Incremental Deductive Verification for Relational Model Transformations

Zheng Cheng, Massimo Tisi

Incremental Deductive Verification for Relational Model Transformations

Zheng Cheng and Massimo Tisi
AtlanMod Team (Inria, IMT Atlantique, LS2N), France
Email: {zheng.cheng, massimo.tisi}@inria.fr

***Abstract*—In contract-based development of model transformations, continuous deductive verification may help the transformation developer in early bug detection. However, because of the execution performance of current verification systems, re-verifying from scratch after a change has been made would introduce impractical delays. We address this problem by proposing an incremental verification approach for the ATL model-transformation language. Our approach is based on decomposing each OCL contract into sub-goals, and caching the sub-goal verification results. At each change we exploit the semantics of relational model transformation to determine whether a cached verification result may be impacted. Consequently, less postconditions/sub-goals need to be re-verified. When a change forces the re-verification of a postcondition, we use the cached verification results of sub-goals to construct a simplified version of the postcondition to verify. We prove the soundness of our approach and show its effectiveness by mutation analysis. Our case study presents an approximate 50% reuse of verification results for postconditions, and 70% reuse of verification results for sub-goals. The user perceives about 56% reduction of verification time for postconditions, and 51% for sub-goals.**

I. 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 use of ATL MTs in safety-critical domains (e.g., in automotive industry [31], medical data processing [33], aviation [5]), 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. Typically correctness is specified by MT developers using contracts ([8]–[10], [12], [15], [20], [27], [28]). Contracts are pre/postconditions on the MT to express conditions under which the MT is considered to be correct. In the context of MDE, the contracts are usually expressed in the OMG's Object Constraint Language (OCL) for its declarative and logical nature.

In [12], we developed the *VeriATL* verification system to deductively verify the correctness of ATL transformations [18] w.r.t. given contracts. We also enabled automatic fault localization for VeriATL to facilitate debugging, based on natural deduction and program slicing. To illustrate VeriATL, let us consider a typical workflow in model transformation verification. A developer develops a model transformation *P*, and specifies a set of contracts (in terms of pre/postconditions) to ensure its correctness. Next, VeriATL is automatically executed at the back-end to facilitate fault localization. It decomposes each contract into a proof tree using the designed natural deduction rules and static analysis of the transformation *P*. The leaves of the proof tree are a set of verification sub-goals *S*. Then, the verifier reports to the developer of unverified postconditions and sub-goals. The developer continues to work on the model transformation *P*. Each change to *P* launches the back-end verifier, which verifies the new model transformation *P'* against the same set of contracts. The static analysis of *P'* causes VeriATL to decompose each of the contracts into a new set of sub-goals *S'* to verify, and the cycle goes on.

In this paper we argue that the practical applicability of VeriATL in such workflow strongly depends on the possibility to compute the impact of a change on the verification of postconditions/sub-goals. Intuitively, we want to accelerate the verification after the change by 1) re-using the verification results of postconditions/sub-goals that are not impacted by the change, 2) re-verifying only the impacted part of the postcondition.

Our contributions in this work are at two levels:

- On a fine-grained level, we aim to efficiently verify the set of sub-goals, by reusing the cached verification result of the sub-goals that are not impacted by the change. Specifically, we propose an algorithm that computes the relevant rules that potentially affect the verification result of each sub-goal. Then, the algorithm checks whether a given change is operated on these relevant rules. If not, we can safely reuse the cached verification result. We prove the soundness of this algorithm.
- On a coarse-grained level, we aim to efficiently verify each postcondition. We start from an algorithm which has the same idea of checking whether a given change affects the relevant rules for a postcondition. Moreover, if the cached verification result of a postcondition cannot be simply reused, we propose to verify a corresponding simplified postcondition. We compute each simplified postcondition by using the cached verification results of sub-goals to populate the proof tree of the original postcondition using three-valued logic (Kleene's strong

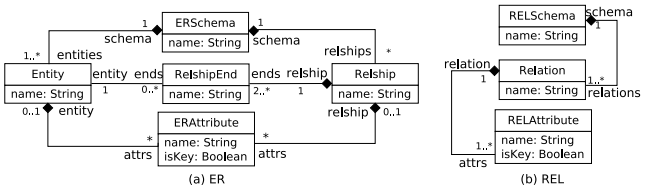

Figure 1. The Entity-Relationship and Relational metamodels

```
1  context ER!EREntity def: Pre1():
2  ER!EREntity->allInstances()->forAll(e | e.attrs->forAll(a1 |
3    e.attrs->forAll(a2 | a1<>a2 implies a1.name<>a2.name));
4
5  context ER!ERRelship def: Pre2():
6  ER!ERRelship->allInstances()->forAll(r | r.attrs->forAll(a1
7    r.attrs->forAll(a2| a1<>a2 implies a1.name<>a2.name));
8  ----------------------------------------------------------
9  context REL!RELRelation def: Post1():
10 REL!RELRelation->allInstances()->forAll(r | r.attrs->forAll(a1 |
11   r.attrs->forAll(a2 | a1<>a2 implies a1.name<>a2.name));
```

Listing 1. The OCL contracts for ER and REL

three-valued logic [19]). We argue that the simplified postcondition we synthesize represents the impacted part of the postcondition and is easier to verify than the original one. The soundness of this algorithm is also proved.

We show the effectiveness of our algorithms by mutation analysis. The evaluation of our case study shows an approximate 50% reuse of the verification result for postconditions, and 70% reuse of the verification result for sub-goals. Consequently, we gain about 56% reduction of verification time for postconditions, and 51% for sub-goals.

Paper organization. We give the background of our work and problem statement in Section II. Section III illustrates our incremental approach for model transformation verification in detail. Section IV proves its soundness. The evaluation is presented in Section V. Section VI compares our work with related research, and Section VII draws conclusions and lines for future work.

II. Background

To give a background of the ATL language and its verification system VeriATL that we developed, we use the *ER2REL* transformation as our running example. It translates Entity-Relationship (*ER*) models into relational (*REL*) models. Both the ER schema and the relational schema have commonly accepted semantics, and thus it is easy to understand their metamodels (Fig. 1): In a ER model, a named *ERSchema* contains a set of named *Entities* and *Relships*. Each *Relship* has two or more *RelshipEnds*, each pointing to an *Entity*. Both *Entities* and *Relships* can contain named *ERAttributes*, including a *key* attribute for each *Entity*. In a REL model, a *RELSchema* contains *Relations* that are made of *RELAttributes* (relations and attributes may typically correspond to tables and columns in a relational database).

