Towards Incremental Deductive Verification for ATL

Zheng Cheng and Massimo Tisi

AtlanMod Team (Inria, Mines Nantes, LINA), France
Email: {zheng.cheng, massimo.tisi}@inria.fr

Abstract. In this work, we address the performance problem in the deductive verification of model transformations written in the ATL language w.r.t. given contracts. Our solution is to enable incremental verification for ATL transformations through caching and reusing of previous verification results. Specifically, we decompose the original OCL contract into sub-goals, and cache the verification result of each of them. We show that for ATL, the syntactic relationship between a model transformation change and a sub-goal can be used to determine if a cached verification result needs to be re-computed. We justify the soundness of our approach and its practical applicability through a case study.

1 Introduction

In [7], we developed the *VeriATL* verification system to deductively verify the correctness of ATL transformations [10] w.r.t. given contracts. On top of that, in [8] we enabled automatic fault localization for VeriATL to facilitate debugging, based on the decompositon of OCL postconditions into verification sub-goals. 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 C (in terms of pre/postconditions) to ensure its correctness. Next, VeriATL is automatically executed at the back-end. It decomposes each contract into a set of sub-goals S to facilitate fault localization. Then, the verifier reports to the developer that N contracts and M corresponding sub-goals are not verified. 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. However, P' causes VeriATL to decompose each of the contracts into a new set of sub-goals S'.

The practical applicability of VeriATL in such workflow strongly depends on the possibility to compute the implications of a change in a time-efficient manner, e.g. which new sub-goals are generated? which old sub-goals disappeared? and which old sub-goals are affected by the change? Our solution is running VeriATL incrementally, i.e. maintaining the result of contracts across changes in the model transformation by reusing cached verification results of sub-goals. To achieve our objective, we propose a syntactic approach, based on the syntactic relationship between a model transformation change and a sub-goal, to determine when the cached verification result of a sub-goal can be reused. In this paper we justify the soundness of our approach and demonstrate its feasibility by a first evaluation.

Paper organization. We give the background of our work and a sample problem in Section 2. Section 3 illustrates our incremental approach for model transformation verification in detail. The evaluation is presented in Section 4. Section 5 compares our work with related research, and Section 6 draws conclusions and lines for future work.

2 Background

We have developed the VeriATL verification system that soundly verifies the correctness of ATL model transformations via Hoare-triples [7]. We demonstrate VeriATL on the *ER2REL* transformation, well-known in the MDE community, that 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).
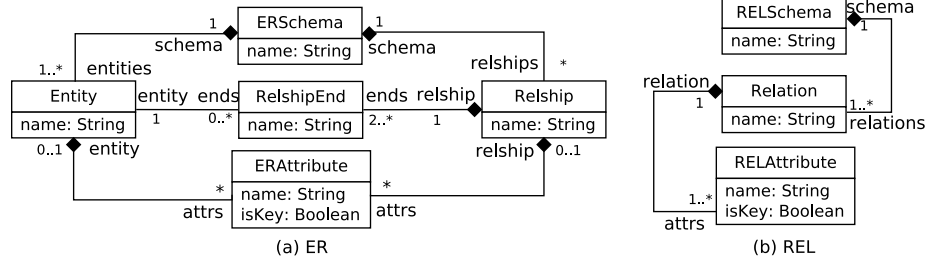

Fig. 1. The Entity-Relationship and Relational metamodels

The ATL transformation *ER2REL* (Listing 1.1) 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.

