Improving Higher-Order Transformations Support in ATL

Massimo Tisi, Jordi Cabot, Frédéric Jouault

AtlanMod, INRIA & École des Mines de Nantes, France
{massimo.tisi, jordi.cabot, frederic.jouault}@inria.fr

Abstract. In Model-Driven Engineering (MDE), Higher-Order Transformations (HOTs) are model transformations that analyze, produce or manipulate other model transformations. In a previous survey we classified them, and showed their usefulness in different MDE scenarios. However, writing HOTs is generally considered a time-consuming and error-prone task, and often results in verbose code.
In this paper we present several proposals to facilitate the definition of HOTs in ATL. Each proposal focuses on a specific kind of scenario. We validate our proposals by assessing their impact over the full list of HOTs described in the survey.

1 Introduction

With the continuous growing in the popularity of the transformational approach in Model Driven Engineering (MDE), a vast number of model transformations is being developed and organized in complex patterns. Model transformations are becoming a common technological means to handle models both at development time and at runtime.

In complex environments, where transformations are a central artifact of the software production and structure, the idea of transformation manipulation naturally arises to automatically generate, adapt or analyze transformation rules. While transformation manipulation can already be performed by means of an independent methodology (e.g., program transformation, aspect orientation), the elegance of the model-driven paradigm allows again the reuse of the same transformation infrastructure, by defining model transformations as models [4]. The transformation is represented by a transformation model that has to conform to a transformation metamodel. Just as a normal model can be created, modified, augmented through a transformation, a transformation model can itself be instantiated, modified and so on, by a so-called Higher Order Transformation (HOT). This uniformity is beneficial in several ways: especially it allows reusing tools and methods, and it creates a framework that can in theory be applied recursively (since transformations of transformations can be transformed themselves).

In previous work [17], we investigated how developers have used HOTs to build applications. We detected four wide usage classes, Transformation Analysis, Synthesis, Modification and Composition, and for each class we described how HOTs were integrated in the application structure. To perform this analysis we gathered a set of 41 HOTs in different transformation languages, comprising all the ATL HOT applications published at that date, to our knowledge.

A preliminary analysis of this dataset confirmed an intuitive perception about HOT programming: while HOTs are becoming more and more necessary for complex transformational scenarios, HOT programming tends to be perceived as a tedious activity. HOTs are generally very verbose, even in simple cases. This seemingly unnecessary verbosity could strongly hamper the diffusion of the HOT paradigm. It makes HOTs less readable and the whole development activity more time-consuming and error-prone.

It is important to remark that the reasons for the verbosity of HOTs are not to ascribe to a fault in the transformation language design. The manipulation of transformation models is made complex by the fact that metamodels of programming languages (such as ATL) are inherently complex. For the sake of uniformity, model transformations deal with these complex metamodels in the same way as they deal with any other metamodel, without providing any specific facility. This paper represents a first step in the direction of improving the definition of this special kind of transformation.

In this sense, the contribution of the paper is a set of proposals for increasing the productivity of HOT programming in ATL, by providing both a support library and directly extending the ATL language syntax and semantics. The paper provides an assessment of the impact of these proposals on the length of real-world HOTs. While we focus our discussion on ATL, the results of this paper can be easily generalized to other transformation environments.

The rest of the paper is organized as follows: Section 2 introduces the basic concept of HOT; Section 3 suggests a set of practical enhancements to improve HOT programming in ATL; Section 4 evaluates the outcome of applying the previous proposals to a set of real-world HOTs; Section 5 compares our approach to HOTs with related work; Section 6 draws the conclusions.

2 Higher-Order Transformations

An essential prerequisite for fully exploiting the power of transformations is the ability to treat them as subjects of other transformations. In an MDE context, this demands the representation of the transformation as a model conforming to a transformation metamodel.

Not all transformation frameworks provide a transformation metamodel. In this work we will refer to the AmmA framework [13] that contains a mature implementation of the ATL transformation language. Within AmmA an ATL transformation is itself a model, conforming to the ATL metamodel. Besides the central classes of *Rule*, *Helper*, *InPattern*, and *OutPattern* the ATL metamodel also incorporates the whole OCL metamodel to write expressions to filter and manipulate models.

Once the representation of a transformation as a transformation model is available, a HOT can be defined as follows:

Definition 1 (Higher-order transformation). *A higher-order transformation is a model transformation such that its input and/or output models are themselves transformation models.*

According to this definition HOTs either take a transformation model as input, produce a transformation model as output, or both.

