Fully Verifying Transformation Contracts for Declarative ATL

Bentley James Oakes*, Javier Troya†, Levi Lúcio*, and Manuel Wimmer†
*School of Computer Science, McGill University, Canada
levi@cs.mcgill.ca, bentley.oakes@mail.mcgill.ca
†Business Informatics Group, Vienna University of Technology, Austria
{troya,wimmer}@big.tuwien.ac.at

***Abstract*—The Atlas Transformation Language (ATL) is today a de-facto standard in model-driven development. It is understood by the community that methods for exhaustively verifying such transformations provide an important pillar for achieving a stronger adoption of model-driven development in industry.**

In this paper we propose a method for verifying ATL model transformations by translating them into DSLTrans, a transformation language with limited expressiveness. Pre-/post-condition contracts are then verified on the resulting DSLTrans specification using a symbolic-execution property prover.

The technique we present in this paper is exhaustive for the declarative ATL subset, meaning that if a contract holds, it will hold when any input model is passed to the ATL transformation being checked. We explore the scalability of our technique using a set of examples, including a model transformation developed in collaboration with our industrial partner.

***Index Terms*—Model transformation, Formal verification, ATL, Contracts, Symbolic execution, Pre-/Post-conditions**

I. Introduction

Graph-based model transformations have become in the last few years the main means for manipulating models in model-driven development. Their simplicity, their allowance for mathematical treatment, and the fact that they can natively manipulate domain-specific concepts expressed in metamodels, all make graph-based model transformations an excellent compromise between strong theoretical foundations and applicability to real-world problems. In particular, the Atlas Transformation Language (ATL) [3] has come to prominence in the model-driven development community. This success is due to ATL's flexibility, support of the main meta-modelling standards, usability that relies on good tool integration with the Eclipse world, and a supportive development community.

Because of the importance of ATL in both the academic and the industrial arenas, specification verification is of prime importance: firstly because the correctness of software built using model-driven development techniques typically relies on the correctness of many operations executed using model transformations; and secondly because tools that allow building verified software are in strong demand, especially in industry where quality and security standards have to be met.

In this paper we address this issue by proposing a novel technique to verify visual pre-/post-condition contracts on ATL specifications. A contract holds for the transformation if, for all input models where the contract's pre-condition is found, the contract's post-condition is also found in the corresponding output model (with optional traceability constraints between the elements of the input and output models). Otherwise, the contract does not hold.

For example, this paper considers the well-known *Families-to-Persons* transformation from the ATL zoo [1], where mother(s), father(s), daughter(s) and son(s) belonging to a family are translated into men and women who are members of a community. One possible contract would try to assert that, for any input model containing a family that includes a mother and a daughter, a man is produced in the output community. We would expect the contract not to hold for the *Families-to-Persons* transformation, because there can exist families that are composed of only a mother and her daughter.

The main contribution of our technique is that, if our prover demonstrates that the contract holds, then it will hold for any input model given to the ATL model transformation. We can thus guarantee the user can safely execute the model transformation without any need for additional testing or runtime checking, as seen in other ATL verification approaches. Our contract language is based on pre-/post-condition contracts, but also includes propositional logic operators for combining contracts. This is further discussed in Section III-D.

We prove that contracts hold or not by translating ATL specifications into transformations defined in a model transformation language called DSLTrans [7]. A theoretical framework has been developed for the DSLTrans model transformation language in which pre-/post-condition contracts can be shown to hold for all input/output pairs resulting from executing a given DSLTrans model transformation, or to not hold for at least one of those input/output pairs [17]. A fully automatic property prover based on this theory has been shown to be applicable to industrial problems [20].

In this paper we focus on verifying the declarative part of ATL, given its similarity to the DSLTrans model transformation language. It is common practice to use this subset of the language for the majority of transformation requirements. Additionally, using only declarative ATL normally results in clearer, more readable and more maintainable model transformations than when the imperative part of the language is used.

The presentation of our work will follow this outline:

Section II briefly introduces DSLTrans and its relevant constructs. Following this, Section III presents the *Families-to-Persons* transformation, introduces its ATL representation

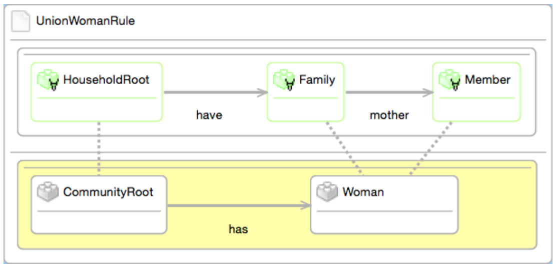

Fig. 1. The *UnionWomanRule* DSLTrans rule

and DSLTrans counterpart, and explains some relevant pre-/post-condition contracts for this transformation.

Section IV focuses on the higher-order transformation that automatically transforms declarative ATL specifications into their semantically-equivalent DSLTrans counterparts.

Performance results obtained from applying our tool to a number of examples, including a transformation obtained from our industrial partner, are presented in Section V. These results are discussed within the section and show that our technique is feasible, even for sizable transformations.

Finally, we wrap-up in Sections VI and VII by describing related work and presenting our conclusions and future work.

II. DSLTrans Model Transformation Language

DSLTrans is a visual graph-based and rule-based model transformation engine that has two important properties enforced by construction: all its computations are both *terminating* and *confluent* [7]. These properties stem from the fact that DSLTrans does not allow unbounded loops during execution, making it a Turing-incomplete computing language [7]. Besides their obvious importance in practice, *termination* and *confluence* were instrumental in the implementation of our verification technique for pre-/post-condition contracts.

Model transformations are expressed in DSLTrans as sets of graph rewriting rules, having an upper part (named *MatchModel*), a lower part (*ApplyModel*) and, optionally, negative application conditions. The main construction used in the scheduling of model transformation rules in DSLTrans is a *layer*. Each model transformation rule in the layer cannot match over the output of any other rule, and cannot modify the input graph during the rewriting phase (termed *out-place* execution). Layers are organized sequentially and the output model that results from executing a given layer is passed as input to the next layer in the sequence.

