# A Deductive Approach for Fault Localization in ATL Model Transformations

Zheng Cheng, Massimo Tisi

# A Deductive Approach for Fault Localization in ATL Model Transformations

Zheng Cheng and Massimo Tisi

AtlanMod team (Inria, IMT Atlantique, LS2N), France
Email: {zheng.cheng, massimo.tisi}@inria.fr

**Abstract.** In model-driven engineering, correct model transformation is essential for reliably producing the artifacts that drive software development. While the correctness of a model transformation can be specified and checked via contracts, debugging unverified contracts imposes a heavy cognitive load on transformation developers. To improve this situation, we present an automatic fault localization approach, based on natural deduction, for the ATL model transformation language. We start by designing sound natural deduction rules for the ATL language. Then, we propose an automated proof strategy that applies the designed deduction rules on the postconditions of the model transformation to generate sub-goals: successfully proving the sub-goals implies the satisfaction of the postconditions. When a sub-goal is not verified, we present the user with sliced ATL model transformation and predicates deduced from the postcondition as debugging clues. We provide an automated tool that implements this process. We evaluate its practical applicability using mutation analysis, and identify its limitations.

## 1 Introduction

Model-driven engineering (MDE), i.e. software engineering centered on software models and their transformation, is widely recognized as an effective way to manage the complexity of software development. One of the most widely used languages for model transformation (MT) is the AtlanMod Transformation Language (ATL) [18]. Like several other MT languages, ATL has a relational nature, i.e. its core aspect is a set of so-called matched rules, that describe the mappings between the elements in the source and target model.

With the increasing complexity of ATL MTs (e.g., in automotive industry [25], medical data processing [29], aviation [6]), it is urgent to develop techniques and tools that prevent incorrect MTs from generating faulty models. The effects of such faulty models could be unpredictably propagated into subsequent MDE steps, e.g. code generation. Therefore, the correctness of ATL is our major concern in this research. Typically *correctness* is specified by MT developers using contracts [9–13, 19, 21, 23]. Contracts are pre/postconditions that express under which condition the MT is considered to be correct. In the context of MDE, contracts are usually expressed in OCL [22] for its declarative and logical nature.

In [12], Cheng et al. developed the *VeriATL* verification system to deductively verify the correctness of ATL transformations w.r.t. given contracts. VeriATL automatically generates the axiomatic semantics of a given ATL transformation in the Boogie intermediate

verification language [5], combined with a formal encoding of EMF metamodels [26] and OCL contracts. The Z3 automatic theorem prover [20] is used by Boogie to verify the correctness of the ATL transformation.

However, when a contract on the MT is not verified, current verification systems like VeriATL do not report useful feedback to help the transformation developers fix the fault. Consequently, manually examining the full MT and its contracts and reasoning on the implicit rule interactions become a time-consuming routine to debug MTs.

Because of the advancement in computer science in the last couple of decades (e.g. in the performance of satisfiability modulo theory - SMT - solvers), many researchers are interested in developing techniques that can partially or fully automate the localization of faults in software (we refer the reader to [24, 31] for an overview).

In this work, we argue that the characteristics of the considered programming language have a significant impact on the precision and automation of the fault localization. More precisely, we think that in MT languages like ATL, automated fault localization can be more precise because of the available static trace information, i.e. inferred information among types of generated target elements and the rules that potentially generate these types. This idea has recently been introduced in [8] using a conservative and syntactical approach. However, we believe that a deductive approach can fully exploit its potential.

Our deductive approach is based on a set of sound natural deduction rules. It includes 4 rules for the ATL language based on the concept of static trace information, and 16 ordinary natural deduction rules for propositional and predicate logic [16]. Then, we propose an automated proof strategy that applies these deduction rules on the input OCL postcondition to generate sub-goals. Each sub-goal contains a list of hypotheses deduced from the input postcondition, and a sub-case of the input postcondition to be verified. Successfully proving the sub-goals soundly implies the satisfaction of the input OCL postcondition. When a sub-goal is not verified, we exploit its hypotheses in two ways to help the user pinpoint the fault: (a) slicing the ATL MT into a simpler transformation context; (b) providing debugging clues, deduced from the input postcondition to alleviate the cognitive load for dealing with unverified sub-cases. Our fault localization approach has been implemented and integrated with VeriATL. We evaluate our approach with mutation analysis. The result shows that: (a) the guilty constructs are presented in the slice; (b) deduced clues assist developers in various debugging tasks (e.g. the elaboration of a counter-example); (c) the number of sub-goals that need to be examined to pinpoint a fault is usually small.

**Paper organization.** We motivate our work by a sample problem in Section 2. Section 3 illustrates our fault localization approach in detail. Evaluation is presented in Section 4, followed by discussion of the limitations identified in our approach. Section 5 compares our work with related research, and Section 6 draws conclusions and lines for future work.

## 2 Motivating Example

As our running example we use the *HSM2FSM* MT. *HSM2FSM* transforms hierarchical state machine (*HSM*) models to flattened state machine (*FSM*) models. Both models conform to the same metamodel (Fig. 1). However, classifiers in the two metamodels are distinguished by the *HSM* and *FSM* prefix. Specifically, a named *StateMachine* contains a set of labelled *Transitions* and named *AbstractStates*. Each *AbstractState* has a concrete type, which is either *RegularState*, *InitialState* or *CompositeState*. A *Transition* links a *source* to a *target AbstractState*. Moreover, *CompositeStates* are only allowed in the models of HSM, and optionally contain a set of *AbstractStates*.