A. 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. In Listing 1 we show three sample pre/postconditions on name uniqueness, that are common semantics of ER and relational schema. Two OCL preconditions specify that within an *Entity*, attribute names are unique (*Pre1*), and within a *Relship*, attribute names are unique (*Pre2*). One OCL postcondition requires that within a *Relation*, attribute names are unique (*Post1*).

B. Developing the ATL Transformation

Then, the developer implements the ATL transformation *ER2REL* (a snippet shown in Listing 2). The transformation is defined via a list of ATL matched rules in a mapping style. The first three rules map respectively each *ERSchema* element to a *RELSchema* element (*S2S*), each *Entity* element to a *Relation* element (*E2R*), and each *Relship* element to a *Relation* element (*R2R*). The remaining three rules generate a *RELAttribute* element for each *Relation* element created in the *REL* model.

Each ATL matched rule has a *from* section where the source pattern to be matched in the source model are 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 (target pattern). The rule initializes the attribute/association 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 element initialization. For example, for the binding *schema* <− *s.schema* in the *E2R* rule on line 10 of Listing 2, the *resolution* algorithm finds that the *S2S* rule matches the right hand side of the binding. Thus, its corresponding *RELSchema* element (created by the *S2S* rule) is returned by the algorithm, which is used to initialize left hand side of the binding[1].

C. 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 ⊢ Post. The Hoare-triple semantically means that, assuming the semantics of the involved EMF metamodels (*MM*) and OCL preconditions (*Pre*), by executing the developed ATL transformation (*Exec*), the specified OCL postcondition has to hold (*Post*).

In our previous work, we have developed the VeriATL verification system that allows such Hoare-triples to be soundly verified [12]. Specifically, the VeriATL system describes in Boogie [3] what correctness means for the ATL language in terms of structural Hoare-triples. Then, VeriATL delegates to Boogie the task of interacting with the SMT solver Z3 [26] for

[1]While not strictly needed for understanding this paper, we refer the reader to [18] for a full description of the ATL language.

```
module ER2REL;
create OUT : REL from IN : ER;

rule S2S {
  from s: ER!ERSchema
  to t: REL!RELSchema (name<−s.name)}

rule E2R {
  from s: ER!Entity
  to t: REL!Relation ( name<−s.name, schema<−s.schema) }

rule R2R {
  from s: ER!Relship
  to t: REL!Relation ( name<−s.name, schema<−s.schema) }

rule EA2A {
  from att: ER!ERAttribute, ent: ER!Entity ( att.entity=ent)
  to t: REL!RELAttribute ( name<−att.name, isKey<−att.isKey, relation<−ent )}

rule RA2A {
  from att: ER!ERAttribute, rs: ER!Relship ( att.relship=rs )
  to t: REL!RELAttribute ( name<−att.name, isKey<−att.isKey, relation<−rs ) }

rule RA2AK {
  from att: ER!ERAttribute, rse: ER!RelshipEnd ( att.entity=rse.entity and
       att.isKey=true )
  to t: REL!RELAttribute ( name<−att.name, isKey<−att.isKey,
       relation<−rse.relship )}
```

Listing 2. A snippet of the ER2REL model transformation in ATL

proving these Hoare-triples. 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 ATL MT shown in Listing 2. This means that the transformation does not guarantee that names of the attributes within each *Relation* are unique in the output model.

D. Localizing the Fault

To alleviate the cognitive load on developers investigating *why* a transformation is incorrect, in previous work we have provided VeriATL with fault localization capabilities by using natural deduction and program slicing [14]. Specifically, we proposed a set of sound natural deduction rules for the ATL language, including rules for propositional logic such as $\forall_i$ (introduction rule for $\forall$), and $\vee_e$ (elimination rule for $\vee$) [16], but also transformation-specific rules based on the concept of *static trace* (i.e. inferred information among types of generated target elements and the rules that potentially generate these types). We show the natural deduction rules that are specific to ATL in Appendix A.

Then, we proposed an automated proof strategy that applies the designed deduction rules on the input OCL postcondition to generate a proof tree. The leaves in the tree are the sub-goals to prove. Each sub-goal contains a list of hypotheses deduced from the original postcondition, and a sub-case of the original postcondition to be verified.

Next, we use hypotheses (in particular, hypotheses about static trace) in the sub-goals to slice the original MT into simpler transformation contexts. We then form a new Hoare-triple for each sub-goal consisting of the semantics of metamodels, original OCL preconditions, sliced transformation

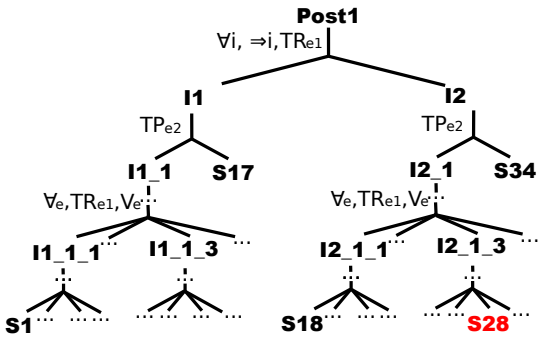

Figure 2. The proof tree w.r.t. *Post1*

```
context ER!ERSchema inv Pre1, Pre2

rule R2R { ... }
rule RA2AK { ... }

context FSM!RELRelation inv S28:
∗hypothesis∗ var r,a1,a2
∗hypothesis∗ REL!RELRelation.allInstances()−>includes(r)
∗hypothesis∗ r.attrs −>includes(a1)
∗hypothesis∗ r.attrs −>includes(a2)
∗hypothesis∗ a1 <> a2
∗hypothesis∗ genBy(r,R2R)
∗hypothesis∗ r.attrs.size() > 0
∗hypothesis∗ genBy(a1,RA2AK)
∗hypothesis∗ genBy(a2,RA2AK)
∗goal∗ a1.name <> a2.name
```

Listing 3. The problematic transformation scenario of the *ER2REL* transformation w.r.t. *Post1*

context (Exec_{trace}), its hypotheses and its conclusion, i.e. MM, Pre, Exec_{trace}, Hypotheses $\vdash$ Conclusion. We send these new Hoare-triples to VeriATL to check. Notice that successfully proving these new Hoare-triples implies the satisfaction of the original OCL postcondition. If any of these new Hoare-triples is not verified, the original 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 developer for fault localization.