A DSLTrans rule can match over the elements of the input model of the transformation and also over elements that have been generated so far in the output model. Matching over elements of the output model of a transformation is achieved using a DSLTrans construct called *backward links*. Backward links allow matching over traces between elements in the input and the output models of the transformation. These traces are explicitly built by the DSLTrans transformation engine during rule execution.

For example, we depict in Figure 1 a rule in the DSLTrans language. When a rule is applied, the graph in the match part of the rule is searched for in the transformation's input model, together with the classes in the apply part of the rule that are connected to *backward links*, graphically represented with dotted lines. An example of a *backward link* can be observed in Figure 1 connecting the *HouseholdRoot* and the *CommunityRoot* match classes. During the rewrite part of rule application, the instances of classes in the apply part of the rule that are not connected to backward links, together with their adjacent relations, are created in the output model. The *UnionWomanRule* rule in Figure 1 will therefore create a *has* relation per matching site found. Although not present in this rule, copying object attribute values from the match to the apply part of the rules is also part of the DSLTrans language, as illustrated in Section III-C.

In addition to the constructs presented in the example in Figure 1, DSLTrans has several others: *existential matching* which allows selecting only one result when a match class of a rule matches an input model, *indirect links* which allow transitive matching over containment relations in the input model, and *negative application conditions* which allow the transformation designer to specify conditions under which a rule should not match. These constructs are not currently used in our verification approach, and the interested reader is referred to [7] for further information.

III. The Families-To-Persons Transformation

In order to highlight some of the relevant behavior of both ATL and DSLTrans, we present the *Families-to-Persons* transformation. This transformation is found in the ATL zoo, and has also been discussed in a number of related works on verification and testing [13].

A. Transformation Overview

The *Families-to-Persons* transformation operates on families made up of fathers, sons, mothers, and daughters, and transforms them into communities made up of male and female members. The source and target metamodels[1] are displayed in Figure 2. The source metamodel, *Households*, contains the elements *Family* and *Member*. These elements are connected by edges, which are typed *father*, *mother*, *son*, and *daughter*. The target metamodel, *Community*, represents a *Community* made up of *Man* and *Woman* elements (the *Person* class is abstract), which are connected to the *Community* by a *has*-typed edge.

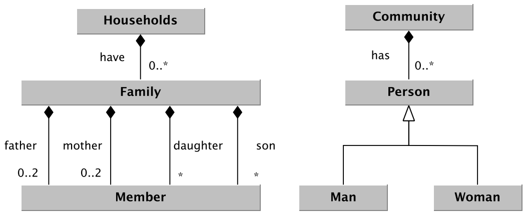

Fig. 2. *Households* and *Community* metamodels

[1] The authors acknowledge that these metamodels are quite artificial, and do not adequately represent real-world families or persons.

The purpose of the transformation is to convert elements of type *Member* into elements of type *Person*. Every *Member* with the role of *father* or *son* is transformed into a *Man*. Likewise, *Members* with the role of *mother* or *daughter* are transformed into a *Woman*. As for the *fullName* given to the *Person* created, it is composed of the *firstName* of the *Member* concatenated with the *lastName* of the *Family* to which the *Member* belongs.

B. The Families-To-Persons ATL Transformation

Listing 1 shows the *Families-to-Persons* ATL transformation. Note that for simplification purposes only the rules involving *father* and *mother* relations are shown.

This transformation is slightly different from the one in the ATL Transformation zoo [1]. In order to consider a higher subset of features of the declarative part of ATL, we have included more than one in-pattern element in some rules. This increases the complexity of internal traces, since now *Persons* are not only created from *Members*, but from the combination of *Members* and *Families*. As mentioned, the intention of this transformation is to translate *Households* units in the source model into *Community* units in the target model. The operation of this transformation, as well as all transformations in ATL, is split into two major steps: creating objects and setting values. Note that this two-step process, however, is not explicit in ATL, and is described here to reduce the conceptual delta between ATL and DSLTrans.

Listing 1. A portion of the *Families-to-Persons* ATL Transformation

```
module Families2Persons;
create OUT : Persons from IN : Families;

rule Households2Community { -- R1
 from
  hh: Families!Households
 to
  c : Persons!Community (
   has <- hh.have->collect(f | thisModule.
    resolveTemp(Tuple{mem=f.father,fam=f}, ’m’)), --B11
   has <- hh.have->collect(f | thisModule.
    resolveTemp(Tuple{mem=f.mother,fam=f}, ’w’)) --B12
 )}

rule Father2Man { -- R2
 from
  mem : Families!Member, fam : Families!Family
        (fam.father=mem)
 to
  m : Persons!Man (
   fullName <- mem.firstName + fam.lastName --B2
  )}

rule Mother2Woman { -- R3
 from
  mem : Families!Member, fam : Families!Family
        (fam.mother=mem)
 to
  w : Persons!Woman (
   fullName <- mem.firstName + fam.lastName  --B3
 )}
```

The first step of the transformation is to create target objects and trace links, the latter being created implicitly and automatically by ATL.

- *R1 - Households* elements are transformed into *Community* elements. Traceability links between them are implicitly created.
- *R2* and *R3* - *Members* who are connected by *mother* and *father* links to a *Family* are transformed into *Man* and *Woman* elements, respectively. Trace links between pairs of *Family-Member* and *Man* and between *Family-Member* and *Woman* are implicitly created.

In the second step, target object feature values are set. These are the rule components marked *B11*, *B12*, *B2*, and *B3* in Listing 1. For example, B11 sets the *has* relationship between the *Community* and a *Person*. The way the appropriate *Person* is selected is internally resolved by using the trace links. In particular and to explicitly show the resolving mechanism, we make use of the ATL *resolveTemp* operation, which makes it possible to point to any of the target model elements generated from a given (sequence of) source model elements by an ATL rule. As for the *Persons* that are created, their names are set to be the *firstName* of the *Member* from which it is generated, plus the *lastName* of the *Family* to which the *Member* belongs, as we can see in *B2* and *B3*.

