Noname manuscript No.
(will be inserted by the editor)

Full Contract Verification for ATL using Symbolic Execution

Bentley James Oakes · Javier Troya · Levi Lúcio · Manuel Wimmer

June 28, 2016

Abstract The Atlas Transformation Language (ATL) is currently one of the most-used model transformation languages and has become a de-facto standard in model-driven engineering for implementing model transformations. At the same time, it is understood by the community that enhancing methods for exhaustively verifying such transformations allows for a more widespread adoption of model-driven engineering in industry. A variety of proposals for the verification of ATL transformations have arisen in the past few years. However, the majority of these techniques are either based on non-exhaustive testing or on proof methods that require human assistance and/or are not complete.

In this paper, we describe our method for statically verifying the declarative subset of ATL model transformations. This verification is performed by translating the transformation (including features like filters, OCL expressions, and lazy rules) into our model transformation language DSLTrans. As we handle only the declarative portion of ATL, and DSLTrans is Turing-incomplete, this reduction in expressivity allows us to use a symbolic-execution approach to generate representations of all possible input models to the transformation. We then verify pre-/post-condition contracts on these representations, which in turn verifies the transformation itself.

The technique we present in this paper is exhaustive for the subset of declarative ATL model transformations. This means that if the prover indicates a contract holds on a transformation, then the contract's pre-/post-condition pair will be true for any input model for that transformation. We demonstrate and explore the applicability of our technique by studying several relatively large and complex ATL model transformations, including a model transformation developed in collaboration with our industrial partner. As well, we present our 'slicing' technique. This technique selects only those rules in the DSLTrans transformation needed for contract proof, thereby reducing proving time.

Keywords Model transformation · ATL · Formal verification · Symbolic execution · Contracts · Pre-/Post-conditions

Bentley James Oakes
School of Computer Science, McGill University, Canada
E-mail: bentley.oakes@mail.mcgill.ca

Javier Troya
Department of Computing Languages and Systems, Universidad de Sevilla, Spain,
E-mail: jtroya@us.es

Levi Lúcio
fortiss GmbH, München, Germany,
E-mail: lucio@fortiss.org

Manuel Wimmer
Business Informatics Group, TU Wien, Austria,
E-mail: wimmer@big.tuwien.ac.at

1 Introduction

Model transformations have become the main means for manipulating models in model-driven engineering [12], as transformations are an excellent compromise between strong theoretical foundations and applicability to real-world problems [34]. In particular, model transformations allow for mathematical treatment based on foundations of graphs and graph transformations, and can natively manipulate domain-specific concepts expressed in metamodels.

For example, the Atlas Transformation Language (ATL) [4, 28] has come to prominence in the model-driven engineering community. This success is due to ATL's flexibility, support of the main meta-modelling

standards, usability that relies on strong tool integration with the Eclipse world, and a supportive development community.

Due to the importance of ATL in both the academic and the industrial arenas, the verification of ATL transformations is of prime importance. This is because the correctness of software built using model-driven engineering techniques typically relies on the correctness of operations executed using model transformations. As well, there is a strong demand for tools that allow the building of verified software, especially in industries where quality and safety standards have to be met.

In this paper we address this issue by detailing our technique for verifying visual pre-/post-condition contracts on ATL transformations. A contract is said to hold on a transformation if it holds on the transformation's input-output pairs. That is, for all input models where the contract's pre-condition holds, the contract's post-condition also holds in the corresponding output model produced by executing the transformation. Traceability constraints between the elements of the input and output models may also be required. Otherwise, the contract does not hold, and consequently, the transformation does not correctly implement the contract.

For example, this paper considers as running example an extended version of the well-known *Families-to-Persons* transformation from the ATL zoo [2], 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 (cf. Section 9). Our contract language is based on pre-/post-condition contracts, but also includes propositional logic operators for combining contracts. Section 5 includes a discussion of contracts, including examples and a summary of contract expressiveness.

We prove that contracts hold or not by translating ATL transformations into transformations defined in a model transformation language called DSLTrans [10]. 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 those input/output model pairs resulting from executing a given DSLTrans model transformation, or to not hold for at least one of those input/output pairs [32]. A fully automatic property prover based on this theory has been shown to be applicable to industrial problems [44].

In this paper we focus on verifying the declarative part of ATL, given the 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.

Please note that this article is an extension of a paper presented at the MoDELS 2015 conference [36]. Besides incremental advancements of the general verification approach, this article introduces four major extensions over the previous paper. First, a substantial subsection (cf. Subsection 4.2.2) has been added, which explains how Object Constraint Language (OCL) expressions are handled in the transformation from ATL to DSLTrans. Second, additional ATL features are now considered in the mapping, namely helpers and conditions (cf. Section 4.1). Third, the evaluation of the approach has been significantly improved. In particular, we now also compare the transformations produced by the HOT with hand-built transformations in two case studies of varying complexity. Finally, the slicing algorithm initially described in [36] has been further improved and is now presented in more detail in Section 7.

The presentation of our work is as follows: Section 2 briefly introduces the ATL and DSLTrans languages and their relevant constructs. Following this, Section 3 presents the extended *Families-to-Persons* ATL transformation, describes how it is executed, and presents its DSLTrans counterpart.

Section 4 provides a pseudo-code algorithm and execution example for the higher-order transformation that automatically transforms declarative ATL transformations into their semantically-equivalent DSLTrans counterparts.

Section 5 discusses our contract proving method. This includes the creation of the artifacts which represent transformation executions through symbolic execution, as well as a description of how contracts are proven using these artifacts. Some relevant pre-/post-condition contracts for the extended *Families-to-Persons* transformation are also described.

Performance results obtained from applying our tool to a number of transformations, including a transform-

ation obtained from our industrial partner, are presented in Section 6. These results are discussed within the section and show that our technique is feasible. In Section 7, our slicing algorithm is discussed, which allows the prover to select only those rules which are needed to prove a particular contract. Results presented in this section demonstrate the reduction in contract proving time.

Section 8 examines the transformations produced by our higher-order transformation (HOT) from ATL code, versus versions produced by hand in our earlier work. Both versions of one transformation are then used for contract proving, to examine the suitability of the HOT to replace the building of DSLTrans transformations by hand. Finally, we wrap-up in Sections 9 and 10 by describing related work and discussing threats to validity, our conclusions, and some thoughts on future work.

2 Preliminaries: ATL and DSLTrans

In this section, we introduce the model transformation languages used in this paper.

2.1 ATL

ATL is a textual rule-based model transformation language that provides both declarative and imperative language concepts. It is thus considered a hybrid model transformation language.

An ATL transformation is composed of a set of transformation rules and helpers. Each rule describes how certain target model elements should be generated from certain source model elements. There are two kinds of rules: *matched rules* and *lazy rules*[1]. Matched rules are automatically executed by the ATL execution engine for every match in the source model according to the input patterns of the matched rules. In contrast, lazy rules have to be explicitly called from another rule, which gives more control over the transformation execution.

The Object Constraint Language (OCL) is used all throughout ATL transformations as an expression language. A *helper* can be seen as an auxiliary OCL function, which can be used to avoid the duplication of the OCL code at different points in the ATL transformation.

Rules are mainly composed of an *input pattern* and an *output pattern*. The input pattern is used to match *input pattern elements* that are relevant for the rule. The output pattern specifies how the *output pattern elements* are created from the input model elements matched by the input pattern. Each output pattern element can have several *bindings* that can be used to initialize the values of the elements in the target model.

Please note that these initializations are performed in a second phase after a first phase where all output elements are created by the matched rules. This separation in two steps enables the initialization of target values independent from the execution order of the rules, as explained in detail in Section 3.3.

Listing 1 *Families-to-Persons* ATL Transformation Excerpt

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

rule Country2Community {
 from
  c: Families!Country
 to
  cmm : Persons!Community (
   persons <- c.families->collect(f|f.mothers),
   ...
  )}

rule Mother2Woman {
 from
  p : Families!Parent
   (p.family.mothers.includes(p))
 to
  w : Persons!Woman (
   fullName <- p.firstName + p.family.lastName
  )}
```

In Listing 1 we give a minimal excerpt of the *Families-to-Persons* transformation, which is fully explained in Section 3. In particular, two matched rules are defined. The first one transforms *Countries* into *Communities*, while the second one creates for each *mother* instance in the source model, a *Woman* instance in the target model. Please note that the second rule also has a filter for selecting from the set of *Parents* only the *mothers*. Bindings are used, for instance, to initialize the *persons* reference of *Communities*, by collecting all *mothers* from all *Families* of the matched *Country*.

For more information on ATL the interested reader is referred to [28].

2.2 DSLTrans

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* [10]. These properties stem from the fact that DSLTrans does not allow unbounded loops during execution, making it a Turing-incomplete computing language [10]. 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,

[1] We include ATL's *called rules* under lazy rules.

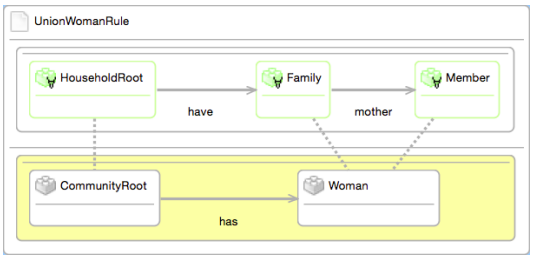


Fig. 1 An example of a DSLTrans rule

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 a layer cannot match over the output of any other rule in the same layer. As well, rules 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 executed, the graph in the *MatchModel* of the rule is searched for in the transformation's input model, together with the classes in the *ApplyModel* of the rule that are connected to *backward links*. An example of a *backward link* can be observed in Figure 1 as a dotted line connecting the *Country* and the *Community* match classes. During the rewrite part of rule application, the instances of classes in the *ApplyModel* of the rule that are not connected to backward links, together with their adjacent relations, are created in the output model.

For example, the *UnionWomanRule* rule in Figure 1 will match over a *Country* element connected to a *Family* element connected to a *Parent* element. If these elements are found in the input model along with the corresponding *Community* and *Woman* elements in the output model, then a *persons* relation will be created between those output elements.

Although not present in this rule, copying object attribute values from the *MatchModel* to the *ApplyModel* of the rules is also part of the DSLTrans language, as illustrated in Section 3.4.

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* for transitive matching over containment relations in the input model, and *negative application conditions* that 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 [10] for further information.

3 The Extended Families-to-Persons Transformation

As our running example we present an extended version of the *Families-to-Persons* transformation described in [36]. The original *Families-to-Persons* transformation can be found in the ATL zoo [2], and has also been discussed in a number of related works on verification and testing [25].

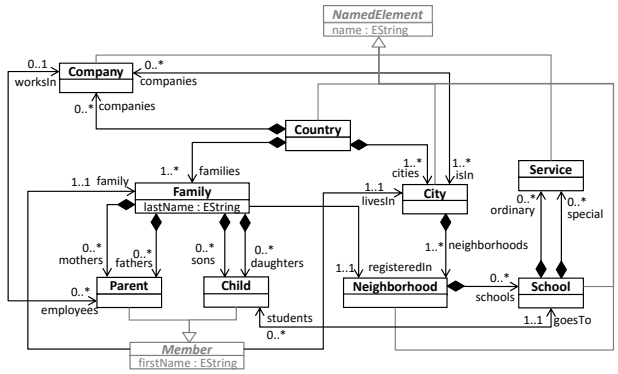
We chose this *Families-to-Persons* transformation as our running example for two reasons. First, it transforms domains whose concepts are easily understandable by anyone (cf. Section 3.1). Second, it has a certain degree of complexity since it uses many features available in the ATL language (cf. Section 3.2).

3.1 Transformation Domains

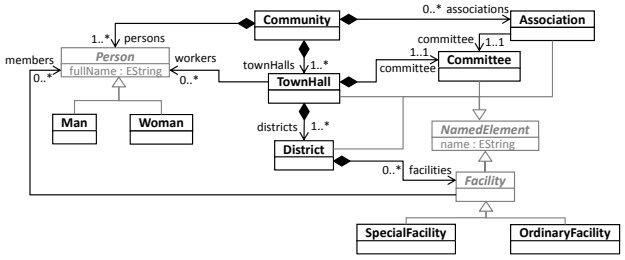
The input and output metamodels of this transformation are shown in Figure 2. Please note that abstract classes are depicted in grey color and with italic names, and inheritance relationships are depicted in grey.

The input metamodel, the *Families_Extended* metamodel, has the *Country* class as a root element. A *Country* is made up of *companies*, *families* and *cities*. A *Family* has a *lastName*, is *registeredIn* a *Neighborhood* and can have any number of *mothers* and *fathers*, who are *Parent*s and may, in turn, work in (*worksIn*) a *Company*. It can also contain any number of *sons* and *daughters*, who are *Child*ren, and every child *goesTo* a *School*. Both parents and children are *Member*s that have a *firstName*, belong to a *family* and each of them *livesIn* a *City*.

A *City* may contain *companies*, and a *Company*, in turn, can be present in (relationship *isIn*) several distinct cities. A *City* is composed of *neighborhoods*, and these can have *schools*, where several *students* are registered. Every *School* has *Service*s, and these may be *special*, for students with special needs, or simply



(a) *Families_Extended* metamodel



(b) *Persons_Extended* metamodel

Fig. 2 Metamodels of the *Families-to-Persons_Extended* transformation

offer *ordinary* services. Finally, countries, cities, companies, neighborhoods and schools have a *name* attribute, which is inherited from the abstract *NamedElement* class.

The output metamodel, *Persons_Extended*, is shown in Figure 2(b). The root class is *Community*, which is made up of *persons*, *townHalls* and *associations*. A *Person* has a *fullName* and can either be a *Man* or a *Woman*. An *Association* has a *Committee* that makes decisions. Every *TownHall* has a roster of *workers* (all the persons that are employed), hosts a *Committee* to make decisions, and also governs several *districts*. A *District* may contain several *facilities*, either of type *SpecialFacility* for those with special needs, or *OrdinaryFacility*. Each *Facility* may have registered several persons as *members*. Finally, associations, town halls, committees, districts and facilities have a *name* attribute.

3.2 Transformation Explanation

Listing 2 displays the ATL code for the *Families-to-Persons_Extended* model transformation, which is composed of 10 rules. In order to better explain the transformation and the mapping to the DSLTrans language (cf. Section 4.2.1), we have annotated many lines in the listing as follows:

- *Ri* stands for *rule i*
- *IPEij* for *in-pattern element j in rule i*
- *Fi* for *filter of rule i*
- *OPEij* for *out-pattern element j of rule i*
- *Bijk* for *binding k in simple out-pattern element j of rule i*

In ATL, the *from* part of rules is called an *in-pattern* and is made up of *in-pattern elements* and an optional *filter*. These in-pattern elements represent the elements from the input model that are matched by the rule, as long as they satisfy the filter. The *to* part of the rule is referred to as an *out-pattern* and is made up of *out-pattern elements* which represent the elements that are created in the output model. They contain *bindings*, which are used to initialize the features of the created elements. The feature initialized by a binding can be either an attribute or a reference to another created element.

In Listing 2, the transformation creates a *Community* from each *Country*, as specified by *R1*. The six bindings of *OPE11* are used to initialize references. Please note that when the same property representing a reference is initialized in more than one binding (property *persons* in *B111*, *B112*, *B113* and *B114* in our case), the result is the union of the elements retrieved in all the bindings.

Let us now focus on *B115*, which initializes the *townHalls* association of the created *Community*. It assigns the *cities* of the *Country* which match the expression (*c.cities*). However, due to the output metamodel, the *townHalls* association of the created *Community* in the output model cannot point to a *City* in the input model. In fact, what the association will actually be pointing to is the element in the output model created from the respective *City* element of the input model. Therefore, there must be a rule that creates something from *City* elements.

In our example, this is *R6*, which creates a *TownHall* from the matched *City*. Therefore, *B115* initializes the *townHall* association of the *Community* created in *OPE11* by referencing the *TownHall* elements created from the *City* elements referred to by the *Country* element matched in *IPE1*. To this purpose, ATL uses an internal trace mechanism, where the correspondences

between elements in the input and output models are tracked.

Listing 2 *Families-to-Persons_Extended* ATL Transformation