In our running example, our fault localization approach generates the user with a proof tree and three problematic transformation scenarios. The proof tree (Fig. 2) is shortened for presentation. We also label the edges of the proof tree with the applied deduction rules for understandability. For demonstration purpose in Listing 3 we only show one of the problematic transformation scenarios (*S28* in Fig. 2).

The scenario consists of the original preconditions (abbreviated at the top), a slice of the transformation (abbreviated in the middle) and a sub-goal with the derived hypotheses and conclusion from the original postcondition (at the bottom). Notice that the *genBy(i, R)* predicates used in the hypotheses come from the application of our natural deduction rules. They specify that a model element *i* is generated by the rule *R*. This kind of hypotheses in the sub-goals help us slice the original MT into simpler transformation contexts.

The scenario in Listing 3 contains the following informa-

tion, 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 *R2R* and *RA2AK* (lines 3 - 4).
- *Debugging clues.* The error occurs when a Relation *r* is generated by the rule *R2R* (line 12), and when its *attributes* include two unique elements *a1* and *a2* (lines 9, 10 and 11) that are generated by the rule *RA2AK* (lines 14 and 15).

By analyzing the problematic transformation scenario in Listing 3, the transformation developer observes that two *Entities* linked by the same *Relship* in the source model might have the key attribute with the same name. This would cause their corresponding *RELAttributes*, generated by the *RA2AK* rule, to have the same name, and thus falsifying *Post1*.

E. Problem Statement

In our view, fixing bugs is an interactive process that involves several changes to the model transformation rules (e.g. add, delete and modify rule). Each of these changes requires VeriATL to re-verify the contracts and generated sub-goals. Depending on the engaged task, the developers might expect VeriATL to provide prompt re-verification result of either postconditions, or sub-goals, or both. For example:

- Prompt re-verification of postconditions would be beneficial when the user wants to know whether all contracts become verified after the fix;
- Prompt re-verification of sub-goals would be necessary when the bug is not yet fixed and the user needs the fault localization capability.

A delay in any of these cases can lead to wasted effort. Therefore, our quest in this work is to investigate whether incremental verification allows VeriATL to meet these expectations of the developer. By incremental verification, we mean the approach of caching the verification result of postconditions/sub-goals, determine when the cache can be reused, and then incrementally re-verify the postconditions/sub-goals that are not reused. Moreover, when incrementally re-verifying the postconditions, we aim to reduce their complexity by constructing simplified postconditions to verify.

For example, our approach aims to determine that, when modifying the *RA2A* rule, the verification result of the sub-goal *S28* for *Post1* can be reused; or that when modifying the *S2S* rule, the verification result of *Post1* does not change; or that when modifying the guard of the *R2R* rule, we can verify the simplified postcondition *I2* shown in Listing 4 instead of the postcondition *Post1*.

```
1  context ER!ERSchema inv Pre1, Pre2
2
3  rule E2R { ... } rule R2R { ... }
4  rule EA2A { ... } rule RA2A { ... } rule RA2AK { ... }
5
6  context REL!RELRelation inv I2:
7  *hypothesis* var r,a1,a2
8  *hypothesis* REL!RELRelation.allInstances()->includes(r)
9  *hypothesis* r.attrs->includes(a1)
10 *hypothesis* r.attrs->includes(a2)
11 *hypothesis* a1 <> a2
12 *hypothesis* genBy(r,R2R)
13 *goal* a1.name <> a2.name
```

Listing 4. Verifying the simplified postcondition of *Post1*

III. Incremental Verification for VeriATL

Before diving into the details of our algorithms for incremental verification for VeriATL, we first introduce some terminology:

- Initial transformation *P*. The model transformation before modification.
- Evolved transformation *P'*. The model transformation after modification.
- Transition operators *op*. We define a list of transition operators that modifies P to P' (Section IV-A).
- Proof Tree: as described in Section II-D, we have designed an automated proof strategy that applies the designed natural deduction rules on the input OCL postcondition. The goal is to deduce more information from the postcondition as hypotheses, and simplify the postcondition as much as possible. Executing our proof strategy generates a proof tree. The non-leaf nodes are intermediate results of rule applications. The leaves in the tree are the sub-goals to prove. Each sub-goal consists of a list of hypotheses and a conclusion to be verified. Dependencies among these sub-goals are shown by the edges in the tree.
- Static trace information $T : SubGoal \times MT \rightarrow \{Rule\}$, which returns a set of ATL matched rules that possibly generate the types for the target elements referred by each sub-goal. This can be obtained by statically analyzing the hypotheses of each sub-goal. For example, for the sub-goal *S28* shown in Listing 3, T(S28, ER2REL)={R2R,RA2AK}.
- Cache of postconditions $C_1 : MT \times OCL \rightarrow Bool$, which stores the verification result of each postcondition of the given model transformations. In practice, we compute an unique identifier for each model transformation, which is used in our caching mechanism. Two syntactically identical model transformations have the same identifier.
- Cache of sub-goals $C_2 : MT \times OCL \times SubGoal \rightarrow Bool$, which stores the verification result of each sub-goals for each postcondition of the given model transformations.

The incremental verification for VeriATL is supported by two algorithms (Algorithm 1 and Algorithm 2). Both algorithms accept 4 arguments: an initial transformation *P*, its evolved transformation *P'*, the transition operator *op* that changes P to P' and the postcondition *post* to be (re-)verified.

If the evolved transformation P' is invalid, e.g. it contains conflicting rules (two ATL matched rules conflict when a set of source elements can match the source pattern of both rules). or is syntactically incorrect, then the incremental verification is stopped (and is restarted when the developer constructs a

valid transformation by further modifications).

(Re-)verify sub-goals. Algorithm 1 is specialized to efficiently fill in and reuse from the cache of sub-goals. It first decomposes the *post* in P' by using a set of natural deduction rules we designed. The result of this decomposition is a proof tree, where the leaf nodes in the tree are the sub-goals we are interested in.

Next, we check if any sub-goal is impacted by the change, by computing the intersection between the relevant rules of a sub-goal and the rule that the transition operator performed on ($\delta_r(op)$). Here, the relevant rules of a sub-goal are captured by the static trace information referred by the sub-goal (*T(s, P')*).

Thus, the verification result of a sub-goal can be reused only when it has been cached, and none of its relevant rules are touched by the transition operators. Otherwise, its re-verification is expected (lines 2 - 5). In practice, we consider a given sub-goal is cached when a sub-goal with the exact match of hypotheses and conclusion of the given sub-goal is found in the cache.