C. DSLTrans Representation

An excerpt of the DSLTrans transformation corresponding to the ATL *Families-to-Persons* transformation shown in Listing 1 is displayed in Figure 3. This excerpt shows three out of five rules. Rule *Households2Community* corresponds to rule R1 in Listing 1, *Father2Man* corresponds to rule R2 and *UnionFather* to binding B11.

The process of constructing a DSLTrans transformation from an ATL one is described in the next section. For now note that DSLTrans specifications obtained from ATL always include only one rule per layer, meaning all rules execute sequentially. This is due to the sequential semantics of ATL that we replicate in DSLTrans. Also, note that rule *Father2Man* in Figure 3 performs an attribute copy from the match to the apply part of the rule, graphically represented with arrows from the *ApplyModel* to the *MatchModel*. More precisely, it fills in the *fullName* attribute of the generated *Man* instance by concatenating the *firstName* and *lastName* attribute values found respectively in the *member* and *family* instances identified by the match part of the rule. This operation replicates the concatenation of the *firstName* and the *lastName* occurring in the rule component marked *B2* in Listing 1.

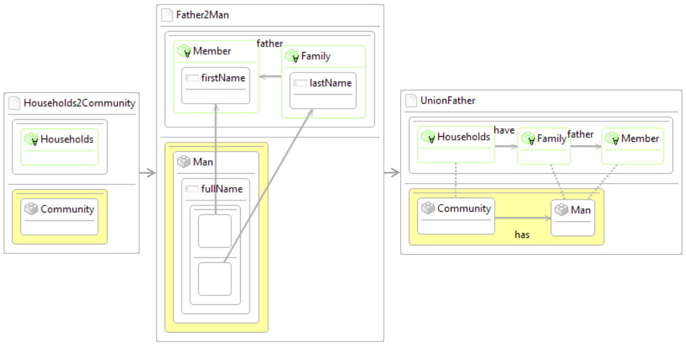

Fig. 3. A portion of the *Families-to-Persons* DSLTrans transformation

D. Proving Contracts

Given a transformation written in the DSLTrans transformation language, our contract prover can prove whether pre-/post-condition contracts will hold or not hold on all executions of this transformation. If a contract holds, then whenever the pre-condition matches over an input model, then the post-condition will match over the corresponding output model.

Contracts are proved through a process that first symbolically constructs all possible executions of the transformation, resulting in a set of so-called *path conditions*. Each path condition will represent a set of concrete executions of the transformation, where each concrete execution is an input/output model pair.

Our proving algorithm begins by generating one empty path condition, representing the case where no rules in the transformation have been executed. Then, each rule in each layer is examined, and its *MatchModel* and *ApplyModel* graphs are combined with the graph of each path condition generated thus far. As each layer in the transformation is considered, the set of path conditions will grow to represent all allowed combinations of rules. As rules may depend on each other because of backward links, such dependencies are verified by the path condition generation algorithm in order to exclude impossible rule combinations. The final set of path conditions produced by the algorithm will then abstract the infinite set of all concrete transformation executions. This is further described in [17], along with a formal discussion of the validity and completeness of this work.

Pre-/post-condition contracts form an implication, which needs to be checked for each path condition generated for the transformation by the above algorithm. In broad terms, a contract holds on a path condition if either the contract's pre-condition cannot be isomorphically found in the path condition, or the contract's pre-condition together with its post-condition can be found in the path condition. The contract does not hold on the path condition if its pre-condition can be isomorphically found in the path condition but its post-condition cannot. Finally, a contract holds for a transformation if it holds for all of its generated path conditions. Contracts are formally described in [17], while extensive discussion of the contract language is found in the PhD thesis of Selim [19].

For example, Figure 4 describes a contract that we want to prove over all transformation executions. The statement it describes is: *'a family with a father, mother, son, daughter should always produce two men and two women in the target community'*. Note that the traceability links in the contract require that the output elements be generated from the attached input elements. Our contract prover is then able to prove whether or not this contract will hold for all transformation executions, and produce any counter-examples if they occur.

Our contract language also allows reasoning about the attributes of elements in the models. Figure 5 describes a contract determining if the full name of the produced *Person* has been correctly created from the last name of the *Family* and the first name of the *Member*.

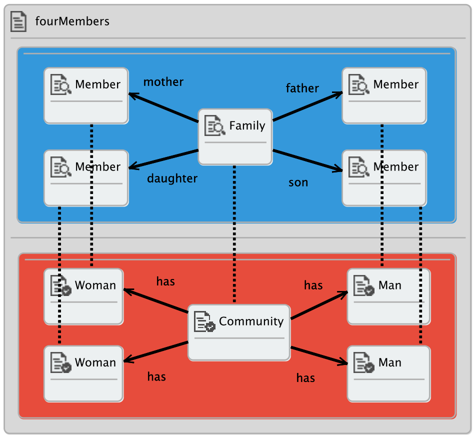

Fig. 4. A contract to verify that two *Woman* and two *Man* elements are correctly produced from the corresponding *Members*

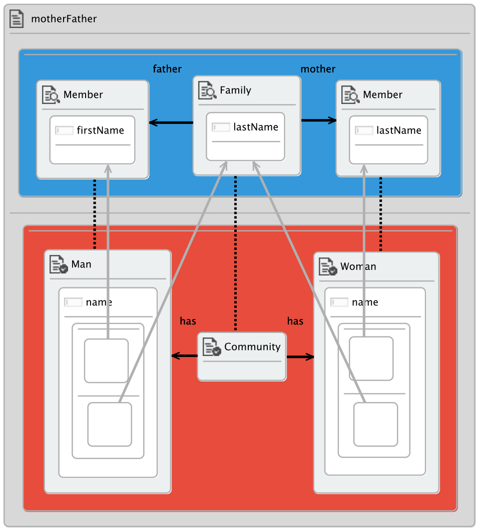

Fig. 5. A contract to verify proper construction of the *name* attribute

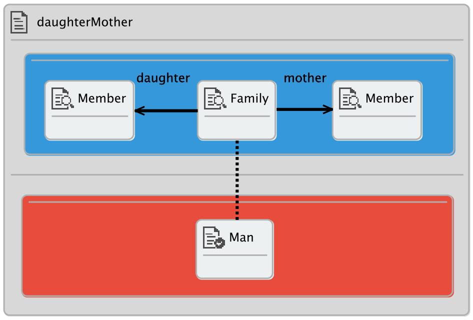

Fig. 6. A contract to verify whether a *Man* element will be produced from a *Family* containing a *daughter* element and a *mother* element - this contract will not hold