Then, the correctness of the ER2REL transformation is specified by using OCL contracts. As shown in Listing 1.2, two OCL preconditions specify that within an *ERSchema*, names of *Entities* are unique (*Pre1*), and names of *Entities* and *Relships* are disjoint (*Pre2*). One OCL postcondition requires names of *Relations* to be unique within each *RELSchema* (*Post1*).

```
1  module ER2REL;
2  create OUT : REL from IN : ER;
3
4  rule S2S { from s: ER!ERSchema to t: REL!RELSchema (name<−s.name)}
5
6  rule E2R { from s: ER!Entity to t: REL!Relation ( name<−s.name, schema<−s.schema) }
7
8  rule R2R { from s: ER!Relship to t: REL!Relation ( name<−s.name, schema<−s.schema) }
9
10 rule EA2A {
11   from att: ER!ERAttribute, ent: ER!Entity (att.entity=ent)
12   to t: REL!RELAttribute ( name<−att.name, isKey<−att.isKey, relation<−ent ) }
13
14 rule RA2A {
15   from att: ER!ERAttribute, rs: ER!Relship ( att.relship=rs )
16   to t: REL!RELAttribute ( name<−att.name, isKey<−att.isKey, relation<−rs ) }
17
18 rule RA2AK {
19   from att: ER!ERAttribute, rse: ER!RelshipEnd ( att.entity=rse.entity and att.isKey=true )
20   to t: REL!RELAttribute ( name<−att.name, isKey<−att.isKey, relation<−rse.relship )}
```

Listing 1.1. The ER2REL model transformation in ATL

```
1  context ER!ERSchema inv Pre1, Pre2
2
3  rule S2S { from s: ER!ERSchema to t: REL!RELSchema (name<−s.name)}
4
5  rule R2R { from s: ER!Relship to t: REL!Relation ( name<−s.name, schema<−s.schema) }
6
7  context REL!RELSchema inv S4:
8  REL!RELSchema.allInstances()−>forAll(s | genBy(s, S2S) implies
9    s. relations −>forAll(r1 | genBy(r1, R2R) implies
10     s. relations −>forAll(r2 | genBy(r2, R2R) implies
11       r1<>r2 implies r1.name<>r2.name ) ) )
```

Listing 1.3. The problematic transformation scenario of the *ER2REL* transformation w.r.t. *Post1*

```
1  context ER!ERSchema inv Pre1:
2  ER!ERSchema.allInstances()−>forAll(s | s.entities−>forAll(e1 |
3    s. entities −>forAll(e2 | e1<>e2 implies e1.name<>e2.name)))
4
5  context ER!ERSchema inv Pre2:
6  ER!ERSchema.allInstances()−>forAll(s | s.entities−>forAll(e |
7    s. relships −>forAll(r | e.name<>r.name)))
8  −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
9  context REL!RELSchema inv Post1:
10 REL!RELSchema.allInstances()−>forAll(s | s.relations−>forAll(r1 |
11   s. relations −>forAll(r2 | r1<>r2 implies r1.name<>r2.name)))
```

Listing 1.2. The OCL contracts for ER and REL

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. The Hoare-triple semantically means that, assuming the semantics of the involved EMF metamodels and OCL preconditions, by executing the developed ATL transformation, the specified OCL postcondition has to hold.

The first version of VeriATL can successfully report that the OCL postcondition *Post1* is not verified by the ER2REL transformation, but it does not provide any information to understand why this contract does not hold. To enable fault localization in VeriATL, we have proposed a systematic approach to decompose OCL postconditions into sub-goals based on static analysis and structural induction. Consequently, we can use the information (i.e. static trace information) in the decomposed sub-goals to slice the original transformation into transformation scenarios, and present the problematic ones to pinpoint the fault.

For example, to verify *Post1*, our decomposition generates 4 sub-goals, where:

1. *s* genBy (i.e., generated by) *S2S*, *r1* genBy *E2R*, *r2* genBy *E2R*.
2. *s* genBy *S2S*, *r1* genBy *E2R*, *r2* genBy *R2R*.
3. *s* genBy *S2S*, *r1* genBy *R2R*, *r2* genBy *E2R*.
4. *s* genBy *S2S*, *r1* genBy *R2R*, *r2* genBy *R2R*.

The 4 sub-goals therefore yield 4 transformation scenarios to be verified. One of the 4 transformation scenarios is problematic (shown in Listing 1.3). VeriATL reports it to the developer to pinpoint the fault in *Post1*. The scenario consists of the original preconditions (abbreviated at the top), a slice of the transformation (in the middle) and an augmented transcription of the problematic sub-goal (at the bottom).