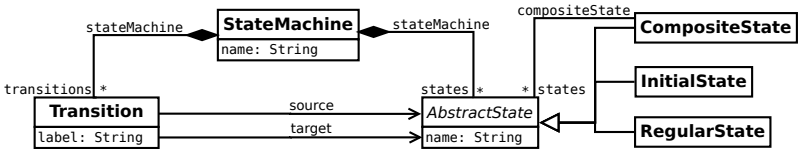


**Fig. 1.** The hierarchical and flattened state machine metamodel

### 2.1 Specifying OCL Contracts

We consider a contract-based development scenario where the developer first specifies correctness conditions for the to-be-developed ATL transformation by using OCL contracts. Let us first consider the contracts shown in Listing 1.1. The precondition *Pre1* specifies that in the input model, each *Transition* has at least one *source*. The postcondition *Post1* specifies that in the output model, each *Transition* has at least one *source*.

```
1 context HSM!Transition inv Pre1:
2   HSM!Transition.allInstances()->forAll(t | not t.source.oclIsUndefined())
3 ------------------------------------
4 context FSM!Transition inv Post1:
5   FSM!Transition.allInstances()->forAll(t | not t.source.oclIsUndefined())
```

**Listing 1.1.** The OCL contracts for HSM and FSM

### 2.2 Developing the ATL Transformation

Then, the developer implements the ATL transformation *HSM2FSM* (a snippet is shown in Listing 1.2[1]). The transformation is defined via a list of ATL matched rules in a mapping style. The first rule maps each *StateMachine* element to the output model (*SM2SM*). Then, we have two rules to transform *AbstractStates*: regular states are preserved (*RS2RS*), initial states are transformed into regular states when they are within a composite state (*IS2RS*). Notice here that initial states are deliberately transformed partially to demonstrate our

[1] Our HSM2FSM transformation is adapted from [9]. The full version can be accessed at: `https://goo.gl/MbwiJC`.

```
1  module HSM2FSM;
2  create OUT : FSM from IN : HSM;
3
4  rule SM2SM { from sm1 : HSM!StateMachine to sm2 : FSM!StateMachine ( name <− sm1.name ) }
5
6  rule RS2RS { from rs1 : HSM!RegularState
7               to rs2 : FSM!RegularState ( stateMachine <− rs1.stateMachine, name <− rs1.name ) }
8
9  rule IS2RS { from is1 : HSM!InitialState (not is1.compositeState.oclIsUndefined())
10              to   rs2 : FSM!RegularState ( stateMachine <− is1.stateMachine, name <− is1.name ) }
11
12 −− mapping each transition, that between two noncomposite states, of the source model into the target model.
13 rule T2TA { ... }
14
15 −− mapping each transition, whose source is a composite state, of the source model into the target model.
16 rule T2TB { ... }
17
18 rule T2TC {
19     from t1 : HSM!Transition, src : HSM!AbstractState, trg : HSM!CompositeState, c : HSM!InitialState
20     ( t1.source = src and t1.target = trg and c.compositeState = trg
21       and not src.oclIsTypeOf(HSM!CompositeState) )
22     to   t2 : FSM!Transition
23     ( label <− t1.label, stateMachine <− t1.stateMachine, source <− src, target <− c }
```

**Listing 1.2.** Snippet of the HSM2FSM model transformation in ATL

problem, i.e. we miss a rule that specifies how to transform initial states when they are **not** within a composite state. The remaining three rules are responsible for mapping the *Transitions* of the input state machine.

Each ATL matched rule has a *from* section where the source pattern to be matched in the source model is specified. An optional OCL constraint may be added as the guard, and a rule is applicable only if the guard evaluates to true on the source pattern. Each rule also has a *to* section which specifies the elements to be created in the target model. The rule initializes the attributes/associations of a generated target element via the binding operator (`<-`). An important feature of ATL is the use of an implicit *resolution* algorithm during the target property initialization. Here we illustrate the algorithm by an example: 1) considering the binding *stateMachine* `<-` *rs1.stateMachine* in the *RS2RS* rule (line 7 of Listing 1.2), its right-hand side is evaluated to be a source element of type *HSM!StateMachine*; 2) the resolution algorithm then resolves such source element to its corresponding target element of type *FSM!StateMachine* (generated by the *SM2SM* rule); 3) the resolved result is assigned to the left-hand side of the binding. While not strictly needed for understanding this paper, we refer the reader to [18] for a full description of the ATL language.

