., Cabot, J., & Gogolla, M. (2012). Verification of ATL transformations using transformation models and model finders. In *14th international conference on formal engineering methods* (pp. 198–213). Kyoto, Japan: Springer. doi: 10.1007/978-3-642-34281-3_16

Calegari, D., Luna, C., Szasz, N., & Tasistro, Á. (2011). A type-theoretic framework for certified model transformations. In *13th brazilian symposium on formal methods* (p. 112-127). Natal, Brazil: Springer. doi: 10.1007/978-3-642-19829-8_8

Cheng, Z., Monahan, R., & Power, J. F. (2015). A sound execution semantics for ATL via translation validation. In *8th international conference on model transformation* (p. 133-148). L'Aquila, Italy: Springer. doi: 10.1007/978-3-319-21155-8_11

Cheng, Z., & Tisi, M. (2018a). Slicing ATL model transformations for scalable deductive verification and fault localization. *International Journal on Software Tools for Technology Transfer*, *20*(6), 645–663.

Cheng, Z., & Tisi, M. (2018b). *Unreachability theorem for Class2Relational in CoqTL. https://github.com/atlanmod/coqtl/blob/65bbeab2010b093fc51bf60693aea45c2dabeff9/fr.inria.atlanmod.coqtl.coq/examples/Class2Relational/theorems/thm_unreachability_byInductionDef.v*.

Clarisó, R., Cabot, J., Guerra, E., & de Lara, J. (2016). Backwards reasoning for model transformations: Method and applications. *Journal of Systems and Software*, *116*, 113–132. doi: 10.1016/j.jss.2015.08.017

Cuadrado, J. S., Guerra, E., & de Lara, J. (2017). Static analysis of model transformations. *IEEE Transactions on Software Engineering*, *43*(9), 868-897. doi: 10.1109/TSE.2016.2635137

Dyck, J., Giese, H., & Lambers, L. (2019). Automatic verification of behavior preservation at the transformation level for relational model transformation. *Software & Systems Modeling*, *18*(5), 2937–2972.

Fernández, M., & Terrell, J. (2013). Assembling the proofs of ordered model transformations. In *10th international workshop on formal engineering approaches to software components and architectures* (p. 63-77). Rome, Italy: EPTCS. doi: 10.4204/EPTCS.108.5

Giese, H., Glesner, S., Leitner, J., Schäfer, W., & Wagner, R. (2006, October). Towards Verified Model Transformations. In D. Hearnden, J. Süß, B. Baudry, & N. Rapin (Eds.), *Proc. of the 3rd international workshop on model development, validation and verification (modeva), genova, italy* (pp. 78–93). Le Commissariat à l'Energie Atomique - CEA.

Hamiaz, M. K., Pantel, M., Combemale, B., & Thirioux, X. (2014). A formal framework to prove the correctness of model driven engineering composition operators. In *International conference on formal engineering methods.*

Hidaka, S., Jouault, F., & Tisi, M. (2017). On additivity in transformation languages. In *20th international conference on model driven engineering languages and systems* (pp. 23–33). Austin, TX, USA: ACM/IEEE.

Hildebrandt, S., Lambers, L., Becker, B., & Giese, H. (2012). Integration of Triple Graph Grammars and Constraints. In C. Krause & B. Westfechtel (Eds.), *Proceedings of the 7th international workshop on graph based tools (grabats 2012)* (Vol. 52, p. 1-12). EC-EASST.

Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., & Wehrheim, H. (2010). Showing full semantics preservation in model transformation - a comparison of techniques. In D. Méry & S. Merz (Eds.), *Integrated formal methods* (pp. 183–198). Berlin, Heidelberg: Springer Berlin Heidelberg.