Figure 6 describes a contract that does not hold over all executions of the transformation. The contract's statement is, *'a family with a mother and a daughter will always produce a community with a man'*. It is easy to see that the input model

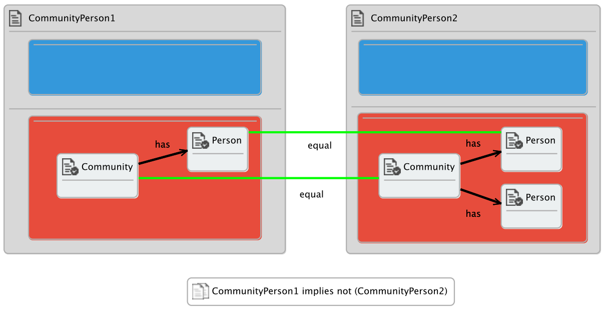

Fig. 7. Using propositional logic to express contracts

which contains only mother and daughter elements should not produce a man in the target community. Our contract prover will then find multiple counter-examples which cause the contract to not hold.

Contracts can also be combined using propositional logic and binding sites to enhance the expressiveness of the contract language [20], [19]. The last contract presented for the *Families-to-Person* transformation, in Figure 7, demonstrates the use of this propositional logic in our contract prover. Note the implication between the two contracts. For each path condition where the first contract holds, the second contract must not hold and the match sites for the elements connected by the *equal* relation must be the same. This contract's statement is *'If a Community is connected to a Person element, that Community is connected to only that one Person element'*.

Note that the contract language we present in this paper relies only on constructs that are found in the input and output metamodels, plus traceability links. Given that both ATL and DSLTrans operate on EMF metamodels, the contract language can thus be used seamlessly to describe pre-/post-conditions we wish to check on either ATL or DSLTrans transformations. This fact is a major advantage for our work: contracts can be expressed exactly in the same language and have the same semantics for both an ATL transformation and its semantically equivalent DSLTrans representation.

IV. Mapping ATL into DSLTrans

In this section we first present the features of the declarative part of ATL that we consider for the translation to DSLTrans. Then, we describe the mapping between ATL transformations and DSLTrans transformations.

A. ATL Subset Selected

In this first version of our translator from ATL to DSLTrans we have considered almost the complete set of features available in the declarative part of the ATL language. In Table I we show the features that we consider and those that we do not. The *using block* that can be used in ATL rules is an optional mechanism for declaring local constants. Consequently, the same rule can be written without using this block. Regarding helpers, they can be seen as simple OCL queries that are reusable throughout the transformation. For this reason, once again, equivalent rules can be written with and without helpers.

TABLE I
Features of Declarative ATL considered

Matched Rules	✓	Filters	✓
Lazy Rules	✓	OCL Expressions	✓
Several Bindings	✓	Helpers	×
Several *InPatternElements*	✓	Conditions	×
Several *OutPatternElements*	✓	Using Block	×

Finally, conditions, such as *if_then_else_endif* expressions that can be written in bindings, are not considered in our approach either. In the case where the condition checks something on the matching element of the ATL rule, then such a rule could be divided into two rules where the information about the condition is added in the filters, so that including the condition in the binding is avoided. An example of an ATL transformation containing the non-considered features and an equivalent one considering them can be found on our website [2].

Consequently, the fact that our current prototype does not consider the three features mentioned above does not impose significant limitations on the ATL transformations that can be translated into DSLTrans, since we can avoid having these three features by writing the ATL transformations slightly differently. This makes our current implementation sufficiently powerful to be of interest. In any case, we plan the inclusion of these features in future versions of our mapping.

B. Mapping between ATL and DSLTrans

In order to map ATL onto DSLTrans, we must explicitly represent the semantics of ATL in DSLTrans. This includes using backward links to make explicit in DSLTrans the implicit binding step in ATL for resolving associations between objects created in the transformation. We will explain the mapping we have established by using the *Families-To-Persons* case study shown in Listing 1 as a running example, whose partial DSLTrans representation is shown in Figure 3. The process is divided into two steps.

In the first step, every matched rule in ATL is translated into a rule in DSLTrans. Matched rules are declaratively matched by the ATL engine, so they are not called explicitly from anywhere. The order of these rules in DSLTrans does not matter, since they are independent from each other. Consequently, we choose the same order as in which they appear in the ATL transformation.

The match part of the created rules contains the elements appearing in the *from* part of the ATL rules. If the ATL rule has a filter, then some more elements and associations may appear in the match part in order to satisfy the conditions of the filter. An example of this are the *Member* and *Family* elements in rule *Father2Man* in Figure 3, which are linked by association *father* as indicated in the filter of R2 in Listing 1.

In the apply part of the rules created, there is an element for each element created by the ATL rule (in the *to* part). As well, associations are created between the elements when this is specified in the ATL rule.

If there are bindings in the elements created in the ATL rule that are initializing attributes (not references), then these attributes must appear in the elements created in the apply part. Besides, attributes must also appear in the elements in the match part if their value is used to initialize the values of the attributes in the apply part. An example are attributes *fullName* and *lastName* of *Member* and *Family* in rule *Father2Man*, which are used to initialize attribute *fullName* of *Man*. Note that, in this step, bindings that initialize references are ignored, such as bindings B11 and B12 in rule R1.