### 2.3 Formally Verifying the ATL Transformation

The source and target EMF metamodels and OCL contracts combined with the developed ATL transformation form a Hoare triple which can be used to verify the correctness of the ATL transformation, i.e. MM, Pre, Exec $\vdash$ Post. The Hoare triple semantically means that, assuming the axiomatic semantics of the involved EMF metamodels ($MM$) and OCL

```
1  context HSM!Transition inv Pre1: ...
2
3  rule RS2RS { ... }
4  rule IS2RS { ... }
5  rule T2TC { ... }
6
7  context FSM!Transition inv Post1_sub:
8  *hypothesis* var t0
9  *hypothesis* FSM!Transition.allInstances()->includes(t0)
10 *hypothesis* genBy(t0,T2TC)
11 *hypothesis* t0.source.oclIsUndefined()
12 *hypothesis* not (genBy(t0.source,RS2RS) or genBy(t0.source,IS2RS))
13 *goal* false
```

**Listing 1.3.** The problematic transformation scenario of the *HSM2FSM* transformation w.r.t. *Post1*

preconditions (*Pre*), by executing the developed ATL transformation (*Exec*), the specified OCL postcondition has to hold (*Post*).

In previous work, Cheng et al. have developed the VeriATL verification system that allows such Hoare triples to be soundly verified [12]. Specifically, the VeriATL system describes in Boogie what correctness means for the ATL language in terms of structural Hoare triples. Then, VeriATL delegates the task of interacting with Z3 for proving these Hoare triples to Boogie. The axiomatic semantics of EMF metamodels and the OCL language are encoded as Boogie libraries in VeriATL. These libraries can be reused in the verifier designs of MT languages other than ATL.

In our example, VeriATL successfully reports that the OCL postcondition *Post1* is not verified by the MT in Listing 1.2. This means that the transformation does not guarantee that each *Transition* has at least one *source* in the output model. Without any capability of fault localization, the developer then needs to manually inspect the full transformation and contracts to understand that the transformation is incorrect because of the absence of an ATL rule to transform *InitialStates* that are not within a *CompositeState*.

### 2.4 Our Goal: Localizing the Fault

In our running example, our proposed fault localization approach presents the user with two problematic transformation scenarios. One of them is shown in Listing 1.3. The scenario consists of the input preconditions (abbreviated at line 1), a slice of the transformation (abbreviated at lines 3 - 5), and a sub-goal derived from the input postcondition. The sub-goal contains a list of hypotheses (lines 7 - 12) with a conclusion (line 13).

The scenario in Listing 1.3 contains the following information, that we believe to be valuable in identifying and fixing the fault:

- *Transformation slicing.* The only relevant rules for the fault captured by this problematic transformation scenario are *RS2RS*, *IS2RS* and *T2TC* (lines 3 - 5).
- *Debugging clues.* The error occurs when a transition *t0* is generated by the rule *T2TC* (lines 8 - 10), and when the *source* state of the transition is not generated (line 11). In addition, the absence of the *source* for *t0* is due to the fact that none of the *RS2RS* and *IS2RS* rules is invoked to generate it (line 12).

From this information, the user could find a counter-example in the source models that falsifies *Post1* (shown in the top of Fig. 2): a transition $t_c$ between an initial state $i_c$ (which is not within a composite state) and a composite state $c_c$, where $c_c$ composites another initial state $i_c'$. This counter-example matches the source pattern of the *T2TC* rule (as shown in the bottom of Fig. 2). However, when the *T2TC* rule tries to initialize the *source* of the generated transition *t2* (line 23 in Listing 1.2), $i_c$ cannot be resolved because there is no rule to match it. In this case, $i_c$ (of type *HSM!InitialState*)) is directly used to initialize the *source* of *t2* (*t2.source* is expected to be a sub-type of *FSM!AbstractState*). This causes an exception of type mismatch, thus falsifying *Post1*. The other problematic transformation scenario pinpoints the same fault, showing that *Post1* is not verified by the MT also when *t0* is generated by *T2TA*.

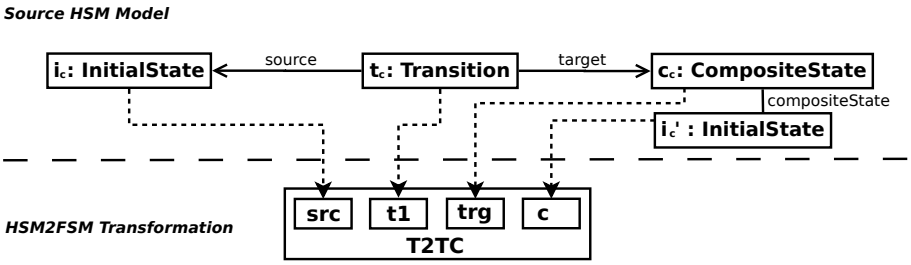


**Fig. 2.** Counter-example derived from Listing 1.3 that falsify *Post1*

In the next section, we describe how we automatically generate problematic transformation scenarios like the one shown in Listing 1.3.

## 3 Overview of Fault Localization for ATL by Natural Deduction and Program Slicing

The flowchart in Fig. 3 shows a bird's eye view of our approach to enable fault localization for VeriATL. The process takes the involved metamodels, all the OCL preconditions, the ATL transformation and one of the OCL postconditions as inputs. We require all inputs to be syntactically correct. If VeriATL successfully verifies the input ATL transformation, we directly report a confirmation message to indicate its correctness (w.r.t. the given postcondition) and the process ends. Otherwise, we generate a set of problematic transformation scenarios (as the one shown in Listing 1.3), and a proof tree to the transformation developer.