```
module Families2Persons_Extended;
create OUT:Persons_Extended from IN:Families_Extended;

rule Country2Community {                                    --R1
 from
  c: Families!Country                                     --IPE11
 to
  cmm : Persons!Community (                               --OPE11
   persons <- c.families->collect(f|f.fathers),           --B111
   persons <- c.families->collect(f|f.mothers),           --B112
   persons <- c.families->collect(f|f.sons),              --B113
   persons <- c.families->collect(f|f.daughters),         --B114
   townHalls <- c.cities,                                 --B115
   associations <- c.cities->collect(cty|cty.companies
    -> collect(cmp|Tuple{ct=cty,cm=cmp}))                 --B116
  )}

rule Father2Man {                                           --R2
 from
  p : Families!Parent                                     --IPE21
   (p.family.fathers.includes(p))                           --F2
 to
  m : Persons!Man (                                       --OPE21
   fullName <- p.firstName + p.family.lastName            --B211
  )}

rule Mother2Woman {                                         --R3
 from
  p : Families!Parent                                     --IPE31
   (p.family.mothers.includes(p))                           --F3
 to
  w : Persons!Woman (                                     --OPE31
   fullName <- p.firstName + p.family.lastName            --B311
  )}

rule Daughter2Woman {                                       --R4
 from
  ch : Families!Child                                     --IPE41
   (ch.family.daughters.includes(ch))                       --F4
 to
  w : Persons!Woman (                                     --OPE41
   fullName <- ch.firstName + ch.family.lastName          --B411
  )}

rule Son2Man {                                              --R5
 from
  ch : Families!Child                                     --IPE51
   (ch.family.sons.includes(ch))                            --F5
 to
  m : Persons!Man (                                       --OPE51
   fullName <- ch.firstName + ch.family.lastName          --B511
  )}

rule City2TownHall{                                         --R6
 from
  c : Families!City                                       --IPE61
 to
  th : Persons!TownHall(                                  --OPE61
   name <- c.name + '_TownHall',                          --B611
   workers <- c.companies -> collect(cmp|cmp.employees)
    -> flatten() -> select(em|em.livesIn=c),              --B612
   committee <- cmt,                                      --B613
   districts <- c.neighborhoods                           --B614
  ),
  cmt : Persons!Committee(                                --OPE62
   name <- c.name + '_TownHall_Committee'                 --B621
  )}


rule CityCompany2Association{                               --R7
 from
  ct : Families!City,                                     --IPE71
  cm : Families!Company                                   --IPE72
   (ct.companies.includes(cm))                              --F7
 to
  a : Persons!Association(                                --OPE71
   name <- ct.name + cm.name                              --B711
   committee <- thisModule.resolveTemp(ct,'cmt') --B712
  )}

rule Neighborhood2District{                                 --R8
 from
  n : Families!Neighborhood                               --IPE81
   (Families!Family.allInstances()
    -> exists(f|f.registeredIn=n))                          --F8
 to
  d : Persons!District(                                   --OPE81
   name <- n.name,
   facilities <- n.schools -> select(sch|sch.ordinary
    -> notEmpty()) -> collect
    (sch|thisModule.CreateOrdinaryFacility(sch)),--B811
   facilities <- n.schools -> select(sch|sch.special
    -> notEmpty()) -> collect
    (sch|thisModule.CreateSpecialFacility(sch))  --B812
  )}

lazy rule CreateOrdinaryFacility{                           --R9
 from
  sch : Families!School                                   --IPE91
 to
  of : Persons!OrdinaryFacility(                          --OPE91
   name <- 'Ordinary_Facility_Service_for_school_'
    + sch.name,                                           --B911
   members <- sch.students                                --B912
  )}