The typical schema of a HOT, particularized for the AmmA framework, is shown in Figure 1. This example reads and writes a transformation, e.g. with the purpose of performing a refactoring. The three operations shown as large arrows at level M1 (Models) are:

- *Transformation injection.* The textual representation of the transformation rules is read and translated into a model representation. This translation in AmmA is performed using TCS [11]. The generated model is an instance of the ATL metamodel.
- *Higher-order transformation.* The transformation model is the input of a model transformation that produces another transformation model. The input, output and HOT transformation models are all conforming to the same ATL metamodel.
- *Transformation extraction.* Finally an extraction is performed to serialize back the output transformation model into a textual transformation specification.

Note that the injection an extraction operations are not always involved in a HOT. For instance, the source transformation model may come from a previous transformation, and already be in the form of a model. Similarly, the target transformation model is sometimes reused as a model without need to serialize it.

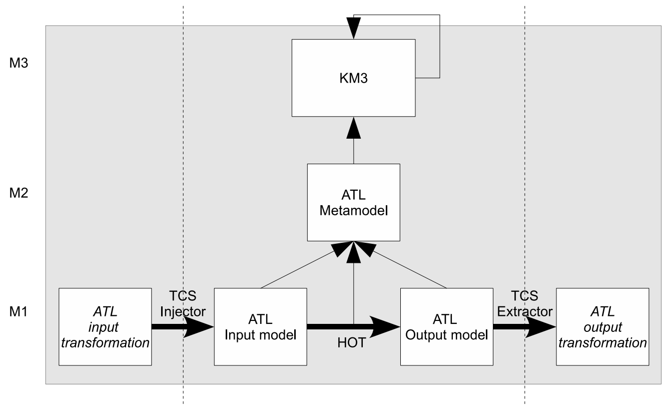

Fig. 1. A typical Higher-Order Transformation.

3 Facilitating HOT Development

To approach the problem of facilitating HOT development we started, in [17], by identifying four groups of similar HOT applications and gathering real-word examples for each one of them. The transformation classes can be characterized by their input and output models, in a schema that we briefly summarize here:

Transformation analysis: HOTs that process other transformations to extract meaningful data. They have at least one transformation as input model, no transformations as output models.

Transformation synthesis: HOTs that create new transformations from data modeled in other forms. They have no transformations as input models, at least one transformation as output model.

Transformation modification: HOTs that manipulate the logic of an input transformation. They have one transformation as input model, one transformation as output model.

Transformation (de)composition: HOTs that merge or split other transformations, according to a (de)composition criterion. They have at least one transformation as input model, at least one as output model, and the input and/or the output models contain more than one transformation.

Inspection of previous work on HOTs and our experience with HOT development in ATL, has highlighted that the development of transformations in each one of these classes shows peculiar and recurrent problems. In this section we describe the problems and, for every class, we present a proposal to allow simpler and more concise HOTs in ATL.

We believe our results can be generalized to other transformation frameworks. In this sense we also believe that this paper could be an useful working example of transformation language extension for higher-order applications in every framework.

The choice to focus the analysis on ATL is justified by the fact that previous work in [17] has shown that ATL is the preferred language for HOTs development to date. The ideas of this paper are based on the analysis of a set of 42 freely available transformations in ATL, that constitute an up-to-date sample of HOT applications in industry and research. Moreover, this significant set of examples are leveraged for validating our proposals in Section 4.

When possible we base our proposals on the built-in extension mechanisms of ATL. ATL provides three ways to reuse transformation modules.

Libraries. Collections of helpers, i.e. query operations whose body is defined by OCL expressions.

Module superimposition. A *Module A* can be *superimposed* to a *Module B*, obtaining a *Module C*, by a simple kind of internal composition such that: 1) *C* contains the union of the sets of transformation rules and helpers of *A* and *B*; 2) *C* does not contain any rule or helper of *B*, for which *A* contains a rule or helper with the same *name* and the same *context*.

Transformation composition. ATL *Modules* can be composed by executing them sequentially or in articulated patterns, usually using ANT scripts (this kind of composition is usually called *external composition*).

In this paper we will make use of libraries and transformation composition. Especially the second technique is particularly powerful, thanks to the uniformity of the transformational paradigm. For instance, HOT composition allows us to extend the ATL language without even touching the language implementation. In fact, it is possible in principle to add a higher-order preprocessing phase before the transformation execution, to have it translated in an equivalent version compatible with the standard environment.