Lano, K., Clark, T., & Kolahdouz-Rahimi, S. (2014). A framework for model transformation verification. *Formal Aspects of Computing*, *27*(1), 193-235. doi: 10.1007/s00165-014-0313-z

Oakes, B. J., Troya, J., Lúcio, L., & Wimmer, M. (2015). Fully verifying transformation contracts for declarative ATL. In *18th ACM/IEEE international conference on model driven engineering languages and systems* (p. 256-265). Ottawa, ON: IEEE. doi: 10.1109/MODELS.2015.7338256

Poernomo, I., & Terrell, J. (2010). Correct-by-construction model transformations from partially ordered specifications in Coq. In *12th international conference on formal engineering methods* (p. 56-73). Shanghai, China: Springer. doi: 10.1007/978-3-642-16901-4_6

Selim, G., Wang, S., Cordy, J., & Dingel, J. (2012). Model transformations for migrating legacy models: An industrial case study. In *8th european conference on modelling foundations and applications* (p. 90-101). Lyngby, Denmark: Springer. doi: 10.1007/978-3-642-31491-9_9

Stenzel, K., Moebius, N., & Reif, W. (2015). Formal verification of QVT transformations for code generation. *Software & Systems Modeling*, *14*, 981–1002.

Tisi, M., & Cheng, Z. (2018). CoqTL: An internal DSL for model transformation in Coq. In *11th international conference on model transformations.* Uppsala, Sweden.

Wagelaar, D. (2014). Using ATL/EMFTVM for import/export of medical data. In *2nd software development automation conference.* Amsterdam, Netherlands.

Werner, B. (2006, 10). Sets in types, types in sets. In (p. 530-546). doi: 10.1007/BFb0014566

Yang, Z., Hu, K., Ma, D., Bodeveix, J.-P., Pi, L., & Talpin, J.-P. (2014). From AADL to timed abstract state machines:

A verified model transformation. *Journal of Systems and Software*, *93*, 42 - 68. doi: https://doi.org/10.1016/j.jss.2014.02.058

About the authors

Julien Cohen is assistant professor at Nantes Université. He is particularly interested in formal verification of properties of program transformations.

Massimo Tisi is an associate professor in the Department of Automation, Informatics, and Production of IMT Atlantique (Nantes, France), and leader of the NaoMod team, at the LS2N laboratory (UMR CNRS 6004). His research interests revolve around software and system modeling, domain specific languages and generative software engineering. You can contact the author at massimo.tisi@imt-atlantique.fr.

Remi Douence is an associate professor at IMT-Atlantique and a member of the Inria team Gallinette (Coq theorem prover). His main research interests are related to formal aspects of programming languages.

A. Why the tactics cannot be fully automatic

Consider a situation where several assumptions can be selected to solve a current goal.

Example :

```
H1 : In e1 s
H2 : In e2 s
==============
In ?e s
```

This goal is solved by the tactic `eassumption`. Contrary to the `assumption` tactic, `eassumption` deals with goal containing an "evar". Not only applying `eassumption` results in solving the goal but it also instantiates the evar in all the sub-goals on the shelve. Here, `?e` will be instantiated by `e1` or `e2`, depending on which assumption has been selected by `eassumption`. Often, only one of the possible instantiations will enable to solve the remaining sub-goals.

The usual way to force `eassumption` to select the correct assumption is to use the backtracking mechanism of Coq: if the `eassumpion` takes place in a sequence of tactics which can fail, `eassumption` can try to select another assumption, and thus instantiate `?e` differently, hopefully leading to a success of the whole tactic.

However, in our situation, failure to instantiate evars with the correct value cannot be detected within the tactic. Indeed, our tactic does not completely solve the goal. It is designed to be used to progress in a proof, to handle CoqTL mechanisms, and after that the user has to solve the goals that are relevant to the logic behind his models and transformation.

Relying on backtracking to ensure that the correct assumption has been selected and that the evar has been correctly instantiated is thus not feasible and letting `eassumption` doing the wrong instantiation would lead the user in blocking situations difficult to diagnose.

For this reason, some information regarding the continuation of the proof need to be given by the user in the general case. We do this by asking the user to specify which assumptions are relevant for his use, by the means of a parameter of the tactic (see for instance tactic `in_modelElements_singleton_fw_tac` in Listing 7).