To generate problematic transformation scenarios, we first perform a systematic approach to generate sub-goals for the input OCL postcondition. Our approach is based on a set of sound natural deduction rules. The set contains 16 rules for propositional and predicate logic such as introduction/elimination rules for $\wedge$ and $\vee$ [16], but also 4 rules

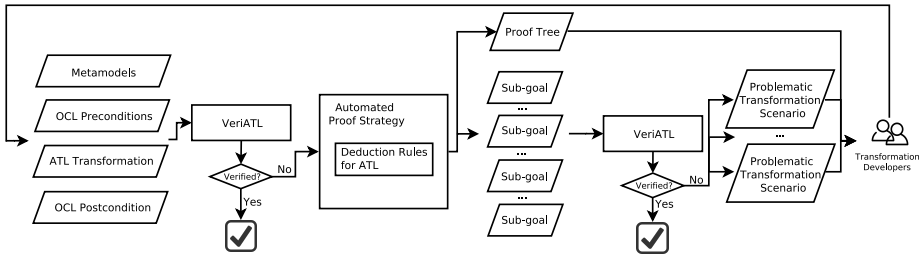


**Fig. 3.** Overview of providing fault localization for VeriATL

specifically designed for ATL expressions (e.g. rewriting single-valued navigation expression).

Then, we design an automated proof strategy that applies the designed natural deduction rules on the input OCL postcondition. Executing our proof strategy generates a proof tree. The non-leaf nodes are intermediate results of deduction rule applications. The leafs in the tree are the sub-goals to prove. Each sub-goal consists of a list of hypotheses and a conclusion to be verified. The aim of our automated proof strategy is to simplify the original postcondition as much as possible to obtain a set of sub-conclusions to prove. As a by-product, we also deduce new hypotheses from the input postcondition and the transformation as debugging clues.

Next, we use the trace information in the hypotheses of each sub-goal to slice the input MT into simpler transformation contexts. We then form a new Hoare triple for each sub-goal consisting of the semantics of metamodels, input OCL preconditions, sliced transformation context, its hypotheses and its conclusion.

We send these new Hoare triples to the VeriATL verification system to check. Notice that successfully proving these new Hoare triples implies the satisfaction of the input OCL postcondition. If any of these new Hoare triples is not verified by VeriATL, the input OCL preconditions, the corresponding sliced transformation context, hypotheses and conclusion of the Hoare triple are presented to the user as a problematic transformation scenario for fault localization. The Hoare triples that were automatically proved by VeriATL are pruned away, and are not presented to the transformation developer. This deductive verification step by VeriATL makes the whole process practical, since the user is presented with a limited number of meaningful scenarios.

Then, the transformation developer consults the generated problematic transformation scenarios and the proof tree to debug the ATL transformation. If modifications are made on the inputs to fix the bug, the generation of sub-goals needs to start over. The whole process keeps iterating until the input ATL transformation is correct w.r.t. the input OCL postcondition.

### 3.1 Natural Deduction Rules for ATL

Our approach relies on 20 natural deduction rules (7 introduction rules and 13 elimination rules). The 4 elimination rules (abbreviated by $X_e$) that specifically involve ATL are shown in Fig. 4. The other rules are common natural deduction rules for propositional and predicate logic [16]. Regarding the notations in our natural deduction rules:

- Each rule has a list of hypotheses and a conclusion, separated by a line. We use standard notation for typing (:) and set operations.
- Some special notations in the rules are $T$ for a type, $MM_T$ for the target metamodel, $R_n$ for a rule $n$ in the input ATL transformation, $x.a$ for a navigation expression, and $i$ for a fresh variable / model element. In addition, we introduce the following auxiliary functions: *cl* returns the classifier types of the given metamodel, *trace* returns the ATL rules that generate the input type (i.e. the static trace information)[2], *genBy(i,R)* is a predicate to indicate that a model element $i$ is generated by the rule $R$, *unDef(i)* abbreviates *i.oclIsUndefined()*, and *All(T)* abbreviates *T.allInstances()*.

$$\frac{x.a : T \qquad T \in cl(MM_T)}{x.a \in All(T) \vee unDef(x.a)}\ \text{TP}_{e1} \qquad \frac{x.a : Seq\ T \qquad T \in cl(MM_T)}{(|x.a| > 0 \wedge \forall i \cdot (i \in x.a \Rightarrow i \in All(T) \vee unDef(i))) \vee |x.a| = 0}\ \text{TP}_{e2}$$

$$\frac{T \in cl(MM_T) \qquad trace(T) = \{R_1, \ldots, R_n\} \qquad i \in All(T)}{genBy(i, R_1) \vee \ldots \vee genBy(i, R_n)}\ \text{TR}_{e1} \qquad \frac{T \in cl(MM_T) \qquad trace(T) = \{R_1, \ldots, R_n\} \qquad i : T \qquad unDef(i)}{\neg(genBy(i, R_1) \vee \ldots \vee genBy(i, R_n))}\ \text{TR}_{e2}$$

**Fig. 4.** Natural deduction rules that specific to ATL

Some explanation is in order for the natural deduction rules that are specific to ATL:

- First, we have two type elimination rules ($TP_{e1}$, $TP_{e2}$). $TP_{e1}$ states that every single-valued navigation expression of the type $T$ in the target metamodel is either a member of all generated instances of type $T$ or undefined. $TP_{e2}$ states that the cardinality of every multi-valued navigation expression of the type $T$ in the target metamodel is either greater to zero (and every element $i$ in the multi-valued navigation expression is either a member of all generated instances of type $T$ or undefined) or equal to zero.
- Second, we have 2 elimination rules for trace ($TR_{e1}$, $TR_{e2}$). These rules state that, given that the rules $R_1$,...,$R_n$ in the input ATL transformation are responsible to create model elements of type $T$ in the target metamodel, we may rightfully conclude that:
  - ($TR_{e1}$): every created element $i$ of type $T$ is generated by one of the rules $R_1$,...,$R_n$.
  - ($TR_{e2}$): every undefined element $i$ of type $T$ is not generated by any of the rules $R_1$,...,$R_n$.