In the second step, a rule in DSLTrans is created for every binding that initializes the value of a reference in the ATL transformation. These rules are independent from each other, so the order does not matter. However, they must go after the rules created in the first step in order to properly utilize backward links as rule dependencies.

An example of such rules is the rule *UnionFather* in Figure 3, which corresponds to binding B11 in Listing 1. In these rules, the elements that appear in the match part are the elements that are traversed in the navigation of the binding, which is written in OCL, plus the associations between them. In our example, the binding includes types *Households* (*hh* is of type *Households*), *Family* and *Member*, and associations *have* and *father*. The apply part contains the association that the binding in ATL is initializing, *has* in our example, plus the elements representing the source and target classes of such association. These elements must have been created in previous rules – created in the first step of the mapping. In fact, backward links are included in the rule in order to indicate from which element(s) in the match part the element in the apply part was created. We see the use of these backward links in rule *UnionFather*, where they link the elements that have been created in the previous two rules.

Lazy rules are used in ATL when we want a reference to point to a concrete element that we create. They give a hybrid flavor (mix of declarative and imperative) to the declarative part of ATL. We also consider them in the second step of the mapping. When a lazy rule is called, we include the elements that are created by the lazy rule in the apply part created from the binding that is calling the lazy rule.

The mapping between ATL and DSLTrans has been implemented with a higher-order transformation (HOT) developed in ATL and available on our website [2]. It is composed of two main matched rules, each of which realizes one of the two steps explained.

V. Results and Discussion

In this section we present an evaluation of our higher-order transformation and contract-proving technique. In particular, we are interested in the following research questions:

- RQ1: Is our technique applicable to a variety of ATL transformations?
- RQ2: How does the time and memory usage of the contract prover differ for each of our case studies?
- RQ3: Given a particular contract, can we reduce the time taken for contract proving?
- RQ4: Does the output of our higher-order transformation differ significantly from a hand-built transformation?

A. Study Setup

This section will describe the case studies used in order to answer our research questions. Our intention is to verify transformations of different sizes and 'shape' in order to reason about how the characteristics of the transformation affects our contract prover.

1) Families-To-Person Transformation: The *Families-to-Persons* transformation described in Section III involves a number of interesting concepts with regards to our verification work. In particular, the rules producing elements in the output model are non-trivial, as those elements have their attributes set through manipulation of the attributes in the input model. This case study tests our technique's ability to correctly transform these attribute-setting rules and then prove contracts on these transformations.

As well, this case study is technically challenging to prove contracts on, as multiple elements in the transformation's rules are similar. For example, multiple rules contain a *Family* element. In order to prove contracts, our contract prover must be able to correctly 'disambiguate' these elements. That is, if there are similar elements in two rules, the contract prover must consider whether these elements match over separate elements in the input model, or over the same element.

2) Copier Transformations: The following 'copier' transformations were taken from the ATL zoo [1], and transformed by the HOT into DSLTrans transformations. The purpose of these transformations is to copy elements from the input model into the output model, which may be useful in transformation chaining. The *ER-Copier* transformation's input and output metamodel is the entity-relationship metamodel, while the *ECore-Copier* transformation operates on a subset of the ECore metamodel.

The first copier transformation, *ER-Copier*, is relatively small, composed of only five ATL rules which are transformed into nine DSLTrans rules by the higher-order transformation. As is a common structure for these 'copier' transformations, the first rules in the transformation copy input elements into the output model. Then, the following rules create the associations between elements.

The second copier transformation, *ECore-Copier*, is a larger transformation. The ATL representation has 11 rules, and the DSLTrans version has 24 rules. Therefore, this case study represents a larger test of our technique. Indeed, thousands of potential rule combinations are created, directly addressing the scalability research question (RQ2).

The contracts we wish to prove on these copying transformations determine if elements have been correctly copied and combined by the transformation. For example, Figure 8 and Figure 9 present two contracts we wish to prove on the *ECore-Copier* transformation. The statement for Contract 1 is, *'All bi-directional associations (represented by two inverse EReferences instances) between EClass instances should have the same end points, i.e., the EClass instances should have equivalent names.'*, while the statement for Contract 2 is *'If there is an EStructuralFeature instance in the target model, it must have the equivalent EClass instance as a container as*

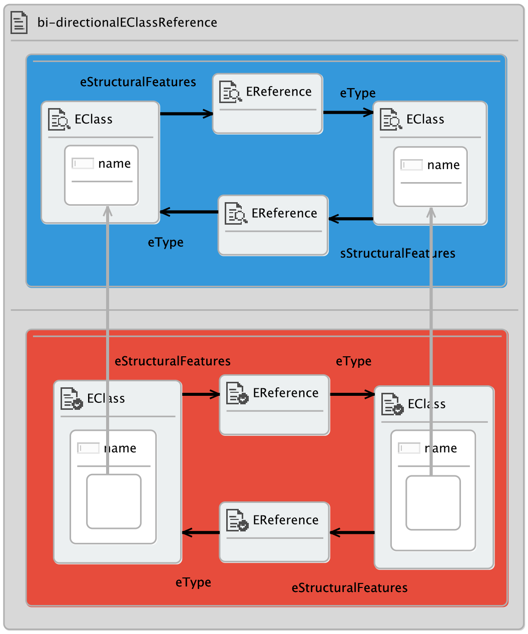

Fig. 8. Contract 1 for the *ECore-Copier* transformation

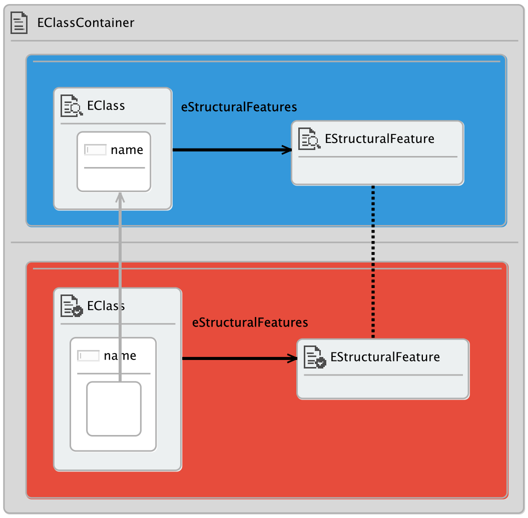

Fig. 9. Contract 2 for the *ECore-Copier* transformation

the corresponding source model EStructuralFeature instance has in the source model.'