When the extension mechanisms are not sufficient for our purposes we suggest extensions to the ATL language implementation and show the outcome of these extensions. Depending on the case, it will be more convenient to extend the ATL virtual machine or the compiler.

3.1 Transformation Analysis

HOTs in the transformation analysis class share a single aspect, i.e. being applied to transformation models conforming to the ATL metamodel. They need to navigate the ATL metamodel and return its instances, by defining filtering conditions on it.

In ATL, the logic for input navigation and filtering is scattered in several places: filters inside input patterns, variable definitions in the *using* part of rules, imperative sections. The navigation and filtering logic contained in these parts is composed by *OclExpressions* that can be very complex. ATL provides an ad-hoc means to modularize this logic, i.e. libraries of *Helpers*.

To facilitate the development of analysis HOTs we propose a HOT library, composed by those helpers that we found to be recurrently used within our set of HOTs. The helpers that we propose can be roughly divided in the following categories.

- Helpers that executes a query over all instances of a metaclass in the input metamodel. Our proposal includes:
  - helpers to retrieve ATL rules depending on the type of their matched or generated elements (*generatingRules*, *matchingRules*, *copyRules*),
  - a HOT helper to retrieve the calls to operations and helpers by providing their name (*callsByName()*),
  - an helper to check if the contextual element belongs to one of the input transformations (*belongsTo()*).
- Helpers that implement a recursive logic, when some associations between metaclasses can be navigated recursively. We provide the following helpers.
  - Within the OCL Package a *PropertyCallExpression* in the OCL metamodel has a *source* association with another *OclExpression*. This expression can have a source too, and so on. A *navigationRootExpression()* helper can navigate recursively the source relations to return the root of the chain.

- Another important example is given by the *refImmediateComposite()* operation, that returns the immediate container of the contextual element. We provide several recursive versions: *firstContainerOfType()* to get a container of a given generic type, *rootExpression()* to get the root of a complex OclExpression, *knownVariables()* and *containedVariables()* to get the variables defined within or before the contextual element.

Distributing the previous helpers in a standard HOT-specific library for ATL allows HOT developers to simply import them in any HOT. This would allow for shorter HOTs but also, in our opinion, it would foster a general improvement of the transformation design and reusability.

Table 1 lists all the helpers showing for each one the context type, the result type and a short description. This library of helpers is publicly available at [16].

Name	**Context**	**Returns**
Description.		
belongsTo(modelName: String)	ATL!OclAny	Boolean
return true if the element belongs to the specified model.		
firstContainerOfType(ATL!OclType)	ATL!OclAny	ATL!OclAny
compute the ATL element of the specified type in which the contextual element is declared		
rootExpression	ATL!OclExpression	ATL!OclExpression
navigationRootExpression	ATL!PropertyCallExp	ATL!OclExpression
return the root OCL element of the containment or navigation tree that includes the contextual element.		
knownVariables	ATL!OclAny	OrderedSet(ATL!VariableDeclaration)
containedVariables	ATL!OclAny	OrderedSet(ATL!VariableDeclaration)
computes an ordered set containing the VariableDeclarations that are defined higher or lower than the contextual ATL element in its namespace tree.		
generatingRules	ATL!OclModelElement	OrderedSet(ATL!Rule)
matchingRules	ATL!OclModelElement	OrderedSet(ATL!Rule)
copyRules	ATL!OclModelElement	OrderedSet(ATL!Rule)
computes an ordered set containing all the rules that can generate, match or copy the contextual element type.		
callsByName(name: String)	thisModule	OrderedSet(ATL!NavigationExp)
computes an ordered set containing the calls to operations with the given name.		

Table 1. A library for HOTs

3.2 Transformation Synthesis

HOTs in the transformation synthesis group are characterized by the task of producing new ATL code, as an output model conforming to the ATL metamodel. The production of ATL code is usually parametrized using one or more input models, but no assumption can be done on the structure of the input metamodels.

In ATL, new output elements are created by specifying them inside the *OutputPatterns* of transformation rules. An *OutputPattern* has to fully describe all the elements to create, with their attributes and associations. Usually "normal" transformation rules have simple output patterns and interact with each other to create complex models. In synthesis HOTs, instead, a single transformation rule is often in charge of creating several elements at once, leading to the development of complex *OutputPatterns*.

In our HOT dataset, typical examples of complex output patterns are related to the creation of:

– the root of a new transformation module, including references to its input and output metamodels;
– a new transformation rule or helper, together with its complete input and output patterns;
– complex OCL expressions.