**Soundness of natural deduction rules.** The soundness of our natural deduction rules is based on the operational semantics of the ATL language. Specifically, the soundness

[2] In practice, we fill in the *trace* function by examining the output element types of each ATL rule, i.e. the *to* section of each rule.

for type elimination rules $TP_{e1}$ and $TP_{e2}$ is straightforward. We prove their soundness by enumerating the possible states of initialized navigation expressions for target elements. Specifically, assuming that the state of a navigation expression $x.a$ is initialized in the form $x.a$<-$exp$ where $x.a$ is of a non-primitive type $T$:

- If *exp* is not a collection type and cannot be resolved (i.e. *exp* cannot match the source pattern of any ATL rules), then $x.a$ is *undefined*[3].
- If *exp* is not a collection type and can be resolved, then the generated target element of the ATL rule that matches *exp* is assigned to $x.a$. Consequently, $x.a$ could be either a member of *All(T)* (when the resolution result is of type $T$) or undefined (when it is not).
- If *exp* is of collection type, then all of the elements in *exp* are resolved individually, and the resolved results are put together into a pre-allocated collection *col*, and *col* is assigned to $x.a$.

The first two cases explain the two possible states of every single-valued navigation expressions ($TP_{e1}$). The third case explains the two possible states of every multi-valued navigation expressions ($TP_{e2}$).

The soundness of trace elimination rules $TR_{e1}$ is based on the surjectivity between each ATL rule and the type of its created target elements [9]: elements in the target metamodel exist if they have been created by an ATL rule since standard ATL transformations are always executed on an initially empty target model. When a type can be generated by executing more than one rule, then a disjunction considering all these possibilities is made for every generated elements of this type.

About the soundness of the $TR_{e2}$ rule, we observe that if a target element of type $T$ is undefined, then clearly it does not belong to *All(T)*. In addition, the operational semantics for the ATL language specifies that if a rule $R$ is specified to generate elements of type $T$, then every target elements of type $T$ generated by that rule belong to *All(T)* (i.e. $R \in trace(T) \Rightarrow \forall i \cdot (genBy(i, R) \Rightarrow i \in All(T))$) [12]. Thus, $TR_{e2}$ is sound as a logical consequence of the operational semantics for the ATL language (i.e. $R \in trace(T) \Rightarrow \forall i \cdot (i \notin All(T) \Rightarrow \neg genBy(i, R))$).

### 3.2 Automated Proof Strategy

A proof strategy is a sequence of proof steps. Each step defines the consequences of applying a natural deduction rule on a proof tree. A proof tree consists of a set of nodes. Each node is constructed by a set of OCL expressions as hypotheses, an OCL expression as the conclusion, and another node as its parents node.

[3] In fact, the value of *exp* is assigned to $x.a$ because of resolution failure. This causes a type mismatch exception and results in the value of $x.a$ becoming undefined (we consider ATL transformations in non-refinement mode where the source and target metamodels are different).

Next, we illustrate a proof strategy (Algorithm 1) that automatically applies our natural deduction rules on the input OCL postcondition. The goal is to automate the derivation of information from the postcondition as hypotheses, and simplify the postcondition as much as possible.

**Algorithm 1** An automated proof strategy for VeriATL

```
1: Tree ← {createNode({}, Post, null)}
2: do
3:     leafs ← size(getLeafs(Tree))
4:     for each node leaf ∈ getLeafs(Tree) do
5:         Tree ← intro(leaf) ∪ Tree
6:     end for
7: while leafs ≠ size(getLeafs(Tree))
8: do
9:     leafs ← size(getLeafs(Tree))
10:    for each node leaf ∈ getLeafs(Tree) do
11:        Tree ← elimin(leaf) ∪ Tree
12:    end for
13: while leafs ≠ size(getLeafs(Tree))
```

Our proof strategy takes one argument which is one of the input postconditions. Then, it initializes the proof tree by constructing a new root node of the input postcondition as conclusion and no hypotheses and no parent node (line 1). Next, our proof strategy takes two sequences of proof steps. The first sequence applies the *introduction* rules on the leaf nodes of the proof tree to generate new leafs (lines 2 - 7). It terminates when no new leafs are yield (line 7). The second sequence of steps applies the *elimination* rules on the leaf nodes of the proof tree (lines 8 - 13). We only apply type elimination rules on a leaf when: (a) a free variable is in its hypotheses, and (b) a navigation expression of the free variable is referred by its hypotheses. Furthermore, to ensure termination, we enforce that if applying a rule on a node does not yield new descendants (i.e. whose hypotheses or conclusion are different from their parent), then we do not attach new nodes to the proof tree.