Algorithm 1 Incremental verification algorithm for sub-goals (P, P', op, post)

1: **for** each $s \in$ leaves(decompose(P', post)) **do**
2: **if** $s \in$ dom(C_2[P][post]) $\wedge$ T(s, P') $\cap$ $\delta_r(op) = \emptyset$ **then**
3: C_2[P'][post][s] $\leftarrow$ C_2[P][post][s]
4: **else**
5: C_2[P'][post][s] $\leftarrow$ verify(s, P')
6: **end if**
7: **end for**

(Re-)verify postconditions. Algorithm 2 is specialized to efficiently verify the postconditions. It can be summarized as: *when verifying a postcondition, reuse the cached postcondition if possible and verify the **simplified** postcondition when necessary*.

The first task in Algorithm 2 is to determine if the cache of the postcondition can be reused:

- First, we decompose the sub-goals of *post* in P and P', and then compute the relevant rules in P and P' that potentially affect the verification result of *post* as the union of the static trace information referred by the sub-goals (lines 3 - 4).
- Next, we check the intersection between the relevant rules for *post* and the rule that the transition operator operated on ($\delta_r(op)$). Thus, the verification result of a postcondition can be reused only when it has been cached, and its sub-goals are referring the same set of relevant rules in P and P', and none of these rules are touched by the transition operators (lines 5 - 6).

If the cache of the postcondition cannot be reused, we label (populate) the whole proof tree for *post* in P' with verification result (*Tree'*$_v$), using three-valued logic [19] and the cached verification result for sub-goals (lines 8):

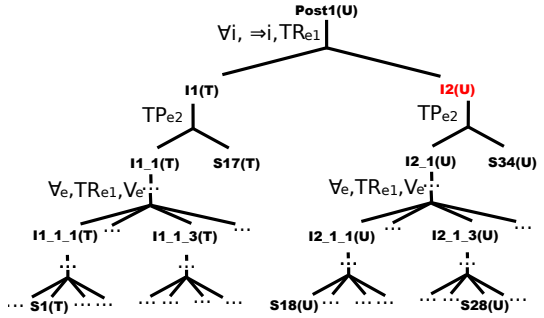

Figure 3. The proof tree w.r.t. *Post1* after the guard of the *R2R* rule is modified and populated with verification results using three-valued logic

- First, we populate the leaves of *Tree'*$_v$. This process is similar to Algorithm 1. If a leaf sub-goal has been cached, and none of its relevant rules are touched by the transition operators, then the cached verification result of the sub-goal is attached to the sub-goal in *Tree'*$_v$. Otherwise, we attach *UNKNOWN* to the leaf sub-goal in *Tree'*$_v$.
- Then, we populate the rest of *Tree'*$_v$ in a bottom-up manner using three-valued logic.

Next, we find a simplified postcondition (Definition 1) in *Tree'* to verify.

Definition 1. Simplified postcondition. A node in a proof tree populated with three-valued logic is named *simplified postcondition* if its verification result is *UNKNOWN*, and if verifying the Hoare-triple it represents (and consequently repopulate the proof tree) would make the verification result of the root of the proof tree to be a non *UNKNOWN* value.

A sub-node has more hypotheses and simpler conclusion than its parent node (which is the intention of our automated decomposition). Thus, we want to find a simplified postcondition in the proof tree to reduce the complexity of verifying the original postcondition. Ideally, the simplified postcondition should be as deep as possible in the proof tree. Therefore, we design the *findSimplifiedPost* function for this job: the function traverses the proof tree in a top-down manner; when it encounters a node *n* labeled with *UNKNOWN* verification result, it looks at the verification results of the children nodes of *n*. If there is exactly one child *c* labeled with *UNKNOWN*, we recursively apply the *findSimplifiedPost* function on the sub-tree with root *c*. Otherwise, we return *n*.

For example, assuming that we are modifying the guard of the rule *R2R*, the proof tree w.r.t. to *Post1* populated with verification results is shown in Fig. 3 (the verification results are attached on each node, where *T* stands for *TRUE* and *U* for *UNKNOWN*). The *findSimplifiedPost* function would return the simplified postcondition *I2* shown in Listing 4.

Finally, we construct a Hoare-triple for the simplified postcondition to check, i.e. MM, Pre, Exec$_{trace}$, Hypotheses $\vdash$ Conclusion. When the simplified postcondition is not a leaf node in the proof tree, we defensively use the union of static trace information of the leaf nodes in the proof tree (R') to

construct the sliced transformation context.

When the simplified postcondition is checked, we update its verification result in *Tree'*$_v$, and repopulate the tree. The verification result of the original postcondition is the label at the root of the *Tree'*$_v$.

Algorithm 2 Incremental verification algorithm for postconditions (P, P', op, post)

```
1: Tree ← decompose(P, post)
2: Tree' ← decompose(P', post)
3: R ← ⋃_{s_i ∈ leaves(Tree)} T(s_i, P)
4: R' ← ⋃_{s_i ∈ leaves(Tree')} T(s_i, P')
5: if post ∈ dom(C_1[P]) ∧ R=R' ∧ R'∩ δ_r(op) = ∅ then
6:    C_1[P'][post] ← C_1[P][post]
7: else
8:    Tree'_v ← populate(Tree', C_2[P][post])
9:    Node' ← findSimplifiedPost(Tree'_v)
10:   Node'_v ← verify(Node', P')
11:   Tree'_v ← repopulate(Node'_v, Tree'_v)
12:   C_1[P'][post] ← result(root(Tree'_v))
13: end if
```

IV. Soundness of the Incremental Verification

In what follows, we prove the soundness of our incremental verification approach.

A. Transition Operators

The transition operators that are used to transit from an *initial transformation* to its *evolved transformation* are shown in Fig. 4. Any modifications on the relational aspect of ATL can be represented as a sequence of applications of these operations. Some explanations are in order:

- The **add** operator adds an ATL rule named *R* that transforms the source pattern elements *srcs* to the target pattern elements *tars*. Initially, the add operator sets the guard of the added rule to *false* to prevent any potential rule conflict. The bindings for the specified target pattern elements are empty. The operator has no effect if the rule with specified name already exists in the initial transformation.
- The **delete** operator deletes a rule named *R*. It has no effect if the rule with the specified name does not exist in the initial transformation.
- The **setguard** operator strengthens/weakens the guard of the rule *R* by replacing its guard with the OCL expression *cond*. It has no effect if a rule with the specified name does not exist in the initial transformation.
- The **bind** operator modifies the way of binding the structural feature *sf* of the target element *tar* in the rule *R* to the binding OCL expression *b*. It has no effect if the rule with specified name does not exist in the initial transformation. If the *sf* or the *tar* does not exist in *R*, the bind operator adds a new binding.