3) Sliced Transformations: We also examined the output of our HOT on a larger version of the *ECore-Copier* transformation. This transformation is composed of 15 ATL rules, which was transformed into 63 DSLTrans rules by the HOT. Due to the exponential effect on contract proving time and memory of adding more rules to a transformation, we considered the time to verify contracts on this transformation to be infeasible. Therefore, we implemented an optimization which 'slices' the DSLTrans transformation to only consider those rules that are needed for a particular contract to be proved, directly addressing the corresponding research question (RQ3).

To perform these experiments, we sliced the 63-rule *Large-ECore-Copier* DSLTrans transformation for the contracts presented in Figure 8 and Figure 9. Slicing was performed manually, though in future work we expect to automate this.

The first step in slicing was to examine the elements in the contract, and determine which rules could either match over or produce those elements. The second step was to determine if those rules required elements that were found in yet other rules, in an iterative process. This dependency analysis was performed very conservatively, and future work will attempt to optimize the process to eliminate more rules. The needed rules are placed into a new transformation, which is likely smaller than the original transformation. However, the number of rules in the slice depends on the particular elements involved in the contract and the rules. For example, slicing the *Large-ECore-Copier* transformation for Contract 1 produced a DSLTrans transformation with 13 rules, while a slicing for Contract 2 produced a transformation with 17 rules. Note that the reason for the larger number of rules needed for Contract 2's slice is that the element *EStructuralFeature* in the contract is a supertype of multiple elements.

4) Industrial Transformation: In order to validate our higher-order transformation, we have applied it to an industrial transformation from earlier contract proving work [20]. The transformation in question takes models defined in a proprietary industrial metamodel, and translates them into the AUTOSAR metamodel, which is an industry standard. Therefore, this transformation is used for model-evolution purposes. Note that the contracts we wish to prove on this transformation are also reproduced from [20].

Our intention with this case study is two-fold. First, we are interested in comparing the time and memory consumption of this industrial example to the other transformations. Secondly, we will compare the time and memory usage of the contract prover when executed on the DSLTrans transformation produced by our higher-order transformation, and on the hand-built transformation found in that earlier work. These results will provide an answer to RQ4 of whether the DSLTrans representation generated by the higher-order transformation is sufficiently efficient for contract verification to replace the hand-built version.

B. Measures

In order to objectively answer our research questions, contract prover experiments were conducted for all case studies mentioned above. For each case study, the success of our contract prover rests on whether the contracts we have indicated hold or do not hold on all path conditions (as appropriate).

The following information was collected during the contract proving process for each case study:

- Number of rules in each transformation
- Number of path conditions produced by the contract prover
- Time required in order to generate all path conditions
- Number of contracts to be proved on the case study
- Time required to prove the contracts
- Total memory usage required by the contract prover

TABLE II
PERFORMANCE RESULTS

	ATL/ DSLTrans Rules	Path Conds. Gen.	Time (s)	Contracts Proved	Time (s)	Memory (MB)
Families-to-Person	5 / 9	52	1.54	4	31.45	45
ER-Copier	5 / 9	70	0.48	1	1.70	43
Ecore-Copier	11 / 24	57890	2894.44	1	1401.45	7800
Sliced Transformation (Contract 1)	15 / 13	73	3.50	1	9.11	72
Sliced Transformation (Contract 2)	15 / 17	28	0.95	1	0.46	71
Industrial (from [20])	5 / 7	3	0.07	9	0.16	43
Industrial (from HOT)	5 / 9	3	0.17	9	0.26	48

Note that the number of rules in the ATL transformation may be different from the DSLTrans transformation produced by the higher-order transformation. Therefore, both counts are reported.

The experiments were run on a 2013 Macbook Air with an Intel Core i5-4250U and 8 GB of RAM, running on Arch Linux and Python 2.7.9. Each experiment was conducted at least five times, with results averaged. Timing information was obtained by using the Python timing package *time*. Memory information was obtained using the `/usr/bin/time` command. Note that the memory usage information will also record the space overhead required by the Python interpreter.

All the artifacts used for our experiments can be found on our website [2].

C. Results

Table II shows the performance results for proving contracts on our case studies. We shall now discuss these results in the context of each of our research questions.

1) RQ1: Applicability of the Technique: To answer our first research question, we have tested our contract prover on a number of transformations of varying sizes and purposes. Most of our transformations were sourced from the ATL zoo, as well as one transformation from our industrial partner. For each case study, contracts we expected to hold were successfully proved. For other contracts, which do not hold in all cases, counterexamples were produced that indicate the exact combination of rules where the contract is not guaranteed to hold. For example, we attempted to prove the *daughterMotherProp* contract (seen in Figure 6) on the *Families-to-Person* transformation. Our contract prover correctly indicated that for input models that only contain *daughter* and *mother* elements, it is not guaranteed that there is a *Man* element in the output model.

Success of our contract prover on these case studies lets us conclude that we can apply our technique to a variety of ATL transformations.

2) RQ2: Time and Memory Characteristics: To answer our second research question, we refer to the results in Table II which contains the performance results of our case studies.

Note that while the number of path conditions generated is certainly dependent on the number of DSLTrans rules in the transformation, there is not a linear formula that can be applied. For example, the industrial transformation produced fewer path conditions than the equally-sized *ER-Copier* transformation. This discrepancy is due to rules in the industrial transformation completely overlapping with each other (as described in [20]). Therefore, the exact number of path conditions produced depends on the complex way in which rules combine with each other. As well, the time taken for path condition generation is also affected by the size of the rules themselves. Our path condition generation is implemented using graph-matching and rewriting, meaning that larger rules will take longer to combine [17].