### 3.3 Transformation Slicing

Executing our proof strategy generates a proof tree. The leafs in the tree are the sub-goals to prove by VeriATL. Next, we use the rules referred by the *genBy* predicates in the hypotheses of each sub-goal to slice the input MT into a simpler transformation context. We then form a new Hoare triple for each sub-goal consisting of the axiomatic semantics of metamodels, input OCL preconditions, sliced transformation context ($Exec_{sliced}$), its hypotheses and its conclusion, i.e. MM, Pre, $Exec_{sliced}$, Hypotheses $\vdash$ Conclusion.

If any of these new Hoare triples is not verified by VeriATL, the input OCL preconditions, the corresponding sliced transformation context, hypotheses and conclusion of the

Hoare triple are constructed as a problematic transformation scenario to report back to the user for fault localization (as shown in Listing 1.3).

Our transformation slicing is based on the independence among ATL rules [28]: each ATL rule is exclusively responsible for the generation of its output elements. Hence, when a sub-goal specifies a condition that a set of target elements should satisfy, the rules that do not generate these elements have no effects on the sub-goal. These rules can hence be safely sliced away.

## 4 Evaluation

In this section, we evaluate the practical feasibility and performance of our fault localization approach for the ATL language. The section concludes with a discussion of the obtained results and lessons learnt.

### 4.1 Research questions

We formulate two research questions to evaluate our fault localization approach:

**(RQ1)** Can our approach **correctly** pinpoint the faults in the given MT?
**(RQ2)** Can our approach **efficiently** pinpoint the faults in the given MT?

### 4.2 Evaluation Setup

Our evaluation uses the VeriATL verification system [12], which is based on the Boogie verifier (version 2.2) and Z3 (version 4.3). The evaluation is performed on an Intel 3 GHz machine with 8 GB of memory running the Windows operating system. VeriATL encodes the axiomatic semantics of the ATL language (version 3.7). The automated proof strategy and its corresponding natural deduction rules are currently implemented in Java.

To answer our research questions, we use the HSM2FSM transformation as our case study, and apply mutation analysis [17] to systematically inject faults. In particular, we specify 14 preconditions and 5 postconditions on the original HSM transformation from [9]. Then, we inject faults by applying a list of mutation operators defined in [8] on the transformation. We apply mutations only to the transformation because we focus on contract-based development, where the contract guides the development of the transformation. Our mutants are proved against the specified postconditions, and we apply our fault localization approach in case of unverified postconditions. We kindly refer to our online repository for the complete artifacts used in our evaluation [1].

### 4.3 Evaluation Results

Table 1 summarizes the evaluation results for our fault localization approach on the chosen case study. The first column lists the identity of the mutants[4]. The second and third columns record the unverified OCL postconditions and their corresponding verification time. The fourth, fifth, sixth and seventh columns record information of verifying sub-goals, i.e. the number of unverified sub-goals / total number of sub-goals (4th), average verification time of sub-goals (5th), the maximum verification time among sub-goals (6th), total verification of sub-goals (7th) respectively. The last column records whether the faulty lines ($L_{faulty}$, i.e. the lines that the mutation operators operated on) are presented in the problematic transformation scenarios ($PTS$) of unverified sub-goals.

**Table 1.** Evaluation metrics for the *HSM2FSM* case study

| | Unveri. Post. | | Sub-goals | | | | $L_{faulty} \in PTS$ |
|---|---|---|---|---|---|---|---|
| | ID | Veri. Time(ms) | Unveri. / Total | Avg. Time (ms) | Max Time (ms) | Total Time (ms) | |
| MT2 | #5 | 3116 | 3 / 4 | 1616 | 1644 | 6464 | True |
| DB1 | #5 | 2934 | 1 / 1 | 1546 | 1546 | 1546 | - |
| MB6 | #4 | 3239 | 1 / 12 | 1764 | 2550 | 21168 | True |
| AF2 | #4 | 3409 | 2 / 12 | 1793 | 2552 | 21516 | True |
| MF6 | #2 | 3779 | 0 / 6 | 1777 | 2093 | 10662 | N/A |
| | #4 | 3790 | 1 / 12 | 1774 | 2549 | 21288 | True |
| DR1 | #1 | 2161 | 3 / 6 | 1547 | 1589 | 9282 | - |
| | #2 | 2230 | 3 / 6 | 1642 | 1780 | 9852 | - |
| AR | #1 | 3890 | 1 / 8 | 1612 | 1812 | 12896 | True |
| | #3 | 4057 | 6 / 16 | 1769 | 1920 | 28304 | True |

First, we confirm that there is no inconclusive verification results of the generated sub-goals, i.e. if VeriATL reports that the verification result of a sub-goal is unverified, then it presents a fault in the transformation. Our confirmation is based on the manual inspection of each unverified sub-goal to see whether there is a counter-example to falsify the sub-goal. This supports the correctness of our fault localization approach. We find that the deduced hypotheses of the sub-goals are useful for the elaboration of a counter-example (e.g. when they imply that the fault is caused by missing code as the case in Listing 1.3).