⟨*operator*⟩ ::= **add from** *srcs* **to** *tars* **as** *R*
| **delete** *R*
| **setguard** *cond* **in** *R*
| **bind** *sf* **with** *b* **of** *tar* **in** *R*

Figure 4. The abstract syntax of transition operators

B. Execution Semantics Preservation

The essential idea of our incremental verification for ATL is based on the concept of execution semantics preservation. To verify a postcondition in a initial transformation *P*, the Hoare-triple *H*: MM, Pre, Exec ⊢ Post is constructed. Once the *P* is evolved to *P'* by applying our transition operator, we obtain a new Hoare-triple *H'*: MM, Pre, Exec' ⊢ Post. Clearly, if the execution semantics of P *Exec* is semantically equivalent to the execution semantics of P' *Exec'*, the verification result of *H* can be reused for *H'*. The same idea supports reusing the verification result of sub-goals.

However, execution semantics preservation is too restrictive for reusing the verification result of postcondition/sub-goals. In our experience, we find that there are many cases when the execution semantics of initial transformation is not strictly preserved by the evolved transformations, but the re-verification is not needed.

Therefore, we propose an approach to relax the role of execution semantics preservation in reusing the verification result of postconditions/sub-goals. Our approach is based on the definition of *weak execution semantics preservation* (Definition 2). Given a postcondition/sub-goal, our approach checks whether the evolved transformation weakly preserves the execution semantics of the corresponding initial transformation for that postcondition/sub-goal. In the affirmative case, the postcondition/sub-goal does not need to be re-verified.

Definition 2. Weak execution semantics preservation for postconditions/sub-goals. When the relevant rules (i.e. static trace information) of a postcondition/sub-goal in the initial and evolved transformations are the same, and these relevant rules do not intersect with the rule that the transition operator operated on, we say that the evolved transformation weakly preserves the execution semantics of the initial transformation for the postcondition/sub-goal.

We briefly demonstrate the execution semantics of ATL (details can be found in [12]), which helps us to prove the soundness of our approach for incrementally verifying postconditions/sub-goals.

Execution semantics of ATL. The execution semantics of an ATL matched rule consists of matching semantics and applying semantics. The matching semantics of a matched rule involves:

- Executability, i.e. the rule does not conflict with any other rules of the developed transformation.
- Matching outcome, i.e. all the source elements that satisfy the source pattern, their corresponding target elements of

target pattern have been created.
- Frame condition, i.e. nothing else is changed in the target model except the created target elements.

Take the *S2S* rule in the *ER2REL* transformation for example, its matching semantics specifies:

- Before matching the *S2S* rule, the target element generated for the *ERSchema* source element is not yet allocated (executability).
- After matching the *S2S* rule, for each *ERSchema* element, the corresponding *RELSchema* target element is allocated (matching outcome).
- After matching the *S2S* rule, nothing else is modified except the *RELSchema* element created from the *ERSchema* element (frame condition).

The applying semantics of a matched rule involves:

- Applying outcome, i.e. the created target elements are initialized as specified by the bindings of the matched rule.
- Frame condition, i.e. nothing else is changed in the target model except the initializations made on the target elements.

Take the *S2S* rule in the *ER2REL* transformation for example, its applying semantics specifies:

- After applying the *S2S* rule, for each *ERSchema* element, the name of its corresponding *RELSchema* target element is equal to its name (applying outcome).
- After applying the *S2S* rule, nothing else is modified, except the value of the name for the *RELSchema* element that created from the *ERSchema element* (frame condition).

C. Soundness Proof

Theorem 1. Soundness of Incremental Verification for Sub-goals. For a given sub-goal, if the evolved transformation weakly preserves the execution semantics of the corresponding initial transformation for this sub-goal, such sub-goal does not need to be re-verified in the evolved transformation.

Proof. We assume our automatic proof strategy can ensure the completeness of the static trace information of each sub-goal (i.e. every target types referred by each sub-goal and every rule that may generate them are correctly identified).

We induct on the type of transition operator *op* that modifies the initial transformation *P* to the evolved transformation *P'*. By hypothesis, P' weakly preserves the execution semantics of P for the sub-goal *s*, and the *op* is one of the following:

- **add** operator. The guard of the added rule is *false*. Thus, in this case, the execution semantics of the evolved transformation is semantically (strongly) equivalent to the initial transformation. Thus, *s* does not need to be re-verified.
- **delete** operator. The deleted rule is not referred by the static trace information of *s* in P and P' (Definition 2). Strictly speaking, this would potentially result in a different applying semantics of the referred rules in the static trace information of *s* in P'. For example, assuming the static trace information of a sub-goal includes only the rule *E2R*, deleting the rule *S2S* would alter the applying outcome of the *E2R* rule, since the binding *schema <– s.schema* cannot be resolved any more. However, each sub-goal essentially specifies a condition that sets of target elements should satisfy. The fact that the deleted rule is not referred by the static trace information of a sub-goal means that it does not generate target elements that affect the sub-goal (because of the independence among ATL matched rules, i.e. each rule has no effects to the target elements generated by other rules [32]). Thus, the deleted rule is irrelevant to the verification result of *s*, so that *s* does not need to be re-verified.
- **setguard** operator. Potentially, the rule that the **setguard** operated on could conflict with any of the rules referred by the static trace information of *s* in P. Consequently, the executability of those referred rules could be altered. However, our incremental verification approach applies only to valid transformation, which are free from rule conflicts. This guarantees that the **setguard** operator preserves the executability of the referred rules in the static trace information of *s*. Clearly, when P' is valid, the matching semantics of the rules in the static trace information of *s* is not affected by the **setguard** operator (because neither the matching outcome nor the frame condition of referred rules are affected). Similar to the delete operator, the fact that the modified rule is not referred by the static trace information of *s* implies proving that *s* does not depend on the execution semantics of the guard-modified rule. Thus, *s* does not need to be re-verified in this case.
- **bind** operator. When the rule of the modified binding is referred by the static trace information of *s*, this implies proving that *s* depends on the execution semantics of the binding-modified rule (but not necessarily on the modified binding). In this case, we defensively re-verify the sub-goal. However, similar to the delete operator, the fact that the rule of the modified binding is not referred by the static trace information of *s* implies proving that *s* does not depend on the execution semantics of the binding-modified rule. Thus, *s* does not need to be re-verified in this case.