lazy rule CreateSpecialFacility{                            --R10
 from
  sch : Families!School                                   --IPE101
 to
  sf : Persons!SpecialFacility(                           --OPE101
   name <- 'Special_Facility_Service_for_school_'
    + sch.name,                                           --B1011
   members <- sch.students                                --B1012
  )}
```

For clarification purposes, Figure 3 shows the creation of an output model from an input model by applying the *Families-to-Persons_Extended* transformation. The left-hand side of the figure displays a model which conforms to the *Families_Extended* metamodel seen in Figure 2(a). The right-hand side of the figure presents the output model obtained, which conforms to the *Persons_Extended* metamodel (cf. Fig. 2(b)).

Note that attributes are ignored in Figure 3, and classes and references are coloured to improve the readability of the figure. Elements are also given an identifier to indicate to which type they belong (e.g., *Cmp1* is an element of type *Company*), and references have the same colour as their source element. In case of bidirectional references, the reference has the colours of both elements, but tags describing a specific end of the reference have the colour of the source element. For simplification and readability purposes, not all tags have been included (for instance, there are missing *livesIn* tags), although most of them are present. Since the attributes and references of the created elements are initialized with bindings, we have annotated the binding responsible for initializing each reference (again, attributes are ignored) in the output model.

The central section of Figure 3 shows the internal traces used by ATL that keep information of which elements in the output model are created from which ones in the input model and the rule responsible for doing so. We can see that the traces also keep information about the identifiers of the *in-pattern elements* and *out-pattern elements* of each rule, something specially useful for the implementation of the *resolveTemp* operation (cf. Section 3.3).

Figure 3 thus illustrates what has been described before, namely that *Comm1* is created from *Ctry1* by *R1*. Its *townHalls* reference is pointing to *TH1*, which has been created by *R6* from *Cty1*, which belongs to *Ctry1.cities*, and assigned by *B115*.

Both *R2* and *R3* take a *Parent* as input. Here, a filter is used to determine if the *Parent* is a *father* (*R2*) or a *mother* (*R3*). For instance, parents *Par1* and *Par2* are transformed to a *Man* with *R2*, while *Par3* is transformed to a *Woman* with *R3*. The same thing happens with *Child*ren. Bindings *B111* - *B114* are used to initialize the *person* reference for the *Community* created in *OPE11* for all parents and children. We see that an

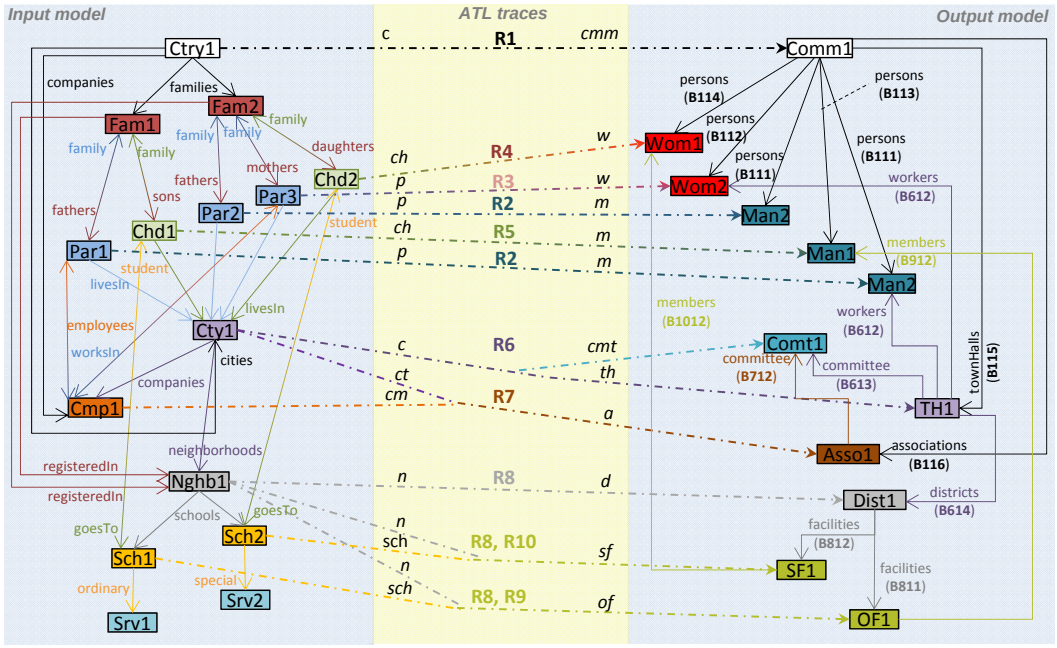


Fig. 3 Execution example of the *Families-to-Persons_Extended* transformation

OCL *collect* operation is necessary, since we need to retrieve elements of specific types.

B116 shows a special case where the *Tuple* operator is used. In this case, the *collect* operation retrieves pairs of {*City, Company*} elements, and the output elements created by these pairs are assigned to the *associations* reference. Therefore, there must be a rule that takes these pairs in the *in-pattern*. In our example, it is *R7*, composed of *IPE71* and *IPE72*, where *F7* makes sure that the *Company* is located in the *City*. As a result, following our example, *Asso1* is created from *Cty1* and *Cmp1*.

B612 uses both the *collect* and *select* operators to pick only those *employees* of the *companies* located in the matched *City* (*IPE61*) that actually live in such city. In our example, *Par1* and *Par3* are selected, so the *workers* reference of *TH1* points to *Wom2* and *Man2*, created from *Par1* and *Par3*, respectively.

Finally, it is worth mentioning the use of *lazy rules* (*R9* and *R10*). Lazy rules are only executed when they are called from other rules. This means that they will create elements only when they receive calls.

In our example, these rules are called from *R8*, and specifically from bindings *B811* and *B812*. Let us focus on binding *B811*. *R8* creates a *District* (*OPE81*) from a *Neighborhood* (*IPE81*) as long as there is at least one *Family* registered in that neighborhood (*F8*). Then, for initializing its *facilities* reference, in *B811* it *selects* those *schools* in the *Neighborhood* that have an *ordinary Service* and *collects* the result produced by lazy rule *CreateOrdinaryFacility*.

This lazy rule (*R9*) receives as parameter the selected schools. It creates an *OrdinaryFacility* (*OPE91*) from the *School* (*IPE91*) and initializes its *members* reference with the *students* of the school. In Figure 3, we represent a joint execution of *R8* and *R9*, since the latter is executed at the same time as the former due to its invocation. The same situation occurs with *R8* and *R10*.

3.3 ATL Semantics and Transformation Execution

The semantics of ATL define how an ATL transformation is internally executed. However, the ATL language has been described in the community in an intuitive and informal manner, by means of definitions of its main features in natural language [47]. This lack of rigorous description can easily lead to imprecisions and misunderstandings that might hinder the proper usage and analysis of the language, and the development of correct and interoperable tools. The other reference implementation of ATL is available as metamodels for the language and its virtual machine, and as a compiler from the language to the virtual machine and an interpreter for the virtual machine. The problem of this kind of implementation is that it is not abstract enough to provide meaningful semantics, and in an implementation-independent manner. Therefore, with the purpose of later describing the mapping from

ATL to DSLTrans, the aim of this subsection is to explain the semantics of ATL and to instantiate them in our running example.

The execution of ATL transformations is split in two major steps: element creation and features initialization. This two-step process, however, is not explicit in ATL, and is described here to reduce the conceptual delta between ATL and DSLTrans.

The **first step** of the execution is the creation of target elements and the above-mentioned trace links, the latter being created implicitly and automatically by ATL. In Figure 3, this step creates all the elements in the output model (but does not set their references nor attributes) as well as the trace links specified in the central part of the figure. This means that *in-pattern elements* are obtained from the input model and *out-pattern elements* are created in the output model.

In the **second step**, the features of the elements created in the output model are set. This means that the bindings are executed and resolved. In order to initialize the references, ATL uses the internal trace links. Although the references are automatically resolved as explained in the previous section, there exists an operation, the so-called *resolveTemp* operation, which can be used to explicitly resolve references. It makes it possible to point to any of the target model elements generated from a given (sequence of) *in-pattern element(s)*. It is specially useful (and, in fact, needed) when the reference specified in a binding has to refer not to the first *out-pattern element* created in another rule, but to any of the rest.

In our running example, *B712* contains an operation of this type. Thus, an *Association* contains a reference named *committee* that has to reference the *Committee* created from the *City* matched in *IPE71*. As we can see in *R6*, every *City* creates a *TownHall* and a *Committee*. The latter is what we want to be referenced from *B712*. Since it is the second element created in the rule, we need the operation *thisModule.resolveTemp(ct,'cmt')*, where *ct* is the identifier of *IPE71* and *cmt* is the identifier of the *out-pattern element* that creates the *Committee* (*OPE62* in our example).

As we highlighted in the previous section, the *resolveTemp* operation provides the reason to store the identifiers of *in-pattern elements* and *out-pattern elements* in the trace links.

3.4 DSLTrans Representation

Figure 4 displays the DSLTrans transformation which corresponds to the ATL *Families-to-Persons_Extended* transformation shown in Listing 2. Let us mention here that we have removed five rules from the figure to improve visual clarity. There is a vertical dotted blue line for each of these rules, located where the rules have been removed. The missed rules are similar to those that surround them, and can therefore be safely ignored in our explanation.

The process of constructing a DSLTrans transformation from an ATL one is described in the next section. For now, note that DSLTrans transformation obtained from ATL through the higher-order transformation includes 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 attribute copies are represented with arrows from the *ApplyModel* of the rule to the *MatchModel*, such as in rule *Neighborhood2District*, where the created *District* gets the same name as the matched *Neighborhood*. The string of an attribute of a created element can also be initialized with the concatenation of several strings. For instance, in rule *Father2Man*, the full name of the created *Man* comes from the concatenation of the first name of the matched *Parent* and the last name of his/her *Family*. Or it can be assigned the string of an attribute of an element in the *MatchModel* concatenated with a given string, such as in rule *City2TownHall*.

4 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, emphasizing the translation of selected OCL operators. Finally, we explain the implementation of the mapping.

4.1 ATL Subset Selected

While the version of our translator from ATL to DSLTrans presented at the MoDELS 2015 conference [36] considered a large set of features available in the declarative part of the ATL language, our current version considers practically the complete set of features. As shown in Table 1, we now handle helpers and conditions. Since we consider almost all the features in the declarative part of ATL, we can assert that our current implementation of the higher-order transformation is sufficiently powerful to be of interest.

The only features that are not considered for the translation are the *using block* and *unique lazy rules*.

The using block is rarely used in ATL model transformations, as it is an optional mechanism for declaring

Fig. 4 DSLTrans version of the *Families-to-Persons_Extended* transformation

local constants in ATL rules. Consequently, the same rule can be written without using this block by always writing the constant content instead of the constant identifier. An example of an ATL transformation containing a using block and an equivalent one without it can be found on our website [3].

Regarding unique lazy rules, they would require special logic to be translated to DSLTrans. In an ATL transformation, the first time that a unique lazy rule is called, with a specific (set of) parameter(s), it creates one or more elements in the output model. The following times the rule is called with the same parameter(s), the elements that were already created are retrieved, but they are not created again. Translating this behavior would require the inclusion of a condition on the DSLTrans side. However, since there is no branching or any mechanism to specify conditions in DSLTrans, we do not translate unique lazy rules to DSLTrans. In any case, although unique lazy rules are a powerful feature of ATL, not many existing ATL transformations [2] use them.

Since DSLTrans is by default terminating and confluent, we require that the ATL transformation is also terminating and confluent before it can be translated to DSLTrans with our approach. This is easily achieved by following some guidelines and good practices when developing an ATL transformation.

First, since we are using the declarative part of ATL, helpers must not be used as global variables, so no information is to be stored in them. Second, we have to ensure that navigations terminate in a defined number of steps (by "step" we mean the traversal of a reference), so we must avoid recursive helpers and recursive lazy rules. Third, since we are dealing with out-place transformations, only ATL transformations written with the default execution mode (and not the so-called *refining* mode [46,47]) can be transformed. Please note that the majority of current ATL transformations are written using the default execution mode [2].

Finally, we require that the ATL transformation must not throw any compilation nor run-time error [20]. Compilation errors indicate, for instance, errors in the syntax, or that a target model element is being used as input for a rule. Indeed, the target model is not navigable, and is only writable [28]. If we want to access the target model in our ATL transformation, we have to make explicit use of the *resolveTemp* function, as shown in binding *B712* of Listing 2 and explained in Section 3.3. Run-time errors are thrown, for example, when the same source model element has been used as input element for two different matched rules.

In order to ensure all the mentioned points are satisfied, we can use the approach by Troya and Vallecillo [47]. Thus, the ATL transformation is translated to a formal domain, Maude [19], where termination and confluence checks can be easily performed.

Table 1 Features of Declarative ATL considered

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

Finally, let us mention that, in our current prototype, we have used ATL versions 3.5 (in Eclipse Luna) and 3.6 (in Eclipse Mars).

4.2 Mapping between ATL and DSLTrans

4.2.1 Semantics

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 binding step which is implicitly present in ATL for resolving associations between elements created in the transformation. Please recall that the semantics of ATL have been described in Section 3.3.

For clarification purposes, in the following discussion we explain the mapping generically and then we instantiate it for the *Families-to-Persons_Extended* case study shown in Listing 2, to describe how the partial DSLTrans representation shown in Figure 4 is built. Specifically, we explain how the transformation from ATL to DSLTrans works, i.e., we textually define its semantics. The semantics for the mapping are divided in two steps, to reflect the semantics of ATL,

Generic Semantics for Step 1 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. In DSLTrans, rules are given an explicit order, since a rule may match over elements that have been created in previous rules. In our case, DSLTrans rules corresponding to matched rules are independent from each other. Thus, their order does not matter, and we apply the same order as in the ATL transformation.

The *MatchModels* of these rules contain the elements appearing in the *from* part of the corresponding ATL rules. There is an element for each *in-pattern element* (IPE) that appears in the ATL rule. As well, if the ATL rule has a filter, then some more elements and associations may appear in the *MatchModel* in order to satisfy the conditions of the filter. Boxes representing

attributes can also appear inside elements of the *MatchModel*, which occurs when such attributes are used to initialize attributes in the *ApplyModel*.

In the *ApplyModels* of the rules created, there is an element for each *out-pattern element* (OPE) declared in the ATL rules. Normally, when more than one OPE is created in an ATL rule, then some OPEs reference others. In DSLTrans, this is specified as an association between the created elements in the *ApplyModel*.

When there are bindings in the OPEs of the ATL rules that are initializing attributes (not references), then these attributes appear in the elements created in the *ApplyModel* in DSLTrans. Recall that such bindings can be defined with helpers as well. Besides, as previously mentioned, attributes must also appear in the elements in the *MatchModel* if their value is used to initialize the values of the attributes in the apply part, and associations are created between them.

Finally, an attribute called *ApplyAttribute* appears within the elements created in the *ApplyModel* in two cases. First, whenever a value is assigned to any of the attributes of the element in the *ApplyModel*. Second, if the DSLTrans transformation is to be executed by the transformation engine, as they are required for optimization purposes.

These *ApplyAttribute* are, in fact, the way DSLTrans simulates the internal traceability mechanism employed by ATL explained in Section 3.3. Thereby, anytime an *ApplyAttribute* appears in a created element, the association of such element with the elements appearing in the *MatchModel* is stored in the traces. Since maintaining these traces in DSLTrans is expensive, traces are only created in the presence of an *ApplyAttribute*. This use of traces is necessary when employing *backward links*, as explained in the next step.

Please note that the presence of *ApplyAttributes* in the rules is not reflected in the output models generated. In fact, the "*ApplyAttribute*" name is a reserved keyword.

Running Example Instantiation for Step 1 This first step is exemplified in Figure 4 in the sequence of rules that goes from *Country2Community* until *Neighborhood2District*. There are six rules in such sequence, and two rules that have been omitted, which correspond to *Mother2Woman* and *Son2Man*. They are very similar to rules *Father2Man* and *Daughter2Woman*. Therefore, these eight rules have a direct mapping with rules R1 – R8 in Listing 2.