Second, as we inject faults by mutation, identifying whether the faulty line is presented in the problematic transformation scenarios of unverified sub-goals is also a strong indication of the correctness of our approach. Shown by the last column, all cases satisfies the faulty lines inclusion criteria. 3 out 10 cases are special cases (dashed cells) where the faulty lines are deleted by the mutation operator (thus there are no faulty lines). In the case of *MF6#2*, there are no problematic transformation scenarios generated since all the

[4] The naming convention for mutants are mutation operator Add(A) / Del(D) / Modify(M), followed by the mutation operand Rule(R) / Filter(F) / TargetElement(T) / Binding(B), followed by the position of the operand in the original transformation setting. For example, *MB1* stands for the mutant which modifies the binding in the first rule.

sub-goals are verified. By inspection, we report that our approach improves the completeness of VeriATL. That is the postcondition (#2) is correct under *MF6* but unable to be verified by VeriATL, whereas all its generated sub-goals are verified.

Third, shown by the fourth column, in 5 out of 10 cases, the developer is presented with at most one problematic transformation scenario to pinpoint the fault. This positively supports the efficiency of our approach. The other 5 cases produce more sub-goals to examine. However, we find that in these cases each unverified sub-goal gives an unique phenomenon of the fault, which we believe is valuable to fix the bug. We also report that in rare cases more than one sub-goal could point to the same phenomenon of the fault. This is because the hypotheses of these sub-goals contain a semantically equivalent set of *genBy* predicates. Although they are easy to identify, we would like to investigate how to systematically filter these cases out in the future.

Fourth, from the third and fifth columns, we can see that each of the sub-goals is faster to verify than its corresponding postcondition by a factor of about 2. This is because we sent a simpler task than the input postcondition to verify, e.g. because of our transformation slicing, the Hoare triple for each sub-goal encodes a simpler interaction of transformation rules compared to the Hoare triple for its corresponding postcondition. From the third and sixth columns, we can further report that all sub-goals are verified in less time than their corresponding postcondition.

### 4.4 Limitations

**Language coverage.** In this work we consider a core subset of the ATL and OCL languages: (a) We consider the declarative aspect of ATL (matched rules) in non-refining mode, many-to-one mappings of (possibly abstract) classifiers with the default resolution algorithm of ATL. Non-recursive ATL helpers can be easily supported by inlining, and many-to-many mappings can be supported by extending our Boogie code generator. We also plan to investigate other features of ATL (e.g. lazy rules) to make our approach more general. (b) We support first-order OCL contracts and we plan to study more complex contracts in future work.

**Completeness of proof strategy.** We define the completeness of a proof strategy meaning that every elements of target types referred by each sub-goal and every rule that may generate them are correctly identified after applying the proof strategy. If not detected, an incomplete proof strategy could cause our transformation slicing to erroneously slice away the rules that the sub-goal might depend on. By manual inspection, we confirm the completeness of our proof strategy in our case study. However, our proof strategy is in generally incomplete because: (a) we might lack deduction rules to continue the derivation of the proof tree; (b) our current proof strategy lacks of a backtracking mechanism when it chooses an unsuitable deduction rule to apply. Our current solution is detecting incomplete cases and reporting them to the user. In practice we check whether every elements of target types referred by each sub-goal are accompanied by a *genBy* predicate (this indicates full derivation). In future, we plan to improve the completeness of our approach by adding other natural deduction rules for ATL and smarter automated proof strategies.

**Completeness of verification.** Although we confirmed that there are no inconclusive sub-goals in our evaluation, our approach could report inconclusive sub-goals in general due to the underlying SMT solver. We hope the simplicity offered by the sub-goals would facilitate the user in making the distinction between incorrect and inconclusive sub-goals. In addition, if an input postcondition is inconclusive, our approach can help users to eliminate verified sub-goals to find the source of its inconclusiveness.

**Threats to validity of evaluation.** We take a popular assumption in the fault localization community that multiple faults perform independently [31]. Thus, such assumption allows us to evaluate our fault localization approach in a one-postcondition-at-a-time manner. However, we cannot guarantee that this is the case for realistic and industrial MTs. We think classifying contracts into related groups could improve these situations.

**Scalability.** The main scalability issue of our approach is that a complex OCL postcondition (e.g. an OCL expression with deeply nested quantifiers) can potentially generate a big number of sub-goals and corresponding problematic transformation scenarios. Verifying and displaying all of them becomes impractical for transformation developers. Since sub-goals are meant to be manually examined by the user, a reasonable solution is allowing the user to specify a bound for the maximum number of unverified cases to generate. To improve scalability we are also investigating the possibility of verifying intermediate nodes in the proof tree, and stop applying deduction rules if they are verified.

**Usability.** Currently, our approach relies on the experience of transformation developer to interpret the deduced debugging clues. We think combining debugging clues with model finders would further help in debugging MT, e.g. by automatically generating the counter-examples [14].

## 5 Related Work

There is a large body of work on the topic of ensuring MT correctness [2]. To our knowledge our proposal is the first applying natural deduction with program slicing to increase the precision of fault localization in MT.

Büttner et al. use Z3 to verify a declarative subset of the ATL and OCL contracts [9]. 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 [10]. However, their tool does not guarantee that the newly generated counter example gives additional information than the previous ones. Oakes et al. statically verify ATL MTs by symbolic execution using DSLTrans [21]. 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.

Sánchez Cuadrado et al. present a static approach to uncover various typing errors in ATL MTs [14], and use the USE constraint solver to compute an input model as a witness for each error. Compared to their work, we focus on contract errors, and provide the user with sliced MTs and modularized contracts to debug the incorrect MTs.