These output patterns are generally very verbose. For example the second one, depending on the complexity of the generated transformation rule, can contain a large number of output elements and feature assignments. This results in a high percentage of LOC spent for building the output part of HOT rules.

Our experience has shown that the impact of output patterns in HOT verbosity is remarkable. For instance, Listing 1.1 shows the example of a HOT rule that synthetizes a new ATL rule. The example is taken from the KM32ATLCopier HOT [9] that, given a metamodel as input (in the KM3 format), generates a copier for that metamodel, i.e. a transformation that makes a copy of any input model conforming to that metamodel. The rule shown in Listing 1.1, generates a copier for each KM3 class.

As it can be seen in the example, the output pattern has to create an element for each metaclass of the abstract syntax, and fill the features of these elements with 1) constant values, 2) references among the elements, 3) variables calculated from the input model.

We propose to extend ATL, by allowing the production of ATL rules directly using an adapted version of the ATL concrete syntax when defining output patterns. Since ATL has an unambiguous concrete syntax, it is possible to parse it and derive which model elements have to be created and which constant values and inter-element references are needed.

Listing 1.2 shows the same rule as Listing 1.1, using the proposed syntax. Output pattens can be directly specified with an embedded concrete syntax by delimiting it using special sequences of characters, i.e. '|[' and ']|'. In the example the output pattern contains the whole rule that has to be generated (i.e. a simple copier). The embedded concrete syntax substitutes the long list of model elements and bindings in Listing 1.1. Inside of the concrete-syntax output patterns, it is possible to include strings obtained dynamically from the input models. As it is shown in Listing 1.2, a special notation is introduced for variables in the concrete syntax, denoted by the '%' character. The variables can be declared and defined in the *using* part of the transforamation rule, as any other variable. They are ideally computed and expanded inside the output pattern at runtime, after the rule has being matched.

This extension to ATL can be easily implemented using the above-mentioned technique of HOT composition: a HOT pre-processor can take as input the transformation in Listing 1.2 and rewrite it as in Listing 1.1, to make it executable on the default ATL environment. In this way, we can simplify the specification of synthesis HOTs, without changing the ATL compiler.

Listing 1.1. Original ATL rule

```
rule Class {
  from s : KM3!Class [...]
  to
    t : ATL!MatchedRule (
      isAbstract <- false,
      isRefining <- false,
      name <- 'Copy' + s.name,
      inPattern <- ip,
      outPattern <- op
    ),
      ip : ATL!InPattern (
        elements <- Sequence{ipe},
        filter <- f
      ),
        ipe :
          ATL!SimpleInPatternElement (
          varName <- 's',
          type <- ipet
        ),
          ipet : ATL!OclModelElement (
            name <- s.name,
            model <- thisModule.metamodel
          ),
          f : ATL!OperationCallExp (
            operationName <- 'oclIsTypeOf',
            source <- fv,
            arguments <- Sequence{ft}
          ),
          fv : ATL!VariableExp (
            name <- 's',
            referredVariable <- ipe
          ),
          ft : ATL!OclModelElement (
            name <- s.name,
            model <- thisModule.metamodel
          ),
      op : ATL!OutPattern (
        elements <- Sequence{ope}
      ),
        ope :
          ATL!SimpleOutPatternElement (
          varName <- 't',
          type <- opet,
          bindings <- b
        ),
          opet : ATL!OclModelElement (
            name <- s.name,
            model <- thisModule.metamodel
          )
}
```

Listing 1.2. Equivalent ATL rule

```
rule Class {
  from s : KM3!Class [...]
  using {
    name : String = 'Copy' + s.name;
    metamodel : String = thisModule.metamodel;
    meName : String = s.name;
  }
  to
```

```
    t : ATL!MatchedRule |[
      rule %name {
        from
          s : %metamodel!%meName (
            s.oclIsTypeOf(%metamodel!%meName)
          )
        to
          t : %metamodel!%meName
      }
    ]|
}
```

Our proposal can be generalized outside HOTs, to the concrete-syntax of any output model. In this case, the ATL launch configuration should allow the developer to specify any concrete syntax for output models, as a TCS file. The pre-processor we propose would parse the provided TCS file and use it into the pre-processing phase.

3.3 Transformation Modification and (De)Composition

Applications that modify ATL transformations and compose (or decompose) them, generally need to parse one or more input transformations and generate one or more output transformations. Hence, they contain both the aspects of input filtering and output creation that characterize transformation analysis and transformation synthesis. For this reason their development can benefit from both the proposals in Sections 3.1 and 3.2.