Let us have a look at the *Father2Man* rule. The ATL rule only has one *in-pattern element*, IPE21 in Listing 2, of type *Parent*, and it also contains a filter. In the corresponding DSLTrans rule, we can see that the *Parent* element is present, and there is also another element and some relationships. They correspond to the filter, as is explained in Section 4.2.2.

A number of rules such as *CityCompany2Association*, *Father2Man*, and *Daughter2Woman* have *MatchModel* elements which contain boxes representing element attributes. These attributes are used to initialized new attributes in the *ApplyModel*, as represented by incoming arrows.

Let us focus now on the *out-pattern*. The rule *City2TownHall* in cf. Listing 2 defines that the created *TownHall* (OPE61) has a reference to the created *Committee* (OPE62) through the association *committee*, as specified in B613. In its corresponding DSLTrans rule (*City2TownHall* in Figure 4), the association *committee* is created from element *TownHall* to element *Committee*.

Bindings that initialize attributes are present in our example as B211, B311, B411, B511, B611, B621, B711, B811, B911 and B1011. Note that the last two bindings are in lazy rules, and will be explained in the second step of the mapping. The initialization of such bindings can be seen in rules *Father2Man*, *Daughter2Woman*, *City2TownHall*, *CityCompany2Association* and *Neighborhood2District* in Figure 4. Note that, in this first step that bindings that initialize references are ignored. This would be all remaining bindings.

Finally, the *ApplyAttribute* attribute is present in all the rules except for *Country2Community*.

Generic Semantics for Step 2 A rule in DSLTrans is created for every binding that initializes the value of a reference in the ATL transformation. Such bindings can include helpers, whose content is considered in the translation as if it appeared directly in the binding. Again, 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. We give them the same order as the order of the bindings in the ATL transformation from which the DSLTrans rules are created.

In the DSLTrans rules created in this step, the left part of the ATL binding, which is the name of the association that is being resolved in the binding, appears in the *ApplyModel*. The source and target classes of the association are also placed in the *ApplyModel* of the rules. If the source and/or target classes of the association are abstract classes, our mapping determines the specific element that has to be added, as we see later in our running example. The elements and associations placed in the *MatchModel* of the created rule are those appearing in the right part of the binding. The right

part of bindings are expressed in terms of OCL expressions, and we explain them in Section 4.2.2.

All the rules created in this step contain backward links. As mentioned before, they allow matching over traces between elements in the input and the output models of the transformation. As explained in Section 2, when an element in the *ApplyModel* is linked with a backward link to one or more elements in the *MatchModel*, then the *ApplyModel* element is not produced in the output model, but is instead referring to elements that were already created in previous rules. This is how the internal traces mechanism of ATL is explicitly modeled in DSLTrans. Of course, an element in the *ApplyModel* can be linked to more than one element in the *MatchedModel* by backward links, and the other way around (several backward links can depart from the same element in the *MatchedModel*). This is equivalent to those traces in ATL that have more than one source/target elements.

According to the explanation given in the first step of the mapping, an *ApplyAttribute* attribute is also produced in the elements in the *ApplyModel* in this step.

Lazy rules can also be present in those ATL bindings that are initializing associations, so they are considered in this step. In this case, the elements appearing within the lazy rule are included in the DSLTrans rule created from the binding. Now, the elements created in the *MatchedModel* are not only those appearing in the right part of the binding, but also those acting as matching elements in the lazy rule. Likewise, the elements created in the *ApplyModel* also contain the elements and associations created in the lazy rule.

Running Example Instantiation for Step 2 The rules created in this step are all those that follow rule *Neighborhood2District* in Figure 4. Note that our transformation assigns these rules a unique but long name. For simplification purposes, this explanation will only use the first part of the name, until the first capital letter appears.

The two *copersons[...]* rules are the mappings to bindings B111 and B114, respectively. The two omitted rules correspond to bindings B112 and B113, and they are very similar to the two rules shown. In the DSLTrans rules created from these bindings, the left part of the ATL binding, which is the name of the association that is being resolved in the binding, appears in the *ApplyModel*. The source and target classes of such association are also placed in the *ApplyModel* of the rules.

Let us focus on the first *copersons[...]* rule. The metamodel in Figure 2(b) indicates that the source class of the *persons* relationship is *Community*, while the target class is *Person*. Regarding the source class, it is also the class that is created in the ATL transformation (OPE11 in our example), so it is straightforwardly included in the DSLTrans rule. As for the target class, the class *Person* is abstract. However we can know which non-abstract class inheriting from it should be chosen. This is resolved by navigating the OCL expression appearing in the right part of the binding. B111 retrieves elements of type *Parent* that have the role of *fathers*. Therefore, according to the *Father2Man* rule, we know the target element of the relationship must be of type *Man*.

Regarding the elements and associations appearing in the *MatchModel* of the created rule, they are those appearing in the right part of the binding. Since the right part of bindings are expressed in terms of OCL expressions, we explain them in Section 4.2.2.

In contrast to the two *copersons[...]* rules, the *tworkers[...]* rule contains an element of type *Person* in the *ApplyModel*. This rule is the mapping of B612. In this case, the target element of the *workers* relationship can be of type *Man* or *Woman*, and no distinction is needed. The only constraint is that the *Person* must have been created from a *Parent*, as indicated by its backward link.

To mention another example of backward links, examine again the two *copersons[...]* rules, where the element *Community* refers to the *Community* created in the *Country2Community* rule, since it is linked with a backward link with element *Country*. Likewise, elements *Man* and *Woman* were created in rules *Father2Man* and *Daughter2Woman*, respectively. This means that the only thing added in these two rules is the association *persons*, whenever the *MatchModel* is found in the input model.

In the *acommittee[...]* rule we see an example of an element in the *ApplyModel* with more than one backward link. It corresponds to binding B712, which initializes the *committee* association. In the DSLTrans rule, the *Association* element is linked backwards with the *Company* and *City* elements. Indeed, an *Association* is created from a *Company* and a *City* by rule *CityCompany2Association*. The *Committee* element is backwardly linked with *City*, since a *Committee* is created from a *City* in rule *City2TownHall*. Binding B712 is actually resolving the *committee* association by making use of the ATL *resolveTemp* operation explained in Section 3.2.

Finally, let us see an example of lazy rules called from a binding. In our running example, there are calls to lazy rules in B811 and B812. The DSLTrans rule *dfacilities[...]* corresponds to B811, while the DSLTrans

rule created from B812 is very similar and is therefore omitted in our explanation.

In Listing 2 we can see that B811 is invoking the lazy rule *CreateOrdinaryFacility* (R9), which creates an element of type *OrdinaryFacility*. It also initializes the *name* attribute of the created element and the association *members*. Therefore, all this is included in the *ApplyModel* of the rule created from B811.

Regarding the *members* association, let us clarify that the new element created by the lazy rule, OPE91, acts as its source. The target is resolved by the OCL expression *sch.students*, which returns an element of type *Child* (cf. metamodel in Figure 2(a)).

4.2.2 OCL Expressions Handled

One of the main challenges when analyzing an ATL transformation is to deal with the OCL expressions it contains, due to the large number of navigation possibilities that OCL offers. Although there are some works dealing with the translation of OCL to graph domains [8, 11], DSLTrans has its own peculiarities. For this reason, in this section we explain how OCL expressions are translated to DSLTrans using our running example, since several OCL operators are present. The most interesting ones are those that deal with collections.

First of all, let us recall that in ATL, OCL expressions may appear in both filters and bindings. As well, they always navigate the input model, as the output model is strictly writable in ATL. In the two steps of the mapping from ATL to DSLTrans explained in Section 4.2.1, filters are translated in the first step, while bindings are considered in the second step. However, in both cases, the elements and associations appearing in OCL expressions are included in the *MatchModel* of the generated DSLTrans rules.

Let us start by explaining navigations in filters, and then discuss those appearing in bindings, which may contain calls to lazy rules.

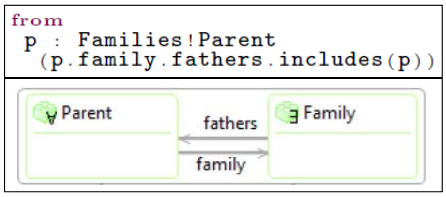
OCL Expressions in Filters Please recall that elements appearing in the *ApplyModel* of DSLTrans rules which do not have backward links are created from those elements in the *MatchModel* that are also not connected to backward links.

In the first step of the mapping (cf. Sect. 4.2.1), recall that there are no backward links in the rules created. Therefore, traces are created between the elements of the *MatchModel* and the *ApplyModel* whenever the *ApplyAttribute* is present. However, we do not want the elements in the *MatchModel* that are introduced due to the filter condition to be included in such traces. This is the reason why DSLTrans distinguishes in the *MatchModel* between *AnyMatchElements* and *ExistsMatchElements*.

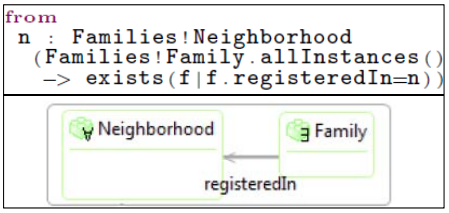
AnyMatchElements correspond to those *in-pattern elements* appearing in the ATL rules and must be included in the traces created. They have the ∀ symbol in their graphical representation. On the other hand, *ExistsMatchElements* are used to state conditions over *AnyMatchElements*, and we use them in our translation in order to insert elements that appear in the OCL expressions of the filters of the ATL transformation. They have the ∃ symbol in their graphical representation and are not considered in the traces.

The reason to decide whether to include an *ExistsMatchElement* or an *AnyMatchElement* in the *MatchModel* is simple, and is according to the content of the filter. If we have a navigation in the filter that reaches a certain element, but this element does not appear in the matching part of the ATL rule (*from* part), then it is represented by an *ExistsMatchElement*.

We can see two examples of our running case study in Figure 5. In the filter of Figure 5(a), we can see that *Parent* (variable *p*) is the element performing the match, and *Family* (reached through the *family* reference) is used as part of the condition. As for the filter of Figure 5(b), the element used as part of the condition is again a *Family*, in this case reached with the use of the *allInstances* and *exists* operations. As we see in both cases, all references appearing in the OCL expression of the filter are included.



(a) *Father2Man* rule



(b) *CityCompany2Association* rule

Fig. 5 Filter elements translated to *ExistsMatchElements*

Conversely, if the elements that are reached through navigations do appear in the matching part of the ATL rule, then *AnyMatchElements* are created in the DSLTrans rule. An example is shown in Figure 6.

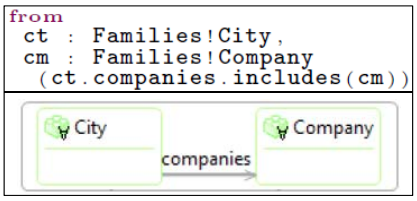
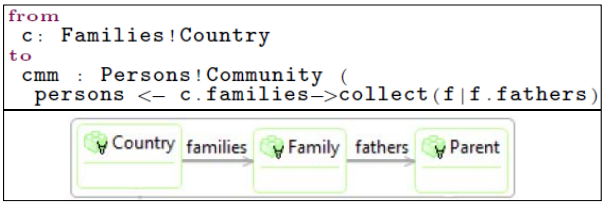


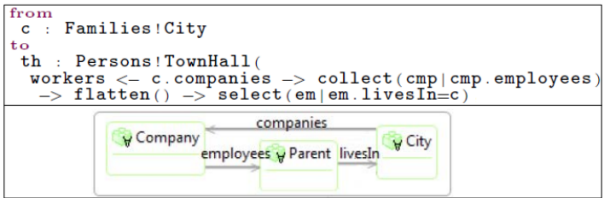
Fig. 6 Filter elements translated to *AnyMatchElements*: *City-Company2Association* rule

OCL Expressions in Bindings For rules generated in the second step of the mapping the right part of the bindings are included in the *MatchModel* of DSLTrans rules. In these rules, all *ApplyModel* elements are connected with backward links to elements in the *MatchModel*, so we do not need to include *ExistsMatchElements*.

Again, elements that are reached through navigations (whether OCL collection operations are used or not) must be introduced in the *MatchModel*. Figure 7 shows two examples where in the two bindings *select* and *collect* operators are present. All references that are included in both bindings are reflected in the *MatchModels* created from them, and all elements reached through said references are represented with *AnyMatchElements*.



(a) *copersons[...]* rule



(b) *tworkers[...]* rule

Fig. 7 Binding contents translated in *MatchModels*

OCL Expressions in Bindings That Call a Lazy Rule We have stated that, since elements have backward links in the *MatchModel* of those rules created from bindings, they will be created as *AnyMatchElement*. However, there is a special case, and this is when there is a call to a lazy rule in the binding, as we can see in Figure 8.

Lazy rules have one or more input parameters (an element of type *School* in our case). The DSLTrans rules created from bindings that contain calls to lazy rules produce new elements, which are precisely the elements created from the lazy rule. This is because the content of the lazy rule is considered when creating the DSLTrans rules, as explained in Section 4.2.1. For this reason, the DSLTrans rule may contain elements in its *MatchModel* that refer to properties of the element(s) passed as parameters to the lazy rule.

This is the case, in our example of Figure 8, with the element *Service* linked with the *ordinary* reference to the *School* element. Since no backward link is connected to this element, it is included as an *ExistsMatchElement*.

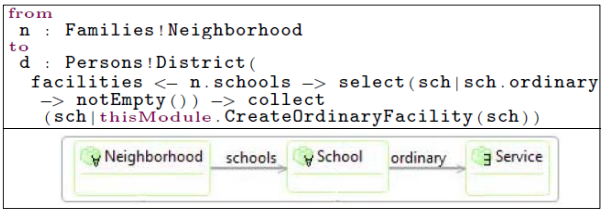


Fig. 8 Binding that contains a call to a lazy rule

4.3 Implementation

The mapping between ATL and DSLTrans has been implemented with a higher-order transformation (HOT) developed in ATL. It is the *ATL2DSLTrans HOT* shown in Figure 9, which is explained in the following together with its inputs and output.

The HOT is composed of three main matched rules for realizing the two-steps mapping described in Section 4.2.1. The first one matches a matched rule of the *ATL Transformation* taken as input and produces a rule in the *DSLTrans Transformation* generated as output. As we mentioned before, in this step we also create the corresponding attributes and filter conditions. In order to know if a binding in the *ATL Transformation* is initializing an attribute or a reference, we need information of the *Output Metamodel* taking part in the *ATL Transformation*. Several lazy and unique lazy rules for creating elements, associations and attributes are invoked from this matched rule. This rule also stores in an internal structure the traces that keep the relation between the elements of the *MatchModel* and *ApplyModel* of the *DSLTrans Transformation* rules that are generated, which is useful for generating backward links in the second step.

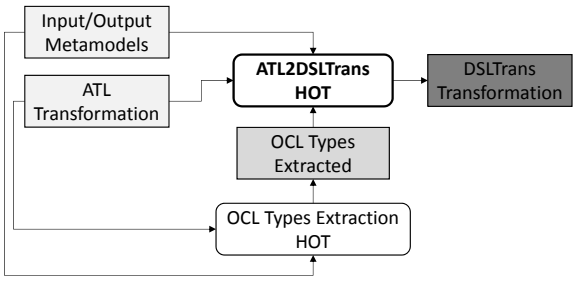


Fig. 9 Mapping Implementation

The second main matched rule deals with the creation of DSLTrans rules from ATL bindings initializing references that do not call any lazy rule, and the third one transforms those bindings that do invoke a lazy rule into DSLTrans rules. Thus, they take as input a binding where a navigation over the input model is realized and produce a rule in the *DSLTrans Transformation*. This is why we need the *Input Metamodel* of the transformation as input for the HOT.

These two main rules implement the second step of the mapping described in Section 4.2.1. Again, these rules call several helpers, lazy and unique lazy rules available in the *ATL2DSLTrans HOT* in order to create the elements appearing in the *DSLTrans Transformation* rules and the associations between them. The last step in these two rules is to create the backward links between elements in the *MatchModel* and the *ApplyModel*, for which the structure previously mentioned is used.

As another input for the *ATL2DSLTrans HOT*, we use a model where the OCL navigations appearing in the filters and the bindings of the input *ATL Transformation* are parsed (model *OCL Types Extracted* in Figure 9). In this parsing, we remove all the collection operators appearing in the navigations in order to obtain the types appearing in such navigations. Since ATL does not offer any support nor API to statically obtain the types of an OCL expression, we make use of another HOT, namely *OCL Types Extraction HOT*, that returns the model with the OCL navigations parsed.

This model is used by the *ATL2DSLTrans HOT* for two purposes. First, when creating the elements and associations in the *MatchModel* of a DSLTrans rule that correspond to the filter of an ATL rule, which is realized in the first step of the mapping. Second, for creating the elements and associations in the *MatchModel* of a DSLTrans rule that correspond to the navigation of a binding, which is realized in the second step of the mapping.

The rationale for having two separated HOTs is twofold. First, the result of the *OCL Types Extraction HOT* can be used with a different purpose and, second, we reduce the complexity of the *ATL2DSLTrans* HOT.