To prove the soundness of incremental verification for postconditions, it is sufficient to prove three lemmas: (a) Lemma 1 ensures the soundness of reusing the cache for postconditions. (b) Lemma2 ensures the soundness of verifying each simplified postcondition. (c) Lemma3 ensures the soundness of repopulating the verification result of the proof tree.

Lemma 1. Soundness of reusing the cache for postconditions. For a given postcondition, if the evolved transformation weakly preserves the execution semantics of the corresponding initial transformation for this postcondition, such postcondition does not need to be re-verified in the evolved transformation.

Proof. We have ensured the soundness of our natural deduction rules by comparing them with the operational semantics of the ATL language. This guarantees the generated sub-goals are a sound abstraction of their corresponding original postcondition. Moreover, because we assumed that our automatic proof strategy can ensure the completeness of the static trace information of each sub-goal, we conclude that the union of the static trace information for each sub-goals of a postcondition contains all the rules that might affect the verification result of such postcondition. Then, the rest of the proof is similar to the inductive proof of Theorem 1.

Lemma 2. Soundness of verifying the simplified postcondition. When the Hoare-triple MM, Pre, Exec$_{trace}$, Hypotheses $\vdash$ Conclusion is verified, it implies the Hoare-triple MM, Pre, Exec, Hypotheses $\vdash$ Conclusion is verified.

Proof. Similar to the proof for the Lemma1.

Lemma 3. Soundness of repopulating the verification result of the proof tree. After verifying the simplified postcondition and repopulating the verification results of the proof tree, the verification result of the root of the proof tree is equivalent to the verification result of the original postcondition.

Proof. Straightforwardly proved by Lemma2, Definition 1 and the soundness of our natural deduction rules (which ensures the soundness of proof tree derivation).

V. Evaluation

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

A. Research questions

We formulate two research questions to evaluate the correctness and efficiency of our incremental verification approach:

(RQ1) Can our approach efficiently verify postconditions when a model transformation is modified?

(RQ2) Can our approach efficiently verify sub-goals of postconditions when a model transformation is modified?

B. Evaluation Setup

To answer our research questions, we use the HSM2FSM transformation with 14 preconditions and 6 postconditions as our case study [4], [8]. Our evaluation consists of two settings. In the first setting, the initial transformation is the original HSM2FSM transformations where all the specified postconditions are verified. Then, we individually generate 6 evolved transformations by applying 6 kinds of transition operators. These transition operators are defined in [7], [25] for mutation analysis [17] to systematically inject faults. Thus, the first setting spans on a range of activities (including extension, maintenance and refactoring) that are encountered when developing model transformations. Prompt re-verification result of postconditions is important under each of these circumstances, to provide the developer with valuable feedback such as whether new faults are introduced.

In the second setting, the initial transformation is the HSM2FSM transformation with all 6 kinds of mutation operators applied on it (*HSM2FSM*^{m}). Then, we individually generate 6 evolved transformations by reversing the applied mutation operators (reversing one per evolved transformation). Thus, the second setting simulates a range of activities that are encountered when debugging model transformations. Prompt re-verification result of postconditions/sub-goals are important under each of these circumstances, to provide the developer with valuable feedback such as whether all bugs are fixed, or which sub-goals remain unverified.

Then, we evaluate the two settings using 3 verification systems: (a) *ORG*. The original VeriATL verification system, which encodes the axiomatic semantics of the ATL language (version 3.7). It is based on the Boogie verifier (version 2.2) and Z3 (version 4.3). (b) *ORG*$_l$. VeriATL with fault localization capability. (c) *INC*. VeriATL with incremental verification capability (INC$_p$ for incremental verification of postconditions, and INC$_s$ for incremental verification of sub-goals). Moreover, as our incremental approach for postconditions refers the cache of sub-goals (Algorithm 2), we also evaluate its performance based on whether the full cache of sub-goals (Cache$_s$) is available or not.

The evaluation is performed on an Intel 3 GHz machine with 8 GB of memory running the Windows operating system. All of our verification system, fault localization and incremental verification approaches are implemented in Java. We kindly refer to our online repository for the complete artefacts used in our evaluation [13].

C. Evaluation Results

Table I summarizes the results for the first evaluation setting. Three columns in the table require some explanations: (a) the first column lists the identifiers of the initial and evolved transformations[2]; (b) the seventh column records the number of postconditions whose verification result are reused when the cache of sub-goals is **not** available during the verification process, the number of postconditions whose verification result are reused when the cache of sub-goals are presented during the verification process (shown in the brackets), and the total number of postconditions; (c) the last column records the number of sub-goals whose verification result are reused, and the total number of sub-goals that are checked during the verification process (Chk$_s$). There are sub-goals that are generated but not checked. This is because we check sub-goals only when its corresponding postcondition is not verified.

The verification time (i.e. the time spent by the Boogie verifier to get the verification result) shown in Table I clearly

[2]The naming convention for evolved transformations is the mutation operator Add(A) / Del(D) / Modify(M), followed by the mutation operand Rule(R) / Guard(G) / Binding(B), followed by the position of the operand in the original transformation setting. For example, *MB6* stands for the mutant which modifies the binding in the sixth rule, which is the initial HSM2FSM transformation applied with the transition operator **bind**.

demonstrates that using our incremental approaches to verify postconditions and sub-goals are consistently faster than using the original and fault localization approaches. For example, by verifying the simplified postconditions, we cut the verification time for postconditions by about 50%. This positively confirms our two research questions. One of the implications is that the developer would learn sooner the changes' effects on the postconditions while editing the model transformation to guide the next changes. Another implication is that the fast turnaround verification time of sub-goals would help the developer quickly comprehend what kind of fault is introduced. In our experience, this results in a better user experience, which could not be achieved without caching.

We can see that in 3 out 6 cases (*DB3, MB6, MG6*), the verification time using the incremental approach for sub-goals are almost the same (or faster) as in the original approach. This suggests that we might generate many sub-goals for postconditions; however verifying these sub-goals does not necessarily take longer than verifying the postconditions. As an extra benefit, we can get more detailed information from the sub-goals for fault localization.

The last two columns of Table I shed more light on why our incremental approach is so effective for verifying postconditions/sub-goals. Around 33% to 50% results of postconditions are reused from the cache of postconditions. Moreover, the last column shows that among 179 checked sub-goals, 131 of them are reused from cache (73%).