The memory usage of our contract prover is dependent upon the number of DSLTrans rules in the transformation, and on the number of path conditions that are created. Note that for small transformations with a few DSLTrans rules, the memory usage is around 45 MB, which is consistent with the overhead to run the Python scripts. For larger transformations which produce thousands of path conditions, significant memory usage is seen. For example, the *ECore-Copier* transformation produces over 50,000 path conditions which consume about 7.8 GB of memory. We note that while this consumption of resources is feasible for a desktop or laptop computer, future work will focus on further efficiency improvements.

Overall, our approach stays within a modest time and memory budget. The smaller transformations have their path conditions generated within a few seconds, and have their contracts proved within a minute. Note that the time to prove contracts is proportional to both the number of path conditions generated for a transformation, as well as the number of contracts to be proved on that transformation.

3) RQ3: Reducing Contract Proving Time: As can be seen with the *ECore-Copier* transformation results, larger transformations create many more path conditions, straining time and memory budgets. This research question examines the possibility of 'slicing' large transformations based on two different contracts.

The results in Table II show the dramatic reduction in contract proving time when slicing is performed, as described in Section V-A3. The original transformation of 15 ATL rules and 63 DSLTrans rules is sliced into 13 DSLTrans rules for Contract 1, and 17 rules for Contract 2. This lowers the contract proving time from a number of hours to less than 15 seconds, as fewer path conditions have to be built and then matched by the contract.

As mentioned, this slicing procedure was performed manually for these experiments. However, as it is so effective in improving performance, our future work will focus on improving and automating this technique.

4) RQ4: Higher-Order Transformation: Our last research question investigates our use of an automatic higher-order transformation to generate DSLTrans transformations from

ATL transformations. In particular, we are interested in whether our contract prover is more efficient on a hand-built transformation, or whether the automatic transformation suffices in its place. The results in Table II show that while path condition generation time and contract proving time have increased, the same contracts held in both cases. Therefore we conclude that the HOT produces a transformation that is a sufficient replacement for a hand-built transformation in the context of proving contracts. Future work will attempt to generalize this result.

Further optimizations of the produced transformation may also be possible. In particular, the DSLTrans representation of ATL transformations generated by the HOT contains more rules than the hand-built transformation, with only one rule in each layer in the transformation. We are currently investigating whether there is a potential for optimization by producing fewer rules, or by placing independent rules into the same layer in a pre-processing step.

D. Threats to Validity

This subsection will discuss what we believe to be the major threats to the validity to our work.

The higher-order transformation has not been formally verified. Thus, we cannot be completely sure that the DSLTrans transformations that are automatically produced are directly equivalent to the original ATL transformation. However, two arguments can be made for the HOT's correctness. The first is that the HOT is relatively simple, as explained in Section IV. It consists of two steps: first creating the rules that generate the output elements and then creating the rules that generate the relations between output elements. This two-step approach makes ATL's semantics explicit, and makes the DSLTrans transformations generated by the HOT easily understandable as well as traceable back to their original ATL specifications. Second, we have compared the contract proof results between a transformation created by hand [20], and the corresponding transformation generated by our higher-order transformation. The results were the same, with similar proving times and memory usage. For future work, we are interested in verifying the higher-order transformation itself using the contract prover we present here.

Scalability is always an issue when exhaustive approaches such as ours are proposed. We have shown with our experiments that our contract prover scales to reasonably sized transformations when the slicing technique is used. However, more experiments with large transformations and contracts involving many elements are necessary to confirm our positive results on the usability and scalability of our technique.

DSLTrans is a Turing-incomplete computing language, having limited expressiveness. This means that ATL transformations that use the refining mode for realizing in-place transformations or imperative constructs cannot in general be translated into DSLTrans to be verified by our approach. However, our technique can be used to verify the declarative subset of ATL, including the class of out-place transformations that is so often used in practice. We are thus confident our technique is usable for a large class of real-world problems.

VI. Related Work

There has been already an extensive work on verifying different aspects of model transformations, e.g., cf. [4] for a survey. With respect to the contribution of this paper, we summarize previous contributions for checking different kind of contracts for model transformations whereas the concrete approaches range from testing to verification approaches.

In [13], [22] the authors describe a method where 'Tracts' can be specified for model transformations. Tracts define a set of constraints on the source and target metamodels, a set of source-target constraints, and a tract test suite, i.e., a collection of source models satisfying the source constraints. The accompanying TractsTool can then automatically transform the source models into the target metamodel, and subsequently verify that the source/target model pairs satisfy the constraints. The advantages of this are that the approach is not computationally-intensive, as tests can be narrowly focused in a modular way. Besides the Tracts approach, there are several other approaches supporting the testing of model transformations based on different kind of contracts such as model fragments [18], graph patterns [15], [6], Triple Graph Grammars (TGGs) [23], dedicated testing languages [16], [10], or as used in Tracts OCL constraints [9], and even a combination of these mentioned approaches [11]. While these mentioned approaches resort to black-box based testing, there are also approaches which allow for white-box based testing of model transformations such as [14].

In contrast to testing approaches, the presented approach in this paper allows for contracts to be proved for all possible transformation executions, i.e., for all possible input models. However, we also keep the same implication idea: the pre-condition of a property sets constraints on the input models to the transformation, and then, the post-condition defines constraints on the output model.

Previous work has also proposed the idea of transforming ATL to formal domains. The work of [21] describes a formal semantics for ATL, such that ATL transformations can be expressed in the formal language Maude. Once expressed in Maude, properties can then be verified over the execution of this transformation, such as reachability of particular states, or that no more than one rule is matched on each source element. In our work, we transform the ATL transformation into the DSLTrans transformation language to prove transformation contracts which is not in the scope of [21].