Notice that the static trace information referred by the sub-goal (the *genBy* predicate on Line 8, 9, 10) helps the transformation developer to understand that the error may happen only when

the relational schema s is generated from the rule S2S, and when the relations $r1$ and $r2$ are both generated from the rule R2R. Therefore, the only relevant rules for the fault are *S2S* and *R2R*. By analyzing the problematic transformation scenario in Listing 1.3, the transformation developer observes that two *Relships* in the source model might have the same name. This would cause their corresponding *Relations*, generated by the *R2R* rule, to have the same name, and thus falsifying *Post1*. More examples of our OCL postcondition decomposition for automatic fault localization in VerATL can be found on our online repository [9].

Fixing transformation bugs by VeriATL is an interactive process. Each change to the ER2REL transformation launches VeriATL to re-verify the contracts and generated sub-goals. In this work, we aim to improve the turnaround time of re-verification by running VeriATL incrementally. Therefore, we propose a solution for determining when the cached verification result of sub-goals can be reused.

For example, our approach aims to determine that when fixing *Post1* by modifying the *R2R* rule, the verification result of the sub-goal for *Post1*, where s genBy *S2S*, $r1$ genBy *E2R*, $r2$ genBy *E2R* can be reused.

3   Incremental Verification for VeriATL

Our approach for running VeriATL incrementally is summarized by Algorithm 1.

First, we distinguish an ATL transformation (*initial transformation*) P from its modified version (*evolved transformation*) P'. The transition from P to P' (line 1) happens through the sequential application of transition operators that we define in Section 3.1. As shown by lines 2 - 4, if P' is invalid, e.g. it contains conflicting rules, then the incremental verification is stopped (and restarted when the user reaches a valid transformation by further modifications).

Next, we enumerate each sub-goal in P to determine whether the execution semantics of its referred rules in the static trace information is preserved by P' (Lines 5 - 11). The execution semantics preservation results of each sub-goal are cached by the array *isPsv*. Specifically, we propose a syntactic approach to determine the execution semantics preservation (Lines 7 - 8), which checks the intersection between the referred rules in the static trace information of a sub-goal and the rule that the transition operator operated on.

Finally, the sub-goal does not need to be re-verified if it is a verified sub-goal in P, and the execution semantics of its referred rules in static trace information is preserved (Lines 14 - 15).

In what follows, we detail each of these steps, and justify the soundness of our approach.

3.1   Transition Operators

The transition operators that are permitted to transit from an *initial transformation* to its *evolved transformation* are shown in Fig. 2. Some explanations are in order:

- The **add** operator adds an ATL rule named R that transforms the input pattern elements *srcs* to the output pattern elements *tars*. Initially, the add operator sets the filter of the added rule to *false* to prevent any potential rule conflict. The bindings for the specified output 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 hass no effect if the rule with the specified name does not exist in the initial transformation.

Algorithm 1 Incremental verification algorithm for VeriATL

```
 1: P' = transit(P, operator)
 2: if hasConflict(P') then
 3:     abort
 4: end if
 5: for each s ∈ sub-goals of P  do
 6:     if trace(s) ∉ dom(isPsv) then
 7:         if trace(s) ∩ affectedRules(operator, P, P') = ∅ then
 8:             isPsv[trace(s)] ← true
 9:         end if
10:     end if
11: end for
12: for each s ∈ sub-goals of P'  do
13:     if s ∈ sub-goals of P then
14:         if trace(s) ∈ dom(isPsv) ∧ isPsv(trace(s)) ∧ verified(s, P) then
15:             continue
16:         else
17:             re-verify(s)
18:         end if
19:     end if
20: end for
```

- The **filter** 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 acts like adding a new binding.

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

Fig. 2. The abstract syntax of transition operators

3.2 Execution Semantics Preservation

The core idea of our proposal is that a sub-goal s does not need to be re-verified after applying a transition operator *op* if *op* preserves the execution semantics of the referred rules in the static trace information of s. This is because proving sub-goals is the process of proving Hoare-triples whose correctness depends on contracts and execution semantics of the involved transformation rules: if neither of these artefacts are changed, the correctness of such Hoare-triple will not change.