We also observe little cache reuse in the *AR* case, especially when w.r.t. the cache of sub-goals. By manual inspection, we find that this is because the hypotheses of newly generated sub-goals is slightly different to the one of the sub-goals in cache. This suggests some further optimization based on semantic analysis. For example, if we have a Hoare-triple for a sub-goal *s* in the initial transformation (MM, Pre, Exec_{trace}, Hypotheses $\vdash$ Conclusion), and a Hoare-triple for a sub-goal *s'* in the evolved transformation (MM, Pre, Exec_{trace}, Hypotheses' $\vdash$ Conclusion), and if we can prove Hypotheses' implies Hypotheses; then, we can safely reuse the verification result of *s* for *s'*. However, determining the general verification cost of such kind of optimization requires further investigation.

Table II summarizes the results for the second evaluation setting, which is structured as in Table I. One of the main differences between the two tables is that we can witness consistent spikes of verification time for sub-goals when using the fault localization approach. This is because the transformations of the second setting have more unverified postconditions than the first setting, which results in more *checked* sub-goals. This is a clear disadvantage for fixing faults because the developer has to suffer from the slow response of the verification tool.

The results show that our incremental approach for sub-goals saves about 64% verification time than the fault localization approach. The fact that 71% (228/323) of checked sub-goals are reused from the cache explains the efficient verification time. However, as there are more checked sub-goals, we find less cases where the verification time of incremental approach for sub-goals is almost the same as in the original approach.

Another main difference is that the cache of sub-goals plays an important role when incrementally verifying postconditions in the second setting. Intuitively, as we have more unverified postconditions in the second setting, we have more unverified sub-goals. Consequently, we find 8 cases (23 - 15) in which, after repopulating the proof tree with the verification result of sub-goals, the verification result of the root of the proof tree can be immediately computed without verifying any simplified postcondition (since *UNKNOWN* $\wedge$ *FALSE* = *FALSE*). However, one thing to notice is that to benefit from the cache of sub-goals, the developer has to wait until it is finished (even partially).

Table I
EVALUATION RESULTS FOR THE 1ST SETTING

	Verification Time (in seconds)					Number of	
	ORG	INC_p Cache$_s$?		ORG_l	INC_s	Reuse / Total_p	Reuse_s / Chk_s
		N	Y				
HSM2FSM	21	21	21	21	21	N/A	N/A
DB3	19	10	9	35	21	2 (2) / 6	11 / 16
MB6	20	9	8	41	17	3 (3) / 6	24 / 28
AG2	21	10	10	42	26	2 (2) / 6	13 / 19
MG6	23	10	9	55	22	3 (3) / 6	28 / 34
DR1	14	6	6	33	26	3 (3) / 6	48 / 60
AR	24	11	11	66	57	2 (2) / 6	7 / 22
TOTAL	121	56	53	272	169	15 (15) / 36	131 / 179

Table II
EVALUATION RESULTS FOR THE 2ND SETTING

	Verification Time (in seconds)					Number of	
	ORG	INC_p Cache$_s$?		ORG_l	INC_s	Reuse / Total_p	Reuse_s / Chk_s
		N	Y				
HSM2FSM^m	20	20	100	100	100	N/A	N/A
DB3^{-1}	18	11	8	105	40	2 (4) / 6	31 / 46
MB6^{-1}	18	9	4	106	21	3 (5) / 6	39 / 46
AG2^{-1}	17	12	8	107	39	2 (4) / 6	31 / 46
MG6^{-1}	18	9	4	103	22	3 (5) / 6	39 / 46
DR1^{-1}	28	8	8	111	38	3 (3) / 6	79 / 93
AR^{-1}	14	9	9	70	55	2 (2) / 6	7 / 46
Total	113	58	41	602	215	15 (23) / 36	228 / 323

D. Discussion

In summary, through our evaluation, we simulate two common settings for developing and fixing model transformations. We gain confidence that our incremental approach would consistently provide efficient verification result of postconditions and sub-goals when a model transformation is modified. However, there are some lessons we learned:

Completeness. Theoretically, our approach could report inconclusive simplified postconditions/sub-goals when the original postcondition should be verified (i.e. incompleteness). We manually checked that our approach is complete for our evaluation, and identify two possible sources of incompleteness: (a) the limits of the underlying SMT solver [21]. (b) unsuccessful application of natural deduction rules by the designed automated proof strategy [14]. We plan to design more

natural deduction rules for ATL and a smarter automated proof strategy to contribute to the completeness of the approach. We also envision enabling interactive theorem proving when we can not perform automatic verification.

Usability. Currently, the cache for postcondition and sub-goals are always computed for our incremental approach. Then, the developer needs to decide whether to use one/either/neither of them in the evolved transformation. The future plan is to integrate our incremental approach into Eclipse IDE to make it more user-friendly (making it a one-button technology). For example, once the developer requests to verify&cache the postconditions, a new background thread could spawn to verify&cache sub-goals. Meanwhile, all the background threads that verify&cache sub-goals of previous evolved transformations should give their priority to the newly spawned background thread.

Coverage. In this work we consider the relational aspect of the ATL language, i.e. ATL matched rules, in non-refinement mode, one-to-one mappings of (possibly abstract) classifiers with the default resolution algorithm of ATL. The non-recursive helpers are supported by inlining them into the developed MT. The imperative and recursive aspects of ATL can be integrated into our approach by learning from the state-of-the-art incremental verification techniques for general-purpose programming languages (Section VI).

VI. Related Work

Incremental verification for general programming languages has drawn the attention of researchers in recent years to improve user experience of program verification. Bobot et al. design a proof caching system for the Why3 program verifier [6]. In Why3, a proof is organized into a set of sub-proofs whose correctness implies the correctness of the original proof. Bobot et al. encrypt the sub-proofs into strings. When the program updates, the new sub-proofs are also encrypted and looking for the best matches in the old sub-proofs. Then, a new sub-proof is heuristically applied with the best matched old sub-proof's proof effort to make the verification more efficient.

Leino and Wüstholz design a two level caching for the proofs in the Dafny program verifier [22]. First, a coarse-grained caching that depends on the call graph of the program under development, i.e. a *caller* program does not need to be re-verified if its *callee* programs remain unchanged. Second, a fine-grained caching that depends calculating the checksum of each contract. The checksum of each contract is calculated by all statements that the contract potentially depends on. Thus, if the checksum does not change, the re-verification is not needed. Moreover, the authors also use explicit assumptions and partial verified checks to generate extra contracts. If they are proved, the re-verification is not needed.