Researchers have proposed several techniques that can partially or fully automate the localization of faults in software [24, 31]. Program slicing refers to detect a set of program statements which could affect the values of interest [27,30], which is used for fault localization of general programming languages. Few works have adapted this idea to localize faults in MTs. Aranega et al. define a framework to record the runtime traces between rules and the target elements these rules generated [4]. When a target element is generated with an unexpected value, the transformation slices generated from the runtime traces are used for fault localization. While Aranega et al. focus on dynamic slicing, our work focuses on static slicing which does not require test suites to exercise the transformation.

The most similar approach to ours is the work of Burgueño et al. on syntactically calculating the intersection constructs used by the rules and contracts [8]. W.r.t. their approach we aim at improving the localization precision by considering also semantic relations between rules and contracts. This allows us to produce smaller slices by semantically eliminating unrelated rules from each scenario. Moreover, we provide debugging clues to help the user better understand why the sliced transformation causing the fault. However, their work considers a larger set of ATL. We believe that the two approaches complement each other and integrating them is useful and necessary.

We implement our approach in Java. However, we believe that integrating our approach to interactive theorem provers/framework such as Coq [7] and Rodin [3] could be beneficial (e.g. drawing on recursive inductive reasoning). One of the easiest ways is through the Why3 language [15], which targets multiple theorem provers as its back-ends.

## 6 Conclusion and Future Work

In summary, in this work we confronted the fault localization problem for deductive verification of MT. We developed an automated proof strategy to apply a set of designed natural deduction rules on the input OCL postcondition to generate sub-goals. Each unverified sub-goal yields a sliced transformation context and debugging clues to help the transformation developer pinpoint the fault in the input MT. Our evaluation with mutation analysis positively supports the correctness and efficiency of our fault localization approach. The result showed that: (a) faulty constructs are presented in the sliced transformation, (b) deduced clues assist developers in various debugging tasks (e.g. generate counter-example), (c) the number of sub-goals that need to be examined to pinpoint a fault are usually small.

Our future work includes facing the limitations identified during the evaluation (Section 4.4). We also plan to investigate how our decomposition can help us in reusing proof efforts. Specifically, due to requirements evolution, the MT and contracts are under unpredictable changes during the development. These changes can invalidate all of the previous proof efforts and cause long proofs to be recomputed. We think that our decomposition of sub-goals would increase the chances of reusing verification results, i.e. sub-goals that are not affected by the changes.

## References


1. A deductive approach for fault localization in ATL model transformations [online]. available: `https://goo.gl/xssbpn` (2016)
2. Ab.Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software & Systems Modeling 14(2), 1003–1028 (2015)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer 12(6), 447–466 (2010)
4. Aranega, V., Mottu, J., Etien, A., Dekeyser, J.: Traceability mechanism for error localization in model transformation. In: 4th International Conference on Software and Data Technologies. pp. 66–73. Sofia, Bulgaria (2009)
5. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: 4th International Conference on Formal Methods for Components and Objects. pp. 364–387. Springer, Amsterdam, Netherlands (2006)
6. 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)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer, 1st edn. (2010)
8. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. IEEE Transactions on Software Engineering 41(5), 490–506 (2015)
9. 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)
10. 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)
11. 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)
12. 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)
13. Combemale, B., Crégut, X., Garoche, P., Thirioux, X.: Essay on semantics definition in MDE - an instrumented approach for model verification. Journal of Software 4(9), 943–958 (2009)
14. Cuadrado, J.S., Guerra, E., de Lara, J.: Uncovering errors in ATL model transformations using static analysis and constraint solving. In: 25th IEEE International Symposium on Software Reliability Engineering. pp. 34–44. IEEE, Naples, Italy (2014)
15. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: 22nd European Symposium on Programming. pp. 125–128. Springer, Rome, Italy (2013)
16. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press (2004)
17. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011)
18. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
19. Lano, K., Clark, T., Kolahdouz-Rahimi, S.: A framework for model transformation verification. Formal Aspects of Computing 27(1), 193–235 (2014)
20. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer, Budapest, Hungary (2008)

21. 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)
22. Object Management Group: The Object Constraint Language Specification (ver. 2.0). `http://www.omg.org/spec/OCL/2.0/` (2006)
23. 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)
24. Roychoudhury, A., Chandra, S.: Formula-based software debugging. Communications of the ACM (2016)
25. 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)
26. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse modeling framework. Pearson Education, 2nd edn. (2008)
27. Tip, F.: A survey of program slicing techniques. Tech. rep., Centrum Wiskunde & Informatica (1994)
28. Tisi, M., Perez, S.M., Choura, H.: Parallel execution of ATL transformation rules. In: 16th International Conference on Model-Driven Engineering Languages and Systems. pp. 656–672. Springer, Miami, FL, USA (2013)
29. Wagelaar, D.: Using ATL/EMFTVM for import/export of medical data. In: 2nd Software Development Automation Conference. Amsterdam, Netherlands (2014)
30. Weiser, M.: Program slicing. In: 5th International Conference on Software Engineering. pp. 439–449. IEEE, NJ, USA (1981)
31. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering Pre-Print(99), 1–41 (2016)