```
Input: Input_MM, Output_MM, ATL_Trans, OCL_Parsed
Output: DSLTrans_Transformation
for all MR ∈ MatchedRule do        // Beginning of Step 1
  Create DSLTrans rule
  for all IPE ∈ MR do
    Create AnyMatchElement
  end for
  if MR contains Filter then
    [Create ExistsMatchElements]   // May happen or not
    Create MatchAssociations
  end if
  for all OPE ∈ MR do
    Create ApplyClass
    for all B ∈ OPE do
      if B initializes an attribute then
        Create ApplyAttribute in ApplyClass
        if Attributes from an IPE are used in B then
          Create MatchAttribute in MatchElement
        end if
      end if
    end for
  end for
  Create corresponding ApplyAssociations
end for
for all MR ∈ MatchedRule do        // Beginning of Step 2
  for all OPE ∈ MR do
    for all B ∈ OPE do
      if B initializes a reference then
        Create DSLTrans rule
        Create AnyMatchElements with the OCL_Exp of B
        Create MatchAssociations
        Create ApplyElements, for OPE and the type of B
        if B invokes a LazyRule (LR) then
          for all OPE in LR do
            Create ApplyElement
          end for
        end if
        Create corresponding ApplyAssociations
      end if
      Create corresponding BackwardLinks
    end for
  end for
end for
```

Fig. 10 ATL to DSLTrans HOT summary

As a summary, the pseudocode presented in Figure 10 describes, at a high level, the two-steps algorithm that maps ATL transformations to DSLTrans transformations. Using the same notation as before in the paper (cf. Section 3.2), *MR* stands for *matched rule*, *LR* for *lazy rule*, *IPE* for *in-pattern element*, *OPE* for *out-pattern element* and *B* for *binding*.

5 Contract Prover

This section will describe the operation of our contract prover such that contracts are proven on all executions of a DSLTrans transformation.

The contract prover we describe here is the engine of the SyVOLT tool, which can currently be used to develop and verify DSLTrans transformations within the Eclipse environment [1,5,33]. Examples of contracts we prove are also presented, along with a brief discussion of the expressibility of the contract language.

5.1 Contract Proving Overview

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

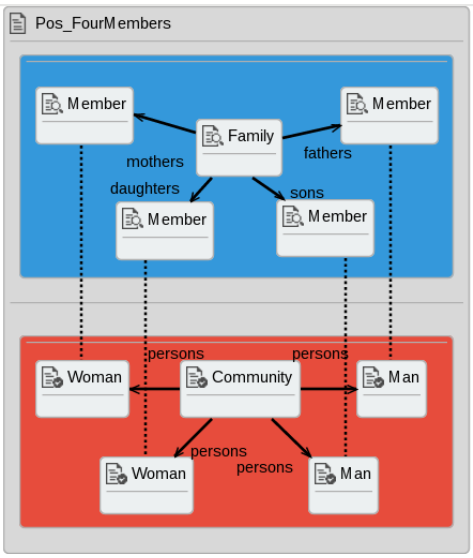


Fig. 11 A contract to verify that two *Woman* and two *Man* elements are produced from the corresponding *Members*

For example, Figure 11 describes a contract to be proved over all transformation executions for the extended *Families-to-Persons* transformation. An informal statement for this contract is: *'an input family with a father, mother, son and daughter should always produce two men and two women in the output community'*. Note that we employ backward links as part of the contract language, where as they are used to require that the output elements be generated from the attached input elements, similar to their use in DSLTrans rules. 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. Further examples of contracts are found in Section 5.3.1.

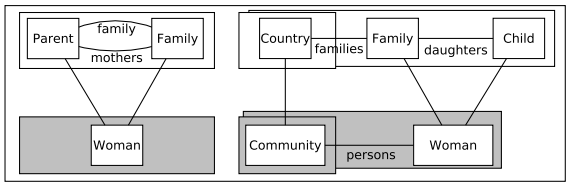


Fig. 12 An example path condition representing the execution of three rules

Contracts are proved through a process that first symbolically constructs all possible executions of the transformation, producing a set of *path conditions*. Each path condition represents the execution of a set of transformation rules, by containing the input and output elements which are produced by the execution of those transformation rules.

For example, the path condition in Figure 12 represents the execution of three rules in the transformation. This representation includes the input and output elements that will be present in the input and output models if these three rules execute. The set of path conditions produced by the prover will therefore partition the set of valid executions of the transformation, where each execution is an input/output model pair. This technique was first proposed in [31] and further detailed in [32].

Pre-/post-condition contracts form an implication, which needs to be checked for each path condition that has been generated by the proving algorithm. In broad terms, a contract holds on a path condition if either the contract's pre-condition elements cannot be 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 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 [32], while extensive discussion of the contract language is found in the PhD thesis of Gehan Selim [42]. Section 5.3.1 and Section 5.4 present further contract examples, while Section 5.5 briefly discusses the expressiveness of the contract language.

5.2 Path Condition Creation

As described in [33], our contract prover constructs all artifacts used for contract proof through matching and rewriting of typed graphs. Therefore, the first

step for the contract proving process is to create T-Core matcher and rewriter primitives from each of the rules in the DSLTrans transformation [45]. These model transformation primitives are at the core of our prover, allowing us to reason about how rules could overlap with each other during transformation execution, and to perform the graph rewriting necessary for our technique.

Note that this use of reasoning about the transformation under study as explicit graphs is in opposition to other approaches in the literature, where the transformation specifications are translated into a SAT solver or theorem prover, and then the proving mechanisms for those tools are used. A further discussion of our approach versus that in the literature can be found in [42].

In order to fully reason about all input models to a transformation, our contract prover creates a set of artifacts that represent all possible executions of the transformation. These artifacts are created by symbolically executing all rules in the transformation, taking into account rule overlapping and dependencies between rules. The rule combinations that are created are termed *path conditions*.

For example, the first path condition could represent the case where no rules in the transformation execute. The next path condition is the case where only the first rule executes, the next is where only the second rule executes, and a fourth path condition is where both the two rules execute.

Note that in our path condition creation process, we only consider one execution of each rule. That is, either a rule does not execute (and does not appear in the path condition), or we assume that it executes some number of times (and the rule appears once). This restriction is due to our abstraction, where we symbolically represent many executions of the same rule by the rule being present only once in each path condition. This abstraction is necessary for analysis purposes, as the infinite number of transformation executions must be covered by a finite number of path conditions. Note that this abstraction is possible because of the monotonicity of a DSLTrans transformation: a rule can only add elements to the output model of a DSLTrans transformation, but never remove them.

As the transformation is made of layers, the symbolic execution process moves through each layer and determines how rules may interact with each other. Unlike generating the powerset of all rules, these rule interactions may in fact decrease the number of path conditions generated by the prover as certain combinations of rules are proven infeasible.

For example, consider a rule R1 which matches on an A element, and a rule R2 which matches on an A element connected to a B element. During an execution of the transformation, it would be impossible for R2 to execute without R1 also executing, as the match graph of R1 is a subset of the match graph of R2. Therefore, the rule R1 is 'subsumed' by the rule R2. Our prover is able to detect these subsumption interactions and resolve them in a step just prior to path condition generation. This lowers the number of path conditions created by disallowing certain rule combinations, as further discussed in [44].

As well, DSLTrans rules can also define *backward links*, as described for the extended *Families-to-Persons* transformation in Section 4.2.1. Recall that these backward links make dependencies on elements created by earlier rules. Specifically, these backward links require that the connected element in the apply part of the rule was created from the connected element in the match part of the rule, by matching over traceability links created during the execution of the transformation. This functionality is therefore similar to the implicit binding step present in ATL, as discussed in Section 4.2.1 under the title *Generic Semantics for Step 2*. During path condition construction, these backward link dependencies prevent some rules from executing, further decreasing the rule combinations possible.

5.2.1 Rule Interaction Cases

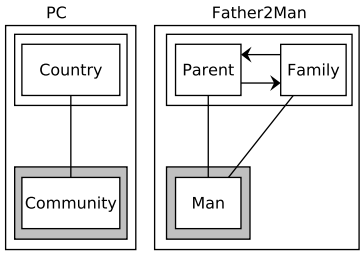
As mentioned, our contract prover combines rules from different layers in the transformation to generate the path conditions. This section will now summarize the three cases in which a rule in a layer may combine with a path condition from a preceding layer. This information is presented to give the reader a sense of the complexities behind the symbolic execution of these rules, and a greater understanding of how path conditions represent executions of the transformation. For interested readers, a formal treatment of these cases is found in [32].

Note that backward links are represented by thick dashed lines between the match and apply elements in the figures below. Traceability links have also been added between elements in rules, and are represented by thin unbroken lines. For clarity, we omit association labels in the figures.

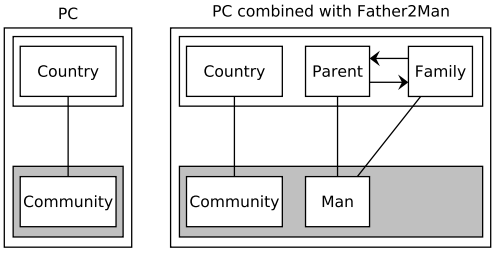
Empty Path Condition The path condition generation process begins with the empty path condition. As mentioned, this represents the set of all transformation executions where no rules execute. As rules are combined with this empty path condition, match and apply elements will be placed in the path condition. These elements will symbolically represent elements present in the input and output model of the transformations represented by that path condition.

No Dependencies In the first case for rule interaction, the rule contains no *backward links*. In the path condition generation algorithm, two different path conditions will be produced. The first path condition produced represents the possibility of the rule not executing, while the other path condition produced represents the possibility of the rule executing.

An example of this case is displayed in Figure 13, where the path condition *PC* is combined with the rule *Father2Man*. Note that the example path condition *PC* already contains *Country* and *Community* elements, to represent the symbolic execution of the rule *Country2Community*. The two path conditions on the right-hand side of Figure 13 show one path condition which is identical to *PC*, and one path condition which also contains the elements from the rule *Father2Man*.



(a) Path condition and rule to combine

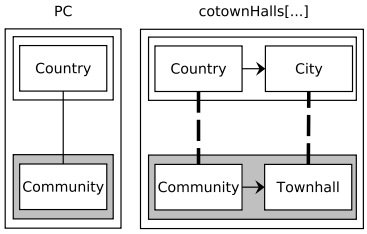


(b) Two path conditions produced by combination

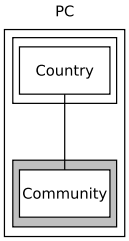
Fig. 13 Combination example where the rule has no dependencies

Unsatisfied Dependencies For the second case in rule interaction, the latter rule contains backward links that cannot be found in the path condition. This implies that the rule cannot execute. The old path condition is retained, and no new path condition is created.

This case is represented by Figure 14. Note that the rule *cotownHalls[...]* contains backward links, which require that a *Community* element to have been created by a *Country* element, as well as a *Townhall* element to have been created by a *City* element. As this second backward link cannot be satisfied by the elements in *PC*, the rule *cotownHalls[...]* cannot execute. Therefore, only the path condition *PC* is retained and no new path condition is created.



(a) Path condition and rule to combine



(b) One path condition produced by combination

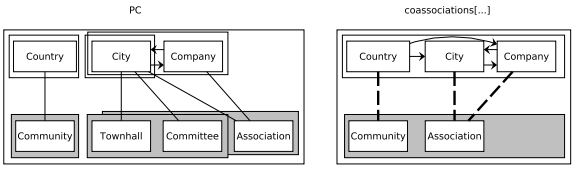
Fig. 14 Combination example where the rule's dependencies are not satisfied

Satisfied dependencies Finally, the most difficult case is where the backward links match onto the path condition, over the traceability links present. Therefore, the rule may or must execute, depending on whether the elements in the match part of the rule already exist in the path condition. In this case, a new path condition is created for every possibility of how the rule may be matched onto the path condition.

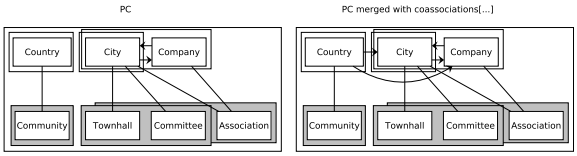
In the partial satisfiability case, not all elements of the rule can be found in *PC*. Figure 15 shows the combination of the path condition *PC* with the rule *coassociations[...]*. Note that the rule contains associations between the *Country* and *City* elements, as well as between the *Country* and *Company* elements. However, these associations are not present in the path condition *PC*.

As the associations are not present in *PC*, this indicates that there is the possibility that the input model may not contain these associations between these elements. Therefore, two path conditions are produced. One path condition represents the case where the input

model does not allow for the rule *coassociations[...]* to execute. The other path condition produced will include these associations, as it represents the case where the rule will execute on the input model. Note that if the rule might match at multiple locations on *PC*, a new path condition would be produced for each location.



(a) Path condition and rule to combine



(b) Two path conditions produced by combination

Fig. 15 Combination example where the rule's dependencies are partially satisfied

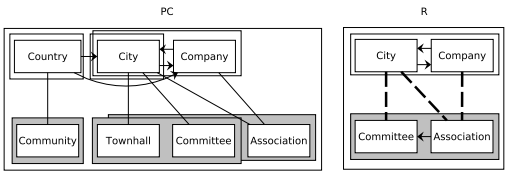
The complement to the partial satisfaction case is the total satisfaction case. In this case, the rule's required elements are all present in the path condition. Therefore, the rule must execute for the transformation executions represented by that path condition.

Figure 16 demonstrates the case where the dependencies of the rule *acommittee[...]* are totally satisfied by the path condition *PC*. Note that all backward links in the rule can be found in *PC*, as well as the required associations.

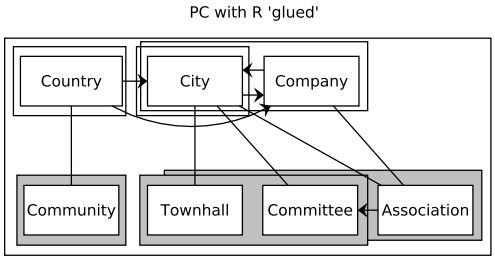
The path condition produced is built by combining the rule onto the path condition locations at the location(s) where the rule matches. Note that in Figure 16, an association has been built between the *Association* and *Committee* elements. As with the partial satisfaction case, the rule may match at multiple locations.

Attribute Setting The setting of attributes for rule elements is also symbolically executed in path condition construction. Essentially, we store in the path condition the equations stating the values for the attributes as they are assumed by the rules, and in subsequent rules we check for value compatibility of the match elements being matched. If the conditions on the attributes on the path condition element and the rule element are conflicting, no path condition is generated.

Note that a fairly trivial solver is currently used, as the only available attribute data type in DSLTrans is



(a) Path condition and rule to combine



(b) One path condition produced by combination

Fig. 16 Combination example where the rule's dependencies are totally satisfied

String. However, our approach is not overly restricted by this approach. From a pragmatic viewpoint, we note that our industrial case study only manipulates *Strings*.

More generally, DSLTrans has been used to write many useful transformations with models that have only *Strings* as attributes. This is because DSLTrans specializes in language translations, as described in [34]. Model transformations of this kind typically do not require complex computations over attributes and the bulk of the work is achieved by node matching and rewriting. *String* attribute data is mostly copied over to the generated model or concatenated with other *String* data.

A possible workaround to the restriction of only having available the *String* type is to convert non-*String* attributes and operations into *Strings*, such that they can be manipulated by the transformation. Then, each output *String* can be evaluated to produce the value in the original type. Note that, using this method, no operations of algebras other than the *String* algebra can be executed by the transformation engine.

A more holistic approach would be to introduce additional data types in the DSLTrans language itself, thus allowing the execution of operations of other algebras. However, much care needs to be used when introducing new data types in DSLTrans, such that those types do not introduce potential non-terminating computations. That would invalidate the fact that all DSLTrans transformations terminate. Additionally, a more powerful solver would also be necessary to allow path condition construction for a DSLTrans extended with additional data types.

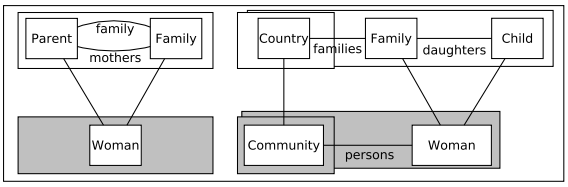


Fig. 17 An example path condition representing the execution of three rules

Partitioning Transformation Executions Following this reasoning about the three cases for interactions of rules, all rules in the transformation are examined and combined into path conditions. This is performed by examining each layer in the DSLTrans transformation in turn.

For the first layer, the empty path condition is combined with each rule. Then for each subsequent layer *n*, the set of path conditions produced by layer *n-1* are combined with rules from layer *n*. For example, the path condition examined in the total satisfiability case above will have been produced by the layer containing *coassociations[...]*, as in the partial satisfiability case.

The resulting path conditions at the end of this process will represent all viable sets of rules that could execute in the transformation. The infinite set of transformation executions will therefore be partitioned by the finite set of path conditions [32]. As each rule contains match and apply elements, the path condition thereby define which elements are present in the input and output models for that partition of transformation executions. Note that the empty path condition, which contains no elements, matches all other transformation executions not represented by another path condition.

For example, Figure 17 symbolically represents the execution of three rules from the extended *Families-to-Persons* transformation: *Country2Community*, *Mother2Woman*, and *copersonsSolveRef[...]Woman*. Note that the *Community* element was produced from the *Country* element in the *Country2Community* rule, and was matched over by the backward link dependency in the *copersonsSolveRef[...]Woman* rule. This path condition represents all transformation executions where these three rules execute, and thus presents the input and output elements and associations which are known to exist if this set of rules executes.

This abstraction of transformation executions by path conditions forms the basis for our technique of contract proving, where proving contracts on the set of produced path conditions allows us to reason about how the contract holds on the transformation's input-output model pairs.

5.3 Contract Proving Process

As path conditions are constructed through reasoning about the interaction of transformation rules, the structure of a path condition is very similar to that of a DSLTrans rule with a match graph and an apply graph, as seen in Figure 17.