Moreover, while the transformation logic for modification and composition is generally strongly dependent on the applicative task, all the transformations of this class share a further important aspect, i.e. the need to select and transport chuncks of unchanged transformation code from the input to the output.

The ATL language provides an ad-hoc execution mode for transformations that perform little changes on the input model, i.e. ATL refining mode. In this mode, model elements that are not matched by an explicit transformation rule are copied, with all their features, to the output model.

However, HOT developers in previous work have sometimes preferred alternatives to the built-in ATL refining mode:

– in some cases developers use superimposition to base their transformation on *ATLModuleCopier*, a verbose HOT that simply performs a copy of the whole input transformation, and they override only a small set of rules;
– other cases make use of several element copier rules, i.e. transformation rules that copy a single ATL element from the input transformation to the output transformation, with all its features and sometimes only slight modifications;

The choice to rely on these alternatives was probably related to limitations in the implementation of refining mode in ATL2004 (e.g., the copying was performed implicitely only for contained elements of copied elements and it was mandatory to specify all bindings). The effort of copying some elements of a transformation, while modifying others, have been reduced by the latest version of the ATL language (ATL2006), that introduces *in-place refining mode*. In this mode every element stays unchanged if it is not explicitely matched by a transformation rule.

Our experience has shown that migrating to ATL2006 refining mode has already a big impact on the length of several HOTs, and we recommend the developers to consider in-place refining mode for every transformation modification and (de)composition.

However, a few HOTs that do not present a general semantics of refinement, need simply to copy a set of elements from the input to the output. In these cases a fine-graned refining mode could be beneficial, allowing the user to choose exactly which subset of the input model is subject to refinement. *Refining rules* are our proposal to give the developer the possibility to specify with minimal effort that a single element have to be copied to the output, together with all its contained and associated elements.

To describe our proposal, we show an excerpt from the MergeHOT transformation, that creates a new transformation by the simple union of transformation rules given in input. In this case, even the ATL2006 version of the refining mode is not optimal, because it is able to refine only a single input module. Listing 1.3 is a solution in normal execution mode, where the developer is forced to include a long list of copying rules for all the elements of the two models to merge (e.g., Binding, NavigationOrAttributeCallExp, VariableExp). Refining rules allow the substitution of all this code (more than 100 LOCs) with the excerpt in Listing 1.4. The refining rule states that the *MatchingRule* have to be copied to the output with a different name, and implicitly triggers the recursive copy of all the elements contained in the *MatchingRule*, making the other HOT rules superfluous.

Listing 1.3. Original ATL excerpt

```
rule matchedRule {
  from
    lr : ATL!MatchedRule (
      lr.isLeft or lr.isRight
    )
  to
    m : ATL!MatchedRule (
      name <- lr.fromLeftOrRight + '_' + lr.name,
      children <- lr.children,
      superRule <- lr.superRule,
      isAbstract <- lr.isAbstract,
      isRefining <- lr.isRefining,

      inPattern <- lr.inPattern,
      outPattern <- lr.outPattern
    )
}
rule inPattern {
  from
    lr : ATL!InPattern (
      lr.isLeft or lr.isRight
    )
  to
    m : ATL!InPattern (
      elements <- lr.elements
    )
}
...[100 lines]
```

Listing 1.4. Equivalent ATL excerpt

```
refining rule matchedRule {
  from
    lr : ATL!MatchedRule (
      lr.isLeft or lr.isRight
    )
  to
    m : ATL!MatchedRule (
      name <- lr.fromLeftOrRight + '_' + lr.name
    )
}
```

Similarly to the previous proposal, refining rules could provide a noticeable gain in HOT conciseness, but also a general impact on ATL productivity outside HOT development.

4 Experimentation

To assess the impact of our proposals on real-world transformations we evaluate a set of HOTs taken from industry and research publications. The set, shown in Table 2, comprises all publicly available HOTs written in ATL, to our knowledge. Most of the transformations in the dataset are already included in the survey from [17] (or they are updated versions of those transformations). We have chosen to remove from the dataset HOTs that are not hand-written but automatically generated, such as ATLCopier[17] because they can't be considered representative of HOT development. The set has also a few additions comprising HOTs for: detecting constraints for chaining transformations [5], generating transformations from matching models [7] or reconstructing matching models from transformations [12]. Moreover Table 2 comprises the HOT we have built in this paper, for the purpose of deriving structural data about our transformation dataset shown in the table. This HOT is executed iteratively over the experimentation set using an ANT script [8]. The Analysis HOT is a second-order HOT, since it expects another HOT as the input model. It generates an Analysis Model, conforming to an Analysis metamodel, whose content is then shown in tabular format. The complete code of the analysis tool can be found in [16].