Logozzo et al. design the Clousot verifier [23]. Clousot captures the semantics of a base program by execution traces (a.k.a semantic conditions). Then, these conditions are inserted into the new version of the program as assumptions. This technique is used to incrementally prove the relative correctness between base and new version of programs.

The Proofcert project aims at sharing proofs across several independent tools by providing a common proof format [24], which would allows an orchestra of tools to collectively prove the correctness of computer systems more automatically.

All the works we just discussed are designed to make the verification of imperative or functional programs faster. Our approach complements these works by focusing on a different program paradigm, i.e. relational programs. The incremental verification of each paradigm has an importance of its own. Moreover the verification of hybrid languages would need to integrate techniques from several works. For instance the imperative and functional parts of the ATL language could benefit from the mentioned works.

Formula-based debugging is an active topic of research [30]. Among these, Qi et al. develop a tool called *DARWIN* to debug evolving programs [29]. We both reuse some of the information from previous versions of program. However, in the work by Qi et al., calibrating is the main responsibility of this reuse to search a new execution path that helps debugging. The role of reuse in our work is for incremental verification.

In addition, according to surveys [1], [2], [11], incremental verification has not been adapted in the model transformation verification. We hope that our approach would be useful in this context.

VII. Conclusion and Future Work

In summary, in this work we propose an incremental verification approach to improve the performance of deductive verification for the ATL language. Our approach caches the verification result of postconditions/sub-goals, determines when the cache can be reused, and then incrementally re-verifies the postconditions/sub-goals that are not reused. Moreover, when incrementally re-verifying the postconditions, we reduce the re-verification complexity by constructing a simplified postconditions to verify. We prove the soundness of our approach and show its effectiveness by mutation analysis. Our evaluation shows an approximate 50% reuse of verification result for postconditions, and 70% reuse of verification result for sub-goals. Consequently, we gain about 56% reduction of verification time for postconditions, and 51% for sub-goals.

In future work we plan to complete the coverage of the ATL language (including functional and imperative parts) and develop the tooling to make verification usable during ATL development. We plan also to evaluate a proactive approach to further improve the performance of incremental verification, by prioritizing the verification of the subgoals that will be most probably impacted by the following change.

Acknowledgment

The present work has been supported by the MONDO (EU ICT-611125) project.

REFERENCES

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

[2] Amrani, M., Lucio, L., Selim, G., Combemale, B., Dingel, J., Vangheluwe, H., Le Traon, Y., Cordy, J.R.: A tridimensional approach for studying the formal verification of model transformations. In: 5th International Conference on Software Testing, Verification and Validation. pp. 921–928. IEEE Computer Society, Washington, DC, USA (2012)

[3] 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)

[4] Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.M.: Barriers to systematic model transformation testing. Communications of the ACM 53(6), 139–143 (2010)

[5] 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)

[6] Bobot, F., Filliâtre, J.C., Marché, C., Melquiond, G., Paskevich, A.: Preserving user proofs across specification changes. In: 5th International Conference on Verified Software: Theories, Tools, Experiments. pp. 191–201. Springer, Menlo Park, CA, USA (2014)

[7] Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. IEEE Transactions on Software Engineering 41(5), 490–506 (2015)

[8] 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)

[9] 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)

[10] 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)

[11] Calegari, D., Szasz, N.: Verification of model transformations: A survey of the state-of-the-art. Electronic Notes in Theoretical Computer Science 292(0), 5–25 (2013)

[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] Cheng, Z., Tisi, M.: Incremental deductive verification for relational model transformations [online]. available: https://github.com/veriatl/genTool (2016)

[14] Cheng, Z., Tisi, M.: A deductive approach for fault localization in ATL model transformations. In: 20th International Conference on Fundamental Approaches to Software Engineering. Uppsala, Sweden (2017)

[15] 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)

[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] Kleene, S.C.: Introduction to Metamathematics. North-Holland (1962)

[20] Lano, K., Clark, T., Kolahdouz-Rahimi, S.: A framework for model transformation verification. Formal Aspects of Computing 27(1), 193–235 (2014)

[21] Leino, K.R.M.: This is Boogie 2. http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf. Microsoft Research, Redmond, USA (2008)

[22] Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: 27th International Conference on Computer Aided Verification. pp. 380–397. Springer, San Francisco, CA, USA (2015)

[23] Logozzo, F., Lahiri, S.K., Fähndrich, M., Blackshear, S.: Verification modulo versions: Towards usable verification. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 294–304. ACM, New York, NY, USA (2014)

[24] Miller, D., Pimentel, E.: A formal framework for specifying sequent calculus proof systems. Theoretical Computer Science 474, 98–116 (2013)

[25] Mottu, J.M., Sen, S., Tisi, M., Cabot, J.: Static analysis of model transformations for effective test generation. In: 23rd International Symposium on Software Reliability Engineering. pp. 291–300. IEEE, Dallas, TX (2012)

[26] 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)

[27] 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)

[28] 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)

[29] Qi, D., Roychoudhury, A., Liang, Z., Vaswani, K.: DARWIN: An approach to debugging evolving programs. ACM Transactions on Software Engineering and Methodology 21(3) (2012)

[30] Roychoudhury, A., Chandra, S.: Formula-based software debugging. Communications of the ACM (2016)

[31] 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)

[32] 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)

[33] Wagelaar, D.: Using ATL/EMFTVM for import/export of medical data. In: 2nd Software Development Automation Conference. Amsterdam, Netherlands (2014)

APPENDIX A: NATURAL DEDUCTION RULES SPECIFIC TO ATL

$$\frac{x.a : T \quad T \in cl(MM_{\mathrm{T}})}{x.a \in All(T) \vee unDef(x.a)}\ \mathrm{TP}_{e1}$$

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

$$\frac{T \in cl(MM_{\mathrm{T}}) \quad trace(T) = \{R_1, ..., R_n\} \quad i \in All(T)}{genBy(i, R_1) \vee ... \vee genBy(i, R_n)}\ \mathrm{TR}_{e1}$$

$$\frac{T \in cl(MM_{\mathrm{T}}) \quad trace(T) = \{R_1, ..., R_n\} \quad i : T \quad unDef(i)}{\neg(genBy(i, R_1) \vee ... \vee genBy(i, R_n))}\ \mathrm{TR}_{e2}$$

Notations. 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), *genBy(i,R)* is a predicate to indicate a model element *i* is generated by the rule *R*, *unDef(i)* abbreviates *i.oclIsUndefined()* and *All(T)* abbreviates *T.allInstances()*.