The meaning of a particular path condition is, 'if the elements in the top component of the path condition are in the input model, then the elements in the bottom component will be in the output model.' Recall that this matches the intended meaning of a contract: 'if the elements in the pre-condition are found in the transformation's input model, then the post-condition elements should be found in the output model'. Therefore, to prove that a contract holds or not on a path condition, it is sufficient to see whether the elements in the contract can be matched onto the path condition, as described in [32].

There are three cases for determining the status of a contract:

- If the pre-condition of the contract, including backward links, does not match the path condition, then the contract is not applicable for that path condition.
- If both the pre-condition and post-condition match, then the contract does hold on that path condition.
- If the pre-condition matches, but the post-condition does not match, then the contract does not hold on that path condition.

Note that a contract may be expected to not hold in all cases for a transformation. For example, consider the contract *DaughterMother*, reproduced in Figure 18. The informal statement for this contract is 'a family with a mother and a daughter will always produce a community with a man'. It is easy to see that an input model which contains only mother and daughter elements should not produce a man in the target community.

Our contract prover will then find multiple counterexample path conditions which cause the contract to not hold, such as the path condition in Figure 17. Note that the pre-condition of the contract does match onto the top component of the path condition, while the *Man* element in the contract post-condition cannot be found in the bottom component of the path condition. Thus the failure of this contract gives further assurance that the transformation is working correctly, as *daughters* and *mothers* are not accidentally transformed into *men*. As this result is expected, this allows the transformation builder to have increased confidence in the validity of the transformation.

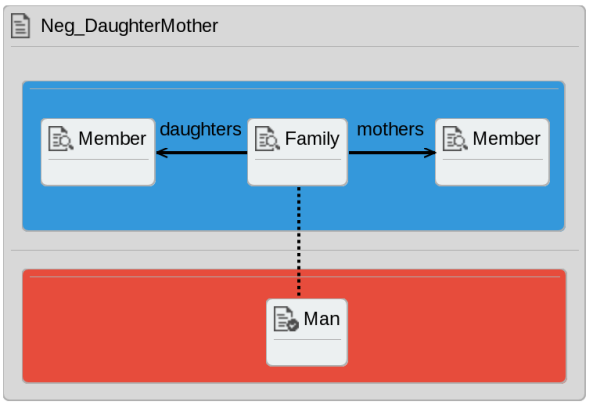


Fig. 18 A contract to verify whether a *Man* element will be produced from a *Family* containing a *daughter* element and a *mother* element - this contract will not hold

If a contract fails to hold on the transformation and it was not expected to fail, then this indicates an error with either the contract or the transformation. The prover will report (and optionally draw) the path conditions which the contract did not hold on. As well, a minimal path condition is reported, which represents the smallest combination of rules that fails. This allows the transformation developer to identify those rule combinations wherein an error may occur, and change the transformation or contract accordingly. Note that our technique only identifies the path conditions where a problem arises. Identification of the erroneous elements and suggestions for repair are not handled at this time.

Note that, despite the fact that the pre-condition of the *DaughterMother* contract cannot be (isomorphically) matched in the path condition in Figure 17, the pre-condition of the contract is still found in that path condition. This is so because there are input models matched by these two rules where the two distinct *Family* elements belonging to the two separate rules will match over the same family instance – remember that in DSLTrans rules can match over the same elements in the input model. The relation between the pre-condition of contracts and path condition is thus such that, any possibility of overlaps (or absence thereof) between elements of the same type belonging to two or more different rules in the path condition, is considered as a matching possibility for the pre-condition of the contract.

The relation between the contract and path condition packs thus more information than a simple graph isomorphism. It is a mixed partial surjective / injective homomorphism between the path condition and the contract typed graphs: while the surjection allows "forgetting" that two or more elements in a path condition belong to different rules, the injection guarantees that elements belonging to the same rule in the path condition have an isomorphic counterpart in the property. Further examples, as well as a formalization of this relation can be found in [32].

In the case that the contract must explicitly reason about the multiplicity of *Family* elements, the contract language includes propositional logic, as in Section 5.3.1.

5.3.1 Further Contract Examples

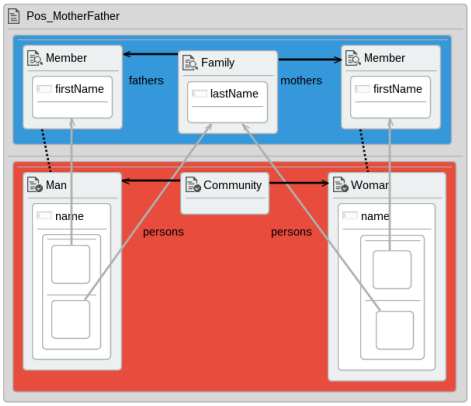


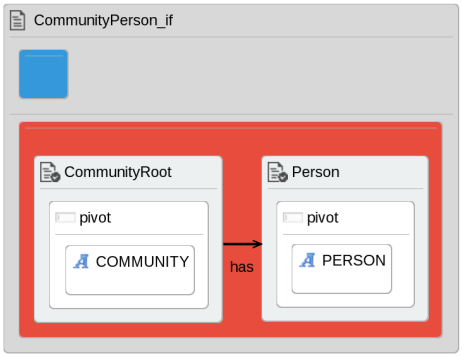
Fig. 19 A contract to verify proper construction of the *name* attribute in the output model

Our contract language also allows reasoning about the attributes of elements in the models. Figure 19 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*.

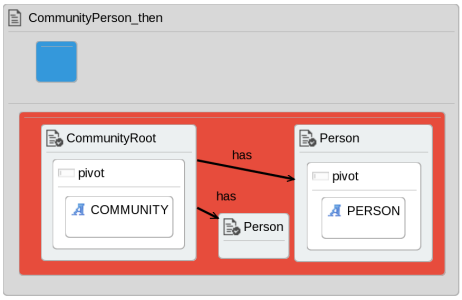
Contracts can also be combined using propositional logic and pivots to enhance the expressiveness of the contract language [42, 44]. For example, we present a contract for the original *Families-to-Persons* transformation in Figure 20. This contract demonstrates the use of propositional logic in our contract prover to form an 'if, then NOT' implication between the two contracts[2]. For this combination contract to be true, for each path condition where the first contract holds, the second contract must not hold and the elements marked by the *pivot* attributes must be the same. This contract's informal statement is 'If a community contains any people, then that community does not contain two (or more) people'. Therefore, this contract expresses constraints on the multiplicity of elements, as will be further discussed in Section 5.5.

Note that the contract language we present in this paper relies only on constructs that are found in the

[2] The logical connections between the contracts is not represented graphically



(a) 'If' portion



(b) 'Then NOT' portion

Fig. 20 Using propositional logic to express 'if, then NOT' contracts

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.

5.4 Contract Results

This section will present the nine contracts we have created for the extended *Families-to-Persons* transformation, to illustrate the utility of contracts for verification.

366 path conditions were created for this transformation, representing a finite partition of the infinite set of the transformation's possible executions. Then, each of the nine contracts were tested against each path condition. Each contract is detailed here as a tuple of an informal statement, the graphical representation, and how many path conditions the contract succeeded or failed on.

Note that a contract's success here means that both the contract's pre-condition and post-condition (including backward links) matched onto the path condition, while failure means that the pre-condition matched, but the post-condition did not. As well, the contract still holds on path conditions where the pre-condition did not match, which is the remainder of the path conditions.

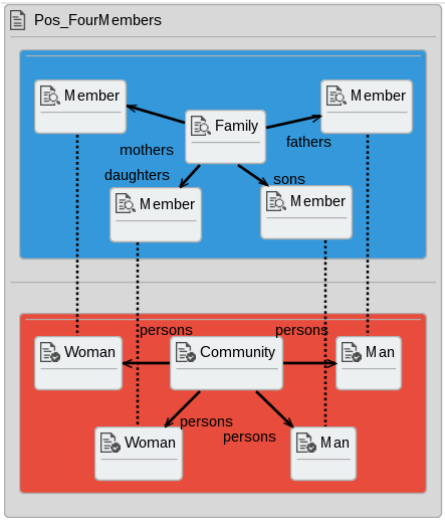
If the contract does not hold on some path conditions, and therefore on not all transformation executions, a brief explanation will describe the rule interactions that prevent the contract from holding.

Note that this section includes contracts that we expect to fail for the transformation. As mentioned, these contracts increase our confidence in the correctness of our transformation, as the prover will produce counter-example path conditions where the contract does not hold. As the path condition represents the execution of a particular set of transformation rules, this allows the user to reason about the interaction between the rules, and determine whether the transformation is erroneous or not.

Note that the following division of contracts into sections is primarily for readability, as they address different areas of the source and target metamodels.

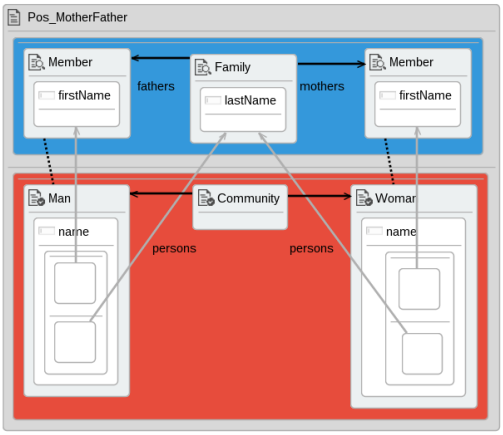
5.4.1 Families-to-Persons Contracts

Pos-FourMembers
Statement: A *Family* with a *father*, *mother*, *son* and *daughter* should always produce two *Man* and two *Woman* elements in the target *Community*.
Expected Result: Holds for all path conditions
Path Conditions Succeeded On: 137
Path Conditions Failed On: 0

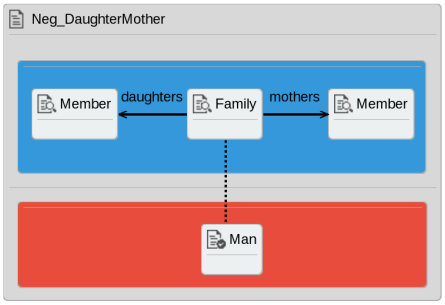


Pos-MotherFather
Statement: The full name of a produced *Person* is correctly created from the concatenation of the first name of the *Member* and the last name of his/her *Family*

Expected Result: Holds for all path conditions
Path Conditions Succeeded On: 236
Path Conditions Failed On: 0

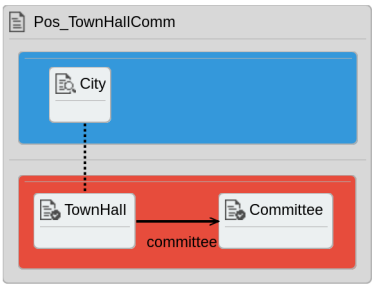


Neg-DaughterMother
Statement: A *Family* with a mother and a daughter will always produce a *Community* with a man
Expected Result: Does not hold for all path conditions
Path Conditions Succeeded On: 178
Path Conditions Failed On: 42
Explanation: A *Man* element will not be produced from an all-female *Family*

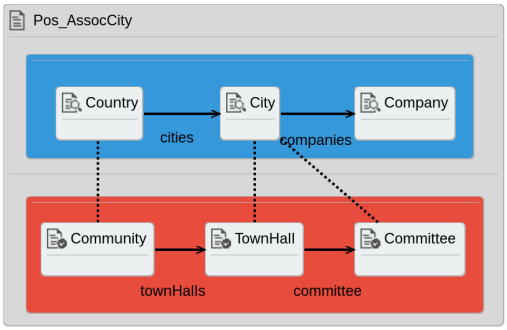


5.4.2 Location Contracts

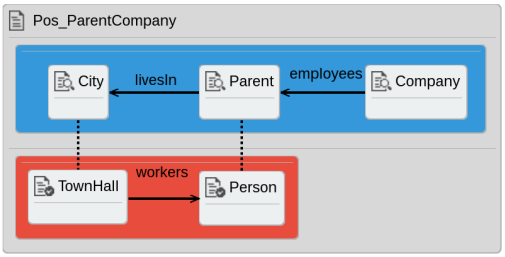
Pos-TownHallComm
Statement: A *TownHall* and a *Committee* are created for every *City*, and the created *TownHall* must have the created *Committee* as its *committee*
Expected Result: Holds for all path conditions
Path Conditions Succeeded On: 352
Path Conditions Failed On: 0



Pos-AssocCity
Statement: A *Community* that contains a *City* with a *Company* should produce a *Community* with a *TownHall* and a *Committee*
Expected Result: Holds for all path conditions
Path Conditions Succeeded On: 287
Path Conditions Failed On: 0

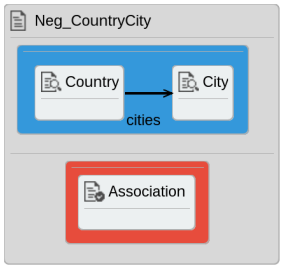


Pos-ParentCompany
Statement: If a *Parent worksIn* a *Company*, the *Person* created from him/her should be within the *workers* of the *TownHall* created from the *City* where the *Person* lives
Expected Result: Holds for all path conditions
Path Conditions Succeeded On: 222
Path Conditions Failed On: 0



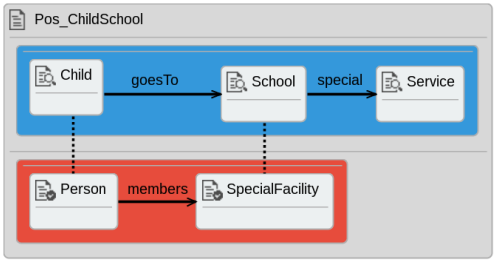
Neg-CountryCity
Statement: If the *Country* has at least one *City*, then at least one *Association* should be created
Expected Result: Does not hold for all path conditions
Path Conditions Succeeded On: 189
Path Conditions Failed On: 176
Explanation: An *Association* is only created if there is

a *Company* in the *City*

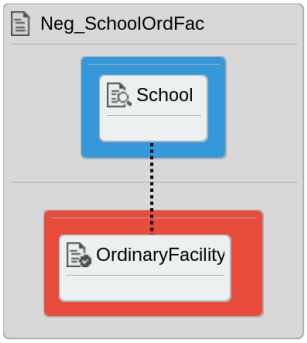


5.4.3 Facility Contracts

Pos-ChildSchool
Statement: If a *Child goesTo* a *School* that has a special *Service*, then a *SpecialFacility* has to be created that has the *Person* created from the *Child* as *members*
Expected Result: Holds for all path conditions
Path Conditions Succeeded On: 168
Path Conditions Failed On: 0



Neg-SchoolOrdFac
Statement: An *OrdinaryFacility* should be created from each *School*
Expected Result: Does not hold for all path conditions
Path Conditions Succeeded On: 168
Path Conditions Failed On: 60
Explanation: A *School* will be transformed into a *SpecialFacility* if it provides a special *Service*



5.5 Contract Expressiveness

This section will briefly discuss the expressiveness of our contract language.

As seen from the above examples, contracts contain a pre-condition and a post-condition which each contain a typed graph. [42] divides the possible contracts into three types: *multiplicity invariants*, *syntactic invariants*, and *pattern contracts*[3].

5.5.1 Multiplicity Invariants

Contracts can express *multiplicity invariants* contained in the source or target metamodel. As seen in Figure 20 in Section 5.3.1, the propositional logic of our contract language allow us to specify that only one *Person* should be connected to a *Community* in the output model. This allows the user to ensure that rules are not combining to produce more elements than desired.

Note however, that due to the abstraction of our approach these multiplicity invariants are not strict. As only one execution of each rule is considered in a path condition, our prover will not generate path conditions where a rule executes multiple times. Thus, multiplicity contracts such as the one in Figure 20 check for the existence of transformation executions where two *Person* elements are *always* created.

5.5.2 Syntactic Invariants

Syntactic invariant contracts check whether the path condition is well-formed with respect to the input or output syntax. An example of this type of contract is shown in Figure 21 for the UML-to-Kiltera case study described in Section 6.1.3. The informal statement for this contract is 'if there is an *Inst* element, then that *Inst* element has the same name as a *ProcDef* element.'

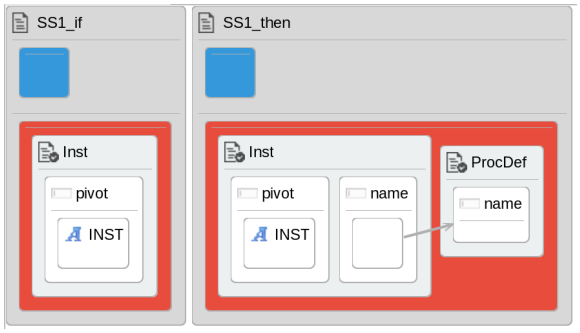


Fig. 21 An example of a syntactic invariant contract

[3] We here exclude *rule reachability* contracts, as the prover now automatically reports the failure of a rule to execute

5.5.3 Pattern Contracts

The final category of contracts described by [42] are *pattern contracts*, which relate elements in the input model to elements in the output model. The nine contracts presented in Section 5.4 for the extended *Families-to--Persons* transformation are of this type. The intention of these contracts is to allow the user to verify that multiple rules are interacting in a valid way, which is difficult from manual inspection of the rules themselves.

5.5.4 Limitations

We note that our contract language allows for the definition of a wide variety of structural conditions. Comparisons can be drawn to [27], where the authors use *PaMoMo* as the basis for their contract language. This allows the authors to define a variety of visual contracts to express both negative and positive contracts.

However, our contract language is currently limited as it can only represent structural conditions and not arbitrary expressions. This renders the contract language much less powerful than other constraint languages such as OCL. For example, the current implementation of our contract language does not contain operators for sets, or for handling non-String attributes. As well, as we are abstractly representing the rules that execute, and the number of times each rule executes, users must be careful about their definition of multiplicity constructs. Contracts also cannot be written to validate instance data for input or output models, such as ensuring that all input names start with a capital letter.

6 Experimental Results

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 complex 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 through transformation slicing?
- RQ4: Does the version of the transformation produced by our higher-order transformation differ significantly from a hand-built transformation?

Note that RQ1 and RQ2 are discussed directly in this section, while RQ3 is examined in Section 7 and RQ4 in Section 8.

6.1 Study Setup

This section will describe the case studies used to answer our research questions.

6.1.1 Families-to-Person Transformations

One of our experiments for this work was the *Families-to-Persons* transformation described in [36]. We retain this transformation for our experiments, as it contains 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 rules in the transformation contain duplicate elements, such as the *Family* element. As described in Section 5.3, our contract prover must be able to correctly perform non-isomorphic matching of the contract onto a path condition. That is, if there are similar elements in two rules in the path condition, the contract prover must resolve whether these elements match over separate elements in the input model, or over the same element.

We also performed experiments on the extended *Families-to-Persons* transformation described in Section 3. This extension expands the number of ATL rules from five in the original transformation to ten in the extended version, which increases the number of DSLTrans rules produced from 9 to 19. Consequently, the number of path conditions produced by our prover grows from 101 to 366.

The purpose of experimenting on this transformation is therefore to examine the performance of our property prover on a larger transformation example which contains more complex rules and contracts. Note that the contracts proved on this transformation have been created by an author of this paper and do not come from past work.

6.1.2 GM-to-AUTOSAR Transformation

Another transformation we examine as a case study is an industrial transformation seen in our earlier contract proving work [44]. The transformation in question takes as input models defined in a proprietary legacy metamodel used at General Motors (GM) for Vehicle Control Software development. The output metamodel is

Table 2 Size of transformation metamodels

Transformation	Metamodel	Num. Elements	Num. Assoc.	Num. Attrib.	Num. Inheri. Relations
Families-to-Person	Input	3	5	2	0
	Output	3	1	2	2
Ext. Families-to-Person	Input	11	21	3	7
	Output	12	8	2	9
GM-to-AUTOSAR	Input	6	10	5	0
	Output	13	8	2	3
UML-to-Kiltera	Input	42	51	6	41
	Output	30	41	9	20

the automotive industry-standard AUTOSAR[4]. Therefore, this transformation is used for model-evolution purposes, migrating the models to the new standard for greater interoperability with tools.

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 DSLTrans transformation produced by our higher-order transformation, and the hand-built transformation built for that earlier work. These results will be discussed in the context of RQ4 for whether the DSLTrans representation generated by the higher-order transformation is sufficiently efficient for contract verification to replace the hand-built version. This will be discussed in Section 8.1, along with a brief comparison of the two DSLTrans transformations.

Note that [42] further discusses the GM-to-AUTOSAR case study, including a detailed description of the transformation and the contracts to be proved.

6.1.3 UML-to-Kiltera Transformation

For our final case study, we have selected a transformation for transforming a subset of UML-RT state machine diagrams into Kiltera, which is a 'language for timed, event-driven, mobile and distributed simulation'. The transformation was proposed in [39] and developed in [37]. Previously, we have studied this transformation to obtain insights into the contract-proving process [43].

We include this case study for reasons similar to the GM-to-AUTOSAR transformation. As the transformation rules in the UML-to-Kiltera contain a large number of elements, especially duplicated elements, we are interested in the performance penalty to the matching and rewriting steps during the contract proving process. One contract in particular is quite troublesome for our matching process, as described in Section 7.2.

As well, we are interested in the differences between the hand-built transformation built for [43], and the version produced by our higher-order transformation (HOT). In particular, a number of improvements were made to the higher-order transformation to correctly generate the correct DSLTrans transformation for this case study, increasing the applicability of our approach in verifying ATL transformations. Further details on the comparison between the hand-built and HOT-produced transformation versions are in Section 8.2.

Further details on this case study are presented in [42], including both metamodels, all contracts, and both ATL and DSLTrans transformations. Note that a number of contracts have been omitted from the current work due to some functional equivalences.