However, in our experience, we find that there are many cases when the execution semantics of the referred rules in the static trace information of a sub-goal is not strictly preserved, but the re-verification of the sub-goal is not needed. Therefore, we propose a syntactic approach to characterize these cases, thereby determining whether the re-verification of a verified sub-goal in the initial transformation is necessary.

Our syntactic approach checks the intersection between the referred rules in the static trace information of a sub-goal and the rule that the transition operator operated on. When the intersection is empty, the sub-goal does not need to be re-verified.

We justify the soundness of our syntactic approach in two steps. First, we briefly demonstrate the execution semantics of the ATL matched rule (details can be found in [7]). Then, we induct on the type of transition operator to prove soundness.

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 is not conflict with any other rules of the developed transformation.
– Matching outcome, i.e. all the source elements that satisfy the input pattern, their corresponding target elements of output 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 equals 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).

Next, we induct on the type of transition operator to prove the soundness of our syntactic approach, i.e. when the intersection between the referred rules in the static trace information of a sub-goal and the rule that the transition operator operated on is empty, it guarantees the re-verification of a sub-goal is unnecessary:

– **add** operator. The filter of the added rule is *false*. Thus, the added rule takes no effects to the rest of rules in the developed transformation when applying the operator, including the rules referred by the static trace information of a sub-goal. Consequently, our syntactic approach is sound in this case.

– **delete** operator. Strictly speaking, deleting a rule that is not referred by the static trace information of a sub-goal could potentially alter the applying semantics of the referred rules in the static trace information of the sub-goal. However, the fact that the deleted rule is not referred by the static trace information of a sub-goal implies that proving the sub-goal does not depend on the execution semantics of the deleted rule. Thus, our syntactic approach is sound in this case.

– **filter** operator. Altering the filter of a rule that is not in the referred rules of a sub-goal could affect the executability of those referred rules, i.e. when the modified rule conflicts with any of the referred rules. However, as shown by lines 2 - 4 of Algorithm 1, our incremental verification approach applies only to valid transformation (i.e., free from rule conflicts), which guarantees that the filter operator preserves the executability of the referred rules in the static trace information of the sub-goal. When applying the **filter** operator to a rule that is not in the referred rules of a sub-goal results a valid transformation, the number of source elements that satisfy the input patterns of those referred rules would not change. As a result, the same number of corresponding target elements of output patterns specified by those referred rules are generated. Thus, the matching outcome and the frame condition of the referred rules of a sub-goal are not affected. These facts conclude that altering the filter of a rule that is not in the referred rules of a sub-goal would not affect the matching semantics of those referred rules. Similar to the delete operator, the fact that the modified rule is not referred by the static trace information of a sub-goal implies proving that the sub-goal does not depend on the execution semantics of the filter-modified rule. Thus, our syntactic approach is sound in this case.

– **bind** operator. When the rule of modified binding is referred by the static trace information of a sub-goal, it implies proving the sub-goal depends on the execution semantics of the binding-modified rule (but not necessarily depends on the modified binding). In this case, we defensively reverify the sub-goal. However, similar to the delete operator, the fact that the rule of modified binding is not referred by the static trace information of a sub-goal implies proving the sub-goal does not depend on the execution semantics of the binding-modified rule. Thus, our syntactic approach is sound in this case.

4 Case Study

While we used ER2REL for illustrative purposes, we evaluate the performance of our approach on a more complex case study, i.e. the *HSM2FSM* transformation that translates a hierarchical state machine (source) to a flattened state machine (target) [3, 5]. Each of the source and target metamodels contains 6 classifiers, 2 attributes and 5 associations. The HSM2FSM transformation contains 7 ATL matched rules. The contracts of the HSM2FSM transformation consists of 14 preconditions and 6 postconditions. Our evaluation uses the VeriATL verification system, which is based on the Boogie verifier (version 2.2). The artefacts used in our case study can be found on our on-line repository: `https://github.com/veriatl/OclCache`.

We pick 4 transition operators and manually apply them individually to the initial HSM2FSM transformation for our evaluation:

(T1) **add** *CS2RS* **from** *cs:HSM!CompositeState* **to** *rs:FSM!RegularState*
(T2) **delete** *SM2SM*
(T3) **filter** *T2TA* **with** *false*
(T4) **bind** *IS2IS FSM!InitialState stateMachine* **with** *null*

While other transition operations are possible, the first two are designed to quantify the performance of our approach when the generated sub-goals in the initial transformation and evolved transformation are different, whereas the latter two evaluate our approach when the generated sub-goals are the same.