The work in [8] automatically transforms transformations in a number of transformation languages (such as ATL) to OCL. As well, similar to our system, the invariant, pre- and post- conditions are described in a graph format. However, in [8] the counter-example conditions for each property are generated. Then a model finder generates a possible counter-example model, before the system determines if the model can be satisfied or not. Note that due to incomplete searching of the model space, the model finder may not find every counter-example. In contrast, our system works by matching the property onto path conditions, which abstracts all possible transformation executions. Thus, our property prover can give a stronger proof. The work by Anastasakis et al. [5] transforms model transformations to Alloy in order to verify if given assertions, i.e., properties, hold for the given transformations. If no target model is found by Alloy for a given source model, the assertion does not hold. As Alloy needs bounds for the model search, models outside the given bounds are not found. In [12] the authors are checking different kinds of model transformation properties based on OCL and the usage of KodKod which requires concrete bounds for property proving.

To the best of our knowledge, in this paper we presented the first approach to fully prove properties defined as contracts for model transformations expressed in declarative ATL.

VII. Conclusion

In this paper, we have presented our novel technique to fully verify pre-/post-condition contracts on declarative ATL transformations. This approach is centered around transforming ATL transformations into DSLTrans, our reduced-expressiveness transformation language. Then, our path condition generator is able to produce a set of path conditions, which represent all possible transformation executions. Contracts are proved to either hold or not hold on each path condition, and thus on all transformation executions.

This paper has also presented a number of case studies designed to answer our four research questions. Results indicate that our contract prover is applicable to a variety of ATL transformations, and that contracts can be proved using a feasible amount of time and memory. As well, we have also introduced a 'slicing' technique, which selects only the rules which are needed to prove a particular contract. This results in a significant decrease in contract proving time. Finally, we investigated whether our automatically-produced transformation is a suitable replacement for a hand-built transformation in contract proving.

Our future work will attempt to address any limitations of this work. In particular, we aim to produce a tool that can be used off-the-shelf to prove properties about a class of existing ATL transformations, fully automatically, by using the DSLTrans language as a hidden back-end. Already, we have begun work to integrate our approach into the Eclipse environment. Another ongoing concern of ours is the time and space requirements to prove contracts on large transformations. As mentioned, we intend to improve and automate the slicing algorithm, as well as investigate other implementation speedups.

Acknowledgements

The authors warmly thank the reviewers for their insightful comments on this submission, as well as Gehan Selim and Cláudio Gomes for their work on the implementation of the contract prover. Bentley James Oakes and Levi Lúcio are researchers working for the NECSIS project, funded by the Automotive Partnership Canada.

The work of Javier Troya and Manuel Wimmer is funded by the European Commission under ICT Policy Support Programme, grant no. 317859 as well as by the Christian Doppler Forschungsgesellschaft and the BMWFW, Austria.

References

[1] ATL Zoo. http://www.eclipse.org/atl/atlTransformations.

[2] ATL2DSLTrans Artifacts. http://msdl.cs.mcgill.ca/people/levi/files/MODELS2015.

[3] Atlas Transformation Language – ATL. http://eclipse.org/atl.

[4] M. Amrani, L. Lucio, G. M. K. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *ICSTW*, pages 921–928, 2012.

[5] K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In *Proc. of MoDeVVa*, 2007.

[6] A. Balogh et al. Workflow-driven tool integration using model transformations. In *Graph Transformations and Model-Driven Engineering*, pages 224–248. Springer, 2010.

[7] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. Dsltrans: A turing incomplete transformation language. In *Software Language Engineering*, pages 296–305. Springer, 2011.

[8] F. Büttner, M. Egea, E. Guerra, and J. De Lara. Checking model transformation refinement. In *Theory and Practice of Model Transformations*, pages 158–173. Springer, 2013.

[9] E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL contracts for the verification of model transformations. *ECEASST*, 24, 2009.

[10] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo. EUnit: A Unit Testing Framework for Model Management Tasks. In *MODELS*, pages 395–409. Springer, 2011.

[11] P. Giner and V. Pelechano. Test-Driven Development of Model Transformations. In *MODELS'09*, pages 748–752. Springer, 2009.

[12] M. Gogolla, L. Hamann, and F. Hilken. Checking Transformation Model Properties with a UML and OCL Model Validator. In *Third International Workshop on Verification of Model Transformations*, pages 16–25, 2014.

[13] M. Gogolla and A. Vallecillo. Tractable model transformation testing. In *ECMFA*, pages 221–235. Springer, 2011.

[14] C. A. González and J. Cabot. ATLTest: A White-Box Test Generation Approach for ATL Transformations. In *MODELS*, pages 449–464, 2012.

[15] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Automated verification of model transformations based on visual contracts. *Autom. Softw. Eng.*, 20(1):5–46, 2013.

[16] D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *GaMMa'06*, pages 13–20. ACM, 2006.

[17] L. Lúcio, B. Oakes, and H. Vangheluwe. A technique for symbolically verifying properties of graph-based model transformations. Technical report, Technical Report SOCS-TR-2014.1, McGill U, 2014.

[18] J.-M. Mottu, B. Baudry, and Y. L. Traon. Model transformation testing: oracle issue. In *ICSTW*, pages 105–112. IEEE, 2008.

[19] G. M. Selim. *Formal Verification of Graph-Based Model Transformations*. PhD thesis, Queen's University, 2015.

[20] G. M. Selim, L. Lúcio, J. R. Cordy, J. Dingel, and B. J. Oakes. Specification and verification of graph-based model transformation properties. In *Graph Transformation*, pages 113–129. Springer, 2014.

[21] J. Troya and A. Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10(5):1–29, 2011.

[22] A. Vallecillo, M. Gogolla, L. Burgueno, M. Wimmer, and L. Hamann. Formal specification and testing of model transformations. In *Formal Methods for Model-Driven Engineering*, pages 399–437. Springer, 2012.

[23] M. Wieber, A. Anjorin, and A. Schürr. On the Usage of TGGs for Automated Model Transformation Testing. In *Theory and Practice of Model Transformations*, pages 1–16, 2014.