6.2 Case Study Summary

This section briefly presents two tables which compare the case studies in terms of their ATL rule composition, as well as certain metrics for their input and output metamodels. This summary is presented to support our claim that our technique is applicable to a variety of ATL transformations. Please note that the metamodels and ATL/DSLTrans transformations for each case study are available on our website [3].

Metrics for the size and complexity of the input and output metamodels for each transformation are presented in Table 2. These metrics include the number of elements, associations, and inheritance relationships present in each metamodel.

Table 3 presents the number of matched rules, lazy rules, and helpers in the ATL transformation for each of our case studies.

6.3 Measures

To objectively answer our defined 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 con-

[4] AUTomotive Open System ARchitecture, `AUTOSAR.org/`

Table 3 Number and classification of rules in each ATL specification

Transformation	Num. Matched	Num. Lazy	Num. Helpers
Families-to-Person	5	0	0
Ext. Families-to-Person	8	2	0
GM-to-AUTOSAR	3	2	0
UML-to-Kiltera	7	13	3

tracts 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
- Maximum memory usage required by the contract prover

Note that the number of rules found 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 3.5.1. Both the path condition construction and contract proving processes were parallelized amongst four threads. 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 [3].

6.4 Results

Table 4 shows the performance results for proving contracts on our case studies. We shall now discuss these results in the context of the first two research questions. RQ3 and RQ4 will be discussed separately in Section 7 and Section 8.

6.4.1 RQ1: Applicability of the Technique

To answer our first research question 'Is our technique applicable to complex ATL transformations?', we have tested our contract prover on a number of transformations of varying sizes sourced from different application domains. Metrics for the case studies are presented in Section 6.2.

For each case study, contracts we expected to hold were successfully proved on all applicable path conditions. For other contracts, which do not hold in all cases, counter-examples were produced that indicate the exact combination of rules where the contract is not guaranteed to hold. These counter-examples were then manually examined to ensure their correctness.

For example, we attempted to prove the *Daughter-Mother* contract on the extended *Families-to-Person* transformation, as detailed in Section 5.3. 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. As this was an expected result, this raises our confidence in the correctness of the transformation.

Success of our contract prover on these case studies lets us conclude that we can apply our technique to a variety of complex ATL transformations with varying rule and metamodel sizes.

6.4.2 RQ2: Time and Memory Characteristics

To answer our second research question, 'How does the time and memory usage of the contract prover differ for each of our case studies?', we refer to the results in Table 4 which contains the performance results of our experiments.

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. As an example, the extended *Families-to-Persons* transformation produced three times more path conditions than the original *Families-to-Persons* transformation, even though the extended version has roughly twice the number of rules.

The exact number of path conditions produced will depend on the complexities of how rules can combine with each other, such as the number of dependencies between rules or even the number of elements in each rule. As well, since our path condition generation is implemented using graph-matching and rewriting, larger rules will also take longer to combine, increasing the time taken for path condition generation [32].

The time to prove contracts on each transformation is also dependent on a number of factors. Similar to path condition generation, the time taken for contract proving is roughly proportional to the number of path conditions generated for a transformation, the number

Table 4 Performance results

	ATL/ DSLTrans Rules	Path Conds. Generated	Time (s)	Contracts Proved	Time (s)	Memory (MB)
Families-to-Person	5 / 9	101	0.24	4	0.52	54
Extended Families-to-Person	10 / 19	366	3.89	10	7.35	59
GM-to-AUTOSAR (handbuilt)	5 / 9	13	0.18	9	0.15	58
GM-to-AUTOSAR (HOT)	5 / 9	10	0.26	9	0.15	60
UML-to-Kiltera	20 / 17	322	1.86	15	11.99	55

of contracts to be proved on that transformation, and the composition of path conditions and contracts.

Note that our contract proving times are different (and may indeed be worse) than those reported in [36] or earlier works. This is primarily due to the replacement of the core matching algorithm in the prover. Recall that matching of contracts onto path conditions is non-isomorphic, as discussed in Section 5.3. Previously our prover had a 'disambiguation' step to explicitly produce all possible overlapping of rules, which could then be matched with a standard isomorphic matcher. A new matching algorithm was designed to bypass this step and take these overlapping elements into account. This algorithmic improvement also has the side-effect of tending to produce more path conditions compared to our earlier works.

We note that this new matching algorithm has not been the subject of heavy optimization, and we expect further speed improvements in the future. In particular, we note that one contract in particular for the UML-to-Kiltera case study has terrible matching performance, taking 142 seconds to prove. This is solely due to our unoptimized matching algorithm. In particular, the contract contains a *New* element connected to four *Name* elements. As well, this structure is repeated a number of times in the transformation's path conditions. As our matching algorithm is currently based on an association-based approach, a combinational explosion occurs when the matcher attempts to return all possible matches. We consider this contract to be an edge case, and its proving time is not included in Table 4 as it is solely an artifact of our unoptimized matching algorithm. Future work will attempt to address this implementation issue.

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 the transformations used here as case studies, the prover memory usage is between 54 to 60 MB, which is less than 10 MB higher than the overhead to run the Python scripts.

Overall, our contract proving approach stays within a modest time and memory budget. All transformations have their path conditions generated and have their contracts proved within 15 seconds and 60 MB of memory. Indeed, even the edge case contract has a reasonable proving time of 142 seconds. Future work will pursue optimizations to our technique in order to prove contracts on even larger and more complex ATL and DSLTrans transformations.

7 Slicing Transformations

This section will expand on the slicing algorithm introduced in [36]. Contrary to the earlier work, this algorithm has now been made an automatic part of the contract prover.

The intention of this algorithm is to create the minimal set of transformation rules for a given contract, such that when this set is symbolically executed by the contract prover, the correct result is produced. Decreasing the number of rules that need to be symbolically executed allows for a significant decrease in the amount of time required for the proving process.

Thus, this slicer algorithm is our attempt to answer RQ3: 'Given a particular contract, can we reduce the time taken for contract proving through transformation slicing?'

7.1 Slicer Overview

There are three steps in our technique to slice the transformation for a contract.

The first step is to decompose the contract into its typed elements and associations. This information allows the slicing algorithm to determine which rules are required to be involved in the contract proving process.

The second step examines all rules in the transformation, and identifies those rules which contain the necessary elements for the contract to match over.

Finally, the third step determines if those rules selected in the second step require elements that are produced by earlier rules. This must be an iterative process to allow all required rules to execute. Note that this dependency analysis is currently performed very conservatively, and future work will attempt to optimize the process to eliminate more rules.

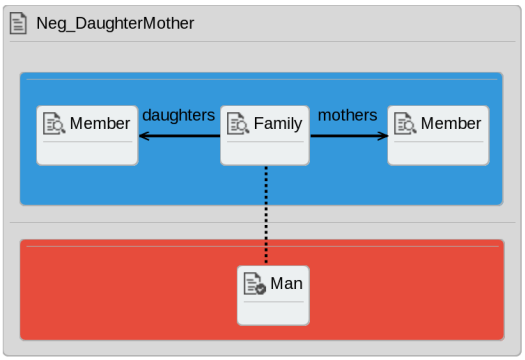


Fig. 22 Example contract for the slicing process

All those rules not required for the contract are then removed from the transformation to be verified. Note that this new transformation may be smaller than the original, but this depends on the specific dependencies between the contract and the transformation rules.

7.1.1 Decomposing Contract and Rules

To support the slicing algorithm, it is necessary to determine which rules the contract (and other rules) require for execution. This is determined by 'decomposing' the underlying typed graph structure to determine precisely which elements it matches over. In the current implementation, the backward links and associations between elements in the graph are extracted, as well as any elements which are not connected to others, termed *isolated elements*.

For example, consider the contract in Figure 22. There are two associations in the pre-condition, where each is composed of a *Member* element connected to a *Family* element. One association is typed by *daughters*, while the other is typed by *mothers*. As well, there is the backward link which connects an element in the pre-condition with an element in the post-condition, enforcing that a *Man* element has been created by the same rule that matched the *Family* element.

The second step to the slicing algorithm is to examine all the rules in the transformation. Recall that the path conditions which the contract matches over have been created by combining rules (cf. Section 5.2). Therefore, for the associations in the contract to appear in the path condition, one of the rules which contain this association must have been symbolically executed.

Each rule is examined to determine if there are isomorphic copies of the contract associations, backward links, or isolated elements present in the rule. If so, then that rule may produce the required association or element, so the rule is marked as crucial to the proving of the contract.

As a clarification for backward links, recall that as backward links must match over traceability links, where an input element produces an output element as part of the same rule. Therefore, this step succeeds when both elements connected by the backward link are found in the rule, despite no explicit traceability link between them.

This collection step is very conservative, as it includes all rules which contain any of the contract associations or isolated elements. However, it is correct to reason about the contract as a collection of associations and isolated elements rather than a complete graph. Recall that in Section 5.3, the contract is not matched isomorphically onto the path condition, due to the path condition structure representing the execution of a set of transformation rules. Therefore, the elements required for the contract may be 'split' amongst many rules in the transformation, necessitating a decomposition approach.

Note that if any association, backward link, or isolated element in the contract cannot be found in the full set of transformation rules, then the contract cannot match on any path condition produced by the proving algorithm. This indicates that either the contract or transformation contains errors and must be fixed.

The final step is to reason about the rules marked as crucial for contract proving in the second step. The decomposition and searching steps described above are repeated for each rule. This produces an extremely conservative rule-dependency graph, which indicates the rules that must be present in the transformation for this contract to be correctly proven.

7.2 Results and Discussion

To measure the impact of the slicing algorithm on contract proving, we examined the effects of slicing the *UML-to-Kiltera* transformation. A number of contracts were proven on both the whole transformation (denoted as the 'original' version) as well as the subset of the transformation returned by the slicing algorithm (the 'sliced' version). Note that the time taken to perform slicing itself was less than 0.05 seconds for all contracts.

The results in Table 5 show the reduction in contract proving time when slicing is performed. The original UML-to-Kiltera transformation, which originally contained 17 DSLTrans rules, has been sliced into subsets ranging from 2 to 15 rules for each contract. Note that the sliced transformations produce at most half the number of path conditions as the original transformation, greatly lowering both path condition construction time and contract proving time. Furthermore,

Table 5 Effect of slicing on contract proving time for the *UML-to-Kiltera* transformation

Name	Version	Rules	PCs	PC Build Time (s)	Prove Time (s)
PP1	*Original*	17	322	1.64	6.77
	Sliced	14	161	0.93	3.26
PP2	*Original*	17	322	1.80	6.63
	Sliced	14	161	0.94	3.15
PP3	*Original*	17	322	1.75	141.15
	Sliced	14	161	0.89	139.41
PP4	*Original*	17	322	1.85	7.02
	Sliced	14	161	1.01	3.42
MM1	*Original*	17	322	1.47	5.29
	Sliced	2	3	0.05	0.09
MM2	*Original*	17	322	1.68	7.01
	Sliced	8	64	0.13	0.12
MM3	*Original*	17	322	1.87	7.06
	Sliced	11	64	0.55	0.62
MM4	*Original*	17	322	1.84	7.00
	Sliced	11	64	0.58	0.64
MM5	*Original*	17	322	1.84	7.00
	Sliced	12	99	0.76	1.18
MM6	*Original*	17	322	1.71	6.33
	Sliced	2	3	0.04	0.08
MM7	*Original*	17	322	1.55	5.65
	Sliced	8	7	0.13	0.11
MM8	*Original*	17	322	1.84	6.84
	Sliced	12	99	0.74	1.14
MM9	*Original*	17	322	1.81	7.03
	Sliced	12	99	0.77	1.13
MM10	*Original*	17	322	1.47	5.29
	Sliced	11	64	0.59	0.67
MM11	*Original*	17	322	1.55	5.81
	Sliced	12	115	0.77	1.97
SS1	*Original*	17	322	1.57	5.89
	Sliced	15	112	0.28	0.74

the results of contract proof were identical for both the normal and sliced versions.

However, the number of rules in the slice depends on the particular elements involved in the contract and the rules. For example, slicing the transformation for the *MM6* contract produced a DSLTrans transformation with 2 rules, while a slicing for the *SS1* contract produced a transformation with 15 rules.

As described in Section 6.4.2, the proving time for the PP3 contract is a significant outlier from the rest of the contracts, even when the transformation is sliced. As mentioned, we consider this to be an artifact of our unoptimized matching algorithm.

These results show that the slicing of transformations based on the contract to be proven can have a large impact on the proving time. Path condition construction time was reduced by 43 to 97 percent, while contract proving time was reduced by 51 to 98 percent (excluding PP3). This answers our research question in the affirmative.

As well, contrary to our earlier work in [36], this slicing can now also be performed automatically during contract proving. Note that the current implementation of the slicer is based on a relatively simple decomposition of contract and rule graphs, along with construction of a conservative rule dependency graph. Future work will ensure that the minimum number of rules are selected in the sliced transformation.

We note that our slicing technique has definite parallels to other transformation verification works. For instance, [13] quantitatively compares the elements in 'Tracts' (an analogous version of our contracts[5]) to transformation rules to suggest to the user which rules are causing the Tract to fail. However, their approach differs from ours in two fundamental ways. First, the approach of [13] focuses on guiding the user towards the problematic rules[6]. Our slicing approach is a performance optimization to reduce the number of path conditions that must be created. Second, our sliced transformation must contain all rules that could change the result of a contract holding or not holding on a transformation. We cannot afford a false result in our verification as is allowed in [13], and thus our set of rules must be conservatively built.

8 Hand-built versus HOT-produced Transformations

This section will investigate our last research question RQ4, 'Does the version of the transformation produced by our higher-order transformation differ significantly from a hand-built transformation?' The case studies of interest are the *GM-to-AUTOSAR* and *UML-to-Kiltera* transformations. As the DSLTrans transformations are generated directly from ATL transformations, it is illuminating to directly compare these produced transformations to hand-built versions created by our academic partners in earlier work. In particular, we are interested in the performance penalty due to non-optimized transformations. If the penalty is minor or non-existent, then the HOT can serve as an automatic replacement to building the transformation by hand.

[5] Related work concerning Tracts is discussed in more detail in Section 9.

[6] Note that this guidance is partially addressed in our work. When path conditions fail a contract, we report the path condition which represents the minimum number of rules. Therefore, it is the interaction of these rules that cause the contract to fail.

8.1 GM-to-AUTOSAR Transformation

As discussed in Section 6.1.2, this transformation migrates models from a proprietary General Motors metamodel to an industry standard metamodel [44].

8.1.1 Transformation Shape

For brevity, the hand-built and HOT-produced versions of the transformation will not be presented as figures. Instead, the transformations are summarized in Tables 6 and 7 by listing the number of match and apply elements in each rule. The full transformations can be found on our website [3].

Table 6 GM-to-AUTOSAR (Hand-built) transformation structure

Layer	Rule Name	Match Elements	Apply Elements
1	MapPN2FiveElements	1	5
	Map Module	3	2
	MapPartition	2	1
2	ConnECU2VirtDev1	2	2
3	ConnVirtDev2Distrib1	3	2
4	ConnVirtDev2Distrib2	2	2
5	ConnECU2VirtDev2	2	2
6	ConnPPortProto	5	2
	ConnRPortProto	5	2
Total	9	25	20

Table 7 GM-to-AUTOSAR (HOT) transformation structure

Layer	Rule Name	Match Elements	Apply Elements
1	createComponent	1	2
2	initSysTemp	3	6
3	initSwc2EcuMap	3	1
4	sysMapping	2	2
5	compostype	3	2
6	mappingcomponent	2	2
7	mappingECUinstance	2	2
8	pportprototype	5	2
9	rportprototype	5	2
Total	9	26	21

Note that both the hand-built and HOT-produced versions of the transformation have nine rules, and the number of match and apply elements produced are approximately equivalent. This indicates that the HOT is producing a transformation that is roughly similar in complexity to what a human would build. Note however that the HOT currently produces a transformation which contains one rule per layer.

8.1.2 Effect on Contract Proving

Our research question asks whether the HOT-produced transformation is sufficient to replace the hand-built transformation when contract proving. The following results in Table 8 show the comparison between proving each contract on the two versions of the transformation. Note that 13 path conditions were generated for the hand-built GM-to-AUTOSAR transformation, while 10 path conditions were generated for the HOT-produced version.