The impact of the 4 transition operations on the re-verification of the generated sub-goals for the 6 postconditions are summarized in Table 1. To fully evaluate our approach, we compute and verify the sub-goals no matter whether the original OCL postcondition is verified or not.

Table 1. Performance metrics on the evaluation scenario

	Post1	Post2	Post3	Post4	Post5	Post6	Avg. Cache Reused
Initial	8(0)	8(0)	16(1)	16(0)	4(0)	16(0)	N/A
T1	6/10	8/8	12/20	12/20	4/4	9/25	69 %
T2	0/4	0/4	15/16	16/16	0/1	16/16	49 %
T3	8/8	4/8	7/16	8/16	4/4	16/16	74 %
T4	4/8	8/8	8/16	8/16	4/4	4/16	63 %

The cells in the first row represents how many sub-goals are generated for the initial transformation, and the number of unverified sub-goals are shown in parenthesis. The cells in the next 4 rows represent the ratio *reused sub-goals/generated sub-goals* for each of the 6 postconditions after applying the designed transition operators.

As shown in the last column of Table 1, when applying our approach, at least 49% of the previous verification results can be reused. However, as our case study is small, more case studies are required to claim the generalization for the performance of our approach.

In addition, we can see in some of the cells that for some post-conditions none of the cached results are reused. By manual examination we confirm that in these cases the newly generated sub-goals and the effects of the transition operation are highly coupled.

5   Related Work

Incremental verification for general programming languages have 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 [4]. 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 [11]. 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 depends on. Thus, if all statements that a contract depends on do 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 [12]. 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.

Proofcert project aims at sharing proofs across several independent tools by providing a common proof format [13].

According to surveys [1, 2, 6], incremental verification has not been adapted in the model transformation verification. We hope that our approach would be useful in this context.

6   Conclusion and Future Work

In summary, in this work we confront the performance problem for deductive verification of the ATL language. Our solution is to evolve the VeriATL deductive verification system with the incremental verification capability through caching and reusing of previous verification results. Specifically, our incremental verification approach is based on decomposing the original OCL postconditions into sub-goals, and caching their verification results. We propose a syntactic approach, based on the syntactic relationship between a model transformation change and a sub-goal, to determine when the cached verification result of the sub-goal can be reused.

Our current work focuses on the performance aspect. We plan to extend our experimental evaluation with further case studies and investigate how to cache the verification results of sub-goals more efficiently. In reality, fixing a bug can be seen as a sequential application of our transition operators. We want to investigate how to optimize the full chain, e.g. caching the verification results of disappeared sub-goals that could reappear in future versions of the model transformation.

Our future work will focus on making the implications of a model transformation change visible to the developers to guide their next move. When a developer edits source code, the sooner the developer learns the changes' effects on program analyses, the more helpful those analyses are [14]. A delay can lead to wasted effort. In fact, the implications of a model transformation change is already computed by our syntactic approach. We will focus on how to present this information to the developers in order to provide an intuitive understanding.

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. 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)
4. 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)
5. 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)

6. 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)
7. 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)
8. Cheng, Z., Tisi, M.: Automatic fault localization for relational model transformation using deductive verification. In: 27th International Symposium on Software Reliability Engineering. p. (Under review). Ottawa, Canada (2016)
9. Cheng, Z., Tisi, M.: Automatic fault localization for relational model transformation using deductive verification [online]. available: `https://github.com/veriatl/OclDecompose` (2016)
10. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
11. 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)
12. 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)
13. Miller, D., Pimentel, E.: A formal framework for specifying sequent calculus proof systems. Theoretical Computer Science 474, 98–116 (2013)
14. Muşlu, K., Brun, Y., Ernst, M.D., Notkin, D.: Reducing feedback delay of software development tools via continuous analysis. IEEE Transactions on Software Engineering 41(8), 745–763 (2015)