We note that all contracts are proved in an almost equivalent amount of time, and have comparable results between the two versions of the transformation. The contract failures for M1 and M3 are expected, as the original ATL transformation contained errors [42].

Table 8 GM-to-AUTOSAR version effect on contract proving

Name	Version	PCs Succ.	PCs Fail.	Prove Time (s)
M1	*Hand-built*	4	8	0.06
	HOT	4	4	0.05
M2	*Hand-built*	12	0	0.05
	HOT	8	0	0.05
M3	*Hand-built*	4	4	0.05
	HOT	4	4	0.05
M4	*Hand-built*	8	0	0.06
	HOT	8	0	0.05
M5	*Hand-built*	12	0	0.06
	HOT	8	0	0.05
M6	*Hand-built*	12	0	0.06
	HOT	8	0	0.05
P1	*Hand-built*	6	0	0.07
	HOT	4	0	0.05
P2	*Hand-built*	6	0	0.07
	HOT	4	0	0.07
S1	*Hand-built*	4	0	0.06
	HOT	4	0	0.08

8.2 UML-to-Kiltera Transformation

As mentioned in Section 6.1.3, the UML-to-Kiltera transformation transforms UML-RT state machine diagrams into the Kiltera language for the purposes of verification and simulation. It is discussed in more depth in [37] and [42].

The authors happily note that the exact same rules were produced by the higher-order transformation from the ATL transformation code. In fact, all elements and names were consistent between the versions, allowing us to declare the hand-built and HOT-produced transformations functionally identical.

Contrary to our other case studies, it is also interesting to note that the DSLTrans version of the UML-to-Kiltera transformation contains only 17 rules compared to the 20 rules in the ATL version. This difference is due to the ATL version containing six trivial lazy rules which perform attribute setting.

8.3 Conclusion

The results for both experiments indicate that the produced DSLTrans transformations are of equivalent quality to the hand-built versions. There is a small to non-existent performance penalty in one case, and in the other the rules produced were identical to the hand-built version. Thus, we believe that it is sufficient to use our higher-order transformation as part of a toolchain to verify ATL transformations.

9 Related Work

There has been already an extensive work on verifying different aspects of model transformations, e.g., cf. [6, 40] for surveys in this domain. 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.

9.1 Model Transformation Testing

In [25, 48] the authors describe their method where 'Tracts' can be specified for model transformations. These 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 [35], graph patterns [9,27], Triple Graph Grammars (TGGs) [49], dedicated testing languages [22, 29], or as used in Tracts OCL constraints [17], and even a combination of these mentioned approaches [23]. 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 [26].

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 of the transformation, and then, the post-condition defines constraints on the output model.

9.2 Model Transformation Verification

Previous work also proposed the idea of transforming ATL to formal domains. The work of [47] 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 [47].

The work in [15] 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 [15] 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.

In [24] the authors are checking different kinds of model transformation properties based on OCL and the usage of KodKod which requires again concrete bounds for property proving. The work by Anastasakis et al. [7] transforms QVT 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. Similar model transformation verification support based on Alloy is presented in [21] which also needs concrete bounds for performing the model search.

Besides the mentioned bounded verification approaches, there are some unbounded approaches using theorem provers for verifying model transformations. For instance, Calegari et al. [16] propose an interactive approach to verify contracts for ATL transformations based on the Coq proof assistant. This approach is unbounded, but requires some user guidance. Another approach using the Coq proof assistant to ensure the correctness of model transformations is presented in [38]. However,

the authors aim to synthesize transformation implementations from specifications which are correct-by-construction instead of verifying independently developed transformations.

Other approaches using theorem provers for model transformation verification go one step further by using modern SMT solvers such as is done in [14, 18]. These approaches do not require user guidance as it was required in the aforementioned Coq based approaches. They translate the ATL transformations as well as the contracts expressed in OCL into first-order logic expressions and use Z3 for performing the theorem proving. Compared to our approach, these approaches are in the same spirit, but they consider a smaller subset of ATL compared to our solution. For instance, currently they do not support lazy rules.

Another work which translates ATL transformations for analysis purposes is presented in [41]. The authors argue that an algebraic graph transformation representation would allow for enhanced verification tasks. However, the authors do not go into detail on this aspect as they mainly focus on the correct translation of ATL language concepts to Henshin concepts. Thus, they consider the exploration of concrete verification tasks as future work.

In [30], the authors present a generic transformation metamodel which can be used as an intermediate language for translating model transformation languages to this representation, before the transformations are transformed into a formal domain for performing analysis. The authors present several verification cases where the proposed framework helps in exploiting different verification formalisms and techniques. In our approach, we also aim for reusing an existing verification formalism provided by DSLtrans and show how a considerable subset of ATL can be translated into DSLtrans to perform contract verification.

9.3 Synopsis

To the best of our knowledge, in this paper we have presented the only approach to fully prove properties defined as contracts for model transformations expressed in declarative ATL including advanced features such as lazy rules.

10 Conclusion

This section will offer a brief discussion as to threats of validity, as well as a number of concluding thoughts on our contract prover and technique.

10.1 Threats to Validity

This subsection discusses 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 4. 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 two transformations created by hand, and the corresponding transformations generated by our higher-order transformation in Section 8. We note that one transformation produced by our HOT was exactly the same (modulo minor rule rearrangement) as the hand-built version. As well, the other transformation was verified with similar proving time compared to the hand-built version. 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 the HOT and the contract prover can transform and verify reasonably sized and complex transformations. As well, the slicing algorithm presented is able to reduce the verification time significantly for a complex transformation. 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.

As DSLTrans is a Turing-incomplete computing language, it has 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 in out-place transformations using the default mode, which is used in many more transformations than the refining mode. We are thus confident our technique is usable for a large class of real-world problems.

Finally, only the *String* type is available in SyVOLT: this limitation implies that proofs in SyVOLT can only

be built for model transformations that manipulate attributes of types *String*. Note that this is not a limitation of the SyVOLT prover itself, but rather of the expressiveness of the DSLTrans language. This limitation can be surmounted in various ways, perhaps by converting non-String attributes and operations into Strings before transformation and doing the reverse operation after the transformation is concluded.

10.2 Conclusion

In this paper, we have expanded on our novel technique from [36] to fully verify pre-/post-condition contracts on declarative ATL transformations. This approach is centered around transforming ATL transformations into DSLTrans, our transformation language with reduced-expressiveness. Our path condition generator is then 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 reasonably sized and complicated ATL transformations, and that contracts can be proved using a feasible amount of time and memory. As well, we have also further detailed our '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 determined that our HOT produces transformations that are a suitable replacement for hand-built transformations in contract proving.

10.3 Future Work

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. Our current focus is on integrating our higher-order transformation into our SyVOLT tool [5].

Another ongoing concern of ours is the time and space requirements to prove contracts on large transformations. We are investigating implementation speedups, such as further optimization of our matching algorithm and refinement of the transformation slicer.

Acknowledgments

The authors warmly thank Gehan Selim and Cláudio Gomes for their contributions to the implementation of the contract prover.

Bentley James Oakes is funded by an NSERC grant, as well as support from the NECSIS project, funded by Automotive Partnership Canada.

The work of Javier Troya is funded by the European Commission (FEDER) and the Spanish and the Andalusian R&D&I programmes under grants and projects BELI (TIN2015-70560-R), THEOS (P10-TIC-5906) and COPAS (P12-TIC-1867).

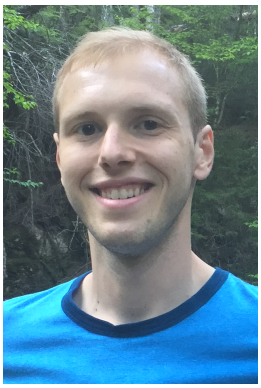
Finally, the work of Manuel Wimmer is funded by the Christian Doppler Forschungsgesellschaft and the BMWFW, Austria.

References

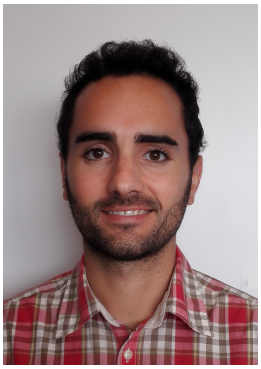
1. A Short Introduction to SyVOLT. https://www.youtube.com/watch?v=8PrR5RhPptY
2. ATL Zoo. http://www.eclipse.org/atl/atlTransformations
3. ATL2DSLTrans Artifacts. http://msdl.cs.mcgill.ca/people/levi/files/MODELS2015_SoSyM
4. Atlas Transformation Language – ATL. http://eclipse.org/atl
5. SyVOLT tool. http://msdl.cs.mcgill.ca/people/levi/contractprover
6. Amrani, M., Lúcio, L., Selim, G.M.K., Combemale, B., Dingel, J., Vangheluwe, H., Traon, Y.L., Cordy, J.R.: A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In: Proc. of ICSTW, pp. 921–928 (2012)
7. Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of Model Transformations via Alloy. In: Proc. of MoDeVVa (2007)
8. Arendt, T., Habel, A., Radke, H., Taentzer, G.: From Core OCL Invariants to Nested Graph Constraints. In: Proc. ICGT, pp. 97–112 (2014). DOI 10.1007/978-3-319-09108-2_7
9. Balogh, A., et al.: Workflow-Driven Tool Integration Using Model Transformations. In: Graph Transformations and Model-Driven Engineering, pp. 224–248 (2010)
10. Barroca, B., Lúcio, L., Amaral, V., Félix, R., Sousa, V.: DSLTrans: A Turing Incomplete Transformation Language. In: Proc. of SLE, pp. 296–305 (2011)
11. Bergmann, G.: Translating OCL to Graph Patterns. In: Proc. of MoDELS, pp. 670–686 (2014). DOI 10.1007/978-3-319-11653-2_41
12. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers (2012)
13. Burgueno, L., Troya, J., Wimmer, M., Vallecillo, A.: Static Fault Localization in Model Transformations. IEEE Transactions on Software Engineering **41**(5), 490–506 (2015)
14. Büttner, F., Egea, M., Cabot, J.: On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers. In: Proc. of MoDELS, pp. 432–448 (2012). DOI 10.1007/978-3-642-33666-9_28

15. Büttner, F., Egea, M., Guerra, E., De Lara, J.: Checking Model Transformation Refinement. In: Proc. of ICMT, pp. 158–173 (2013)
16. Calegari, D., Luna, C., Szasz, N., Tasistro, A.: A Type-Theoretic Framework for Certified Model Transformations. In: Proc. of SBMF, pp. 112–127 (2010). DOI 10.1007/978-3-642-19829-8_8
17. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL Contracts for the Verification of Model Transformations. ECEASST **24** (2009)
18. Cheng, Z., Monahan, R., Power, J.F.: A Sound Execution Semantics for ATL via Translation Validation. In: Proc. of ICMT, pp. 133–148 (2015). DOI 10.1007/978-3-319-21155-8_11
19. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. Springer (2007)
20. Cuadrado, J.S., Guerra, E., de Lara, J.: Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In: Proc. of ISSRE, pp. 34–44 (2014). DOI 10.1109/ISSRE.2014.10
21. Gammaitoni, L., Kelsen, P.: F-Alloy: An Alloy Based Model Transformation Language. In: Proc. of ICMT, pp. 166–180 (2015). DOI 10.1007/978-3-319-21155-8_13
22. García-Domínguez, A., Kolovos, D.S., Rose, L.M., Paige, R.F., Medina-Bulo, I.: EUnit: A Unit Testing Framework for Model Management Tasks. In: Proc. of MoDELS, pp. 395–409 (2011)
23. Giner, P., Pelechano, V.: Test-Driven Development of Model Transformations. In: Proc. of MoDELS, pp. 748–752 (2009)
24. Gogolla, M., Hamann, L., Hilken, F.: Checking Transformation Model Properties with a UML and OCL Model Validator. In: Proc. of VOLT, pp. 16–25 (2014)
25. Gogolla, M., Vallecillo, A.: Tractable Model Transformation Testing. In: Proc. of ECMFA, pp. 221–235 (2011)
26. González, C.A., Cabot, J.: ATLTest: A White-Box Test Generation Approach for ATL Transformations. In: Proc. of MoDELS, pp. 449–464 (2012)
27. Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated Verification of Model Transformations Based on Visual Contracts. Automated Software Engineering **20**(1), 5–46 (2013)
28. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Sci. Comput. Program. **72**(1-2), 31–39 (2008)
29. Kolovos, D.S., Paige, R.F., Polack, F.A.: Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In: Proc. of GaMMa, pp. 13–20 (2006)
30. Lano, K., Clark, T., Rahimi, S.K.: A Framework for Model Transformation Verification. Formal Asp. Comput. **27**(1), 193–235 (2015). DOI 10.1007/s00165-014-0313-z
31. Lúcio, L., Barroca, B., Amaral, V.: A Technique for Automatic Validation of Model Transformations. In: Proc. of MoDELS, pp. 136–150 (2010)
32. Lúcio, L., Oakes, B., Vangheluwe, H.: A Technique for Symbolically Verifying Properties of Graph-Based Model Transformations. Tech. rep., Technical Report SOCS-TR-2014.1, McGill University (2014)
33. Lúcio, L., Oakes, B.J., Gomes, C., Selim, G.M., Dingel, J., Cordy, J.R., Vangheluwe, H.: SyVOLT: Full Model Transformation Verification Using Contracts. In: Proc. of MoDELS 2015 Demo and Poster Session (2015)
34. Lúcio, Levi and Amrani, Moussa and Dingel, Jürgen and Lambers, Leen and Salay, Rick and Selim, Gehan and Syriani, Eugene and Wimmer, Manuel: Model Transformation Intents and their Properties. Software & Systems Modeling pp. 1–38 (2014). DOI 10.1007/s10270-014-0429-x
35. Mottu, J.M., Baudry, B., Traon, Y.L.: Model Transformation Testing: Oracle Issue. In: Proc. of ICSTW, pp. 105–112 (2008)
36. Oakes, B.J., Troya, J., Lúcio, L., Wimmer, M.: Fully Verifying Transformation Contracts for Declarative ATL. In: Proc. of MoDELS, pp. 256–265 (2015)
37. Paen, E.: Measuring Incrementally Developed Model Transformations Using Change Metrics. Master's thesis, Queen's University (2012)
38. Poernomo, I., Terrell, J.: Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq. In: Proc. of ICFEM, pp. 56–73 (2010). DOI 10.1007/978-3-642-16901-4_6
39. Posse, E., Dingel, J.: An Executable Formal Semantics for UML-RT. Software & Systems Modeling **15**(1), 179–217 (2016). DOI 10.1007/s10270-014-0399-z
40. Rahim, L., Whittle, J.: A Survey of Approaches for Verifying Model Transformations. Software & Systems Modeling **14**(2), 1003–1028 (2015). DOI 10.1007/s10270-013-0358-0
41. Richa, E., Borde, E., Pautet, L.: Translating ATL Model Transformations to Algebraic Graph Transformations. In: Proc. of ICMT, pp. 183–198 (2015). DOI 10.1007/978-3-319-21155-8_14
42. Selim, G.M.: Formal Verification of Graph-Based Model Transformations. Ph.D. thesis, Queen's University (2015)
43. Selim, G.M., Cordy, J.R., Dingel, J., Lúcio, L., Oakes, B.J.: Finding and Fixing Bugs in Model Transformations with Formal Verification: An Experience Report. In: Proc. of AMT, pp. 26–35 (2015)
44. Selim, G.M., Lúcio, L., Cordy, J.R., Dingel, J., Oakes, B.J.: Specification and Verification of Graph-Based Model Transformation Properties. In: Proc. of ICGT, pp. 113–129 (2014)
45. Syriani, E., Vangheluwe, H., LaShomb, B.: T-Core: a framework for custom-built model transformation engines. Software & Systems Modeling **14**(3), 1215–1243 (2015). DOI 10.1007/s10270-013-0370-4
46. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Refining Models with Rule-based Model Transformations. Research Report RR-7582, INRIA (2011)
47. Troya, J., Vallecillo, A.: A Rewriting Logic Semantics for ATL. Journal of Object Technology **10**(5), 1–29 (2011). DOI 10.5381/jot.2011.10.1.a5
48. Vallecillo, A., Gogolla, M., Burgueno, L., Wimmer, M., Hamann, L.: Formal Specification and Testing of Model Transformations. In: Formal Methods for Model-Driven Engineering, pp. 399–437 (2012)
49. Wieber, M., Anjorin, A., Schürr, A.: On the Usage of TGGs for Automated Model Transformation Testing. In: Proc. of ICMT, pp. 1–16 (2014)

Author Biographies



Bentley James Oakes is a PhD candidate in the Modelling, Simulation, and Design Lab at McGill University in Canada. His PhD topic is on the verification of model transformations in various domains using the SyVOLT frameworks and tool. Other research interests include causal-block diagrams, intellectual property issues in models, and artificial intelligence. Further information on his research can be found at `http://msdl.cs.mcgill.ca/people/bentley/`.



Javier Troya received his PhD degree in 2013 from the University of Malaga, Spain. He is currently a postdoctoral researcher in the Department of Computer Science and Languages at the University of Seville, Spain. Previously, he has been a postdoctoral researcher in the Business Informatics Group (BIG) at the Vienna University of Technology for more than two years. His research interests include modeling and metamodeling, model transformations, non-functional properties analysis and metamorphic testing. For more information, please visit `http://www.lsi.us.es/~jtroya`.

Levi Lúcio is currently a staff researcher and Project Manager at fortiss GmbH, Germany. He received his PhD from the University of Geneva, Switzerland, in 2008. His research is about bridging software engineering and formal techniques. Some of his concrete areas of interest are model-driven development, model transformation languages, the verification of model transformations, correctness-by-construction, models of concurrency (in particular Algebraic Petri Nets), model evolution, model-based testing and tool construction. Levi is currently developing and leading projects together with avionic and automotive companies to produce IDEs based on frameworks that seamlessly integrate a range of domain specific languages. Such frameworks aim at providing the right languages for the right modelling tasks, while offering verification, refinement and traceability services. One of the main goals of such frameworks is to improve the available means for the certification of safety-critical software by the relevant authorities.

Manuel Wimmer is a post-doctoral researcher at the Business Informatics Group of TU Wien. His research interests include the foundations of model engineering techniques as well as their application in domains such as tool interoperability, legacy modeling tool modernization, model versioning and evolution, software reverse engineering and migration, web engineering, cloud computing, and smart production. For further information about his research activities, please visit `http://big.tuwien.ac.at/staff/mwimmer`.