For each HOT in the dataset Table 2 shows:

- the classification of the HOT inside one of the four classes proposed in [17]
- the execution mode of the transformation, i.e. normal or refining;
- the input and output metamodels of the HOT;
- size metrics for the transformations (number of rules, number of helpers, lines of code);
- experimentation results after individually applying each proposal: HOT-library (1), concrete syntax in output patterns (2) and improved refining mode (3); for each proposal we show:
  - the number of times the specific proposal can be applied on the HOT,
  - the new number of LOCs after applying the proposal,
  - the percentage of improvement;

- global experimentation results from the application of the three proposals at once, comprising:
  - the final number of LOCs,
  - the percentage of improvement with respect to the original code.

The experimentation results were obtained by manually rewriting the HOTs in Table 2 by means of: 1) substituting OCL expressions with calls to our HOT Library, 2) substituting complex output patterns with their resulting concrete syntax, 3) substituting copying rules with in-place rules (in ATL2006) or refining rules. Each one of the substitutions has the effect of decreasing the global length of the HOT.

Results in Table 2 show the optimizations have a clear impact on the simplification of HOTs specification. On average, we can observe a decrease of the LOCs by 35% (up to a 42% when considering the size of the transformations in the computation). Note that LOC as a software size metric counts as well comments and blank lines. Therefore, the real percentage of improvement on the transformation code length is in fact higher.

As expected, the impact of a optimization type on a specific HOT depends on the class the HOT belongs to. Therefore, to get the most of our optimizations with the minimal effort, HOT designers should first classify the HOT in one of the four categories and focus on applying the optimization described for that category.

Even if difficult to numerically quantify, this reduction in the HOT length brings some additional benefits: improves the modularity and reusability of HOTs, reduces the possibility of errors and facilitates their extensibility and maintanibility. We will conduct more experiments in the future to try to validate these assumptions as well.

5 Related Work

This paper can be placed among proposals to speed-up transformation development with pre-existing transformation languages. The main works in this area focus on transformations by example [3],[15], model transformation patterns [10], transformation generation [18]. Our work is the first to introduce specific extensions for HOTs.

In the MDE community there is no previous work in evaluating the extension of a model transformation language for specific transformation classes. This is also due to the fact that, apart from the HOT class, a widely recognized categorization of model transformations, independent of the transformation language, is not available. Some transformation languages, such as RubyTL [6], are designed with extensibility as one of the main requirements, but the potential of this extensibility has yet to be investigated.

Extension mechanisms are instead deeply leveraged in more mature languages in other technical spaces. For example, user communities develop around XSLT extensions [1]. Using the XSLT extension mechanism, [14] propose a library for Higher Order Functions (HOFs). The authors implement in XSLT a set of HOFs,

	Name	Class	Mode	Source Mms	Target Mms	# rules	# helpers	LOCs	Applications (1)	LOCs (1)	Improvement %	Applications (2)	LOCs (2)	Improvement %	Applications (3)	LOCs (3)	Improvement %	Final LOCs	Improvement %	Weighted Improvement %
1	AnalyzeTransformation	Analysis	normal	ATL,MOF	TA	2	12	263	4	222	**15.59**	0	263	**0**	0	263	**0**	222	**15.59**	
2	ATL2AMW	Analysis	normal	ATL,MOF,MOF,EqualMM	EqualMM	6	19	259	2	229	**11.58**	0	259	**0**	0	259	**0**	229	**11.58**	
3	ATL2Problem	Analysis	normal	ATL	Problem	18	20	923	14	470	**49.08**	0	923	**0**	0	923	**0**	470	**49.08**	
4	HOTAnalysis	Analysis	normal	ATL	Analysis	2	3	50	1	31	**38**	0	50	**0**	0	50	**0**	31	**38**	
	TOTAL Analysis					**28**	**54**	**1495**	**21**	**952**		**0**	**1495**		**0**	**1495**		**952**		
	AVERAGE Analysis					**7**	**13.5**	**373.75**	**5.25**	**238**	**28.56**	**0**	**373.75**	**0**	**0**	**373.75**	**0**	**238**	**28.56**	**36.32**
5	AMW2ATL_DoDAF	Synthesis	normal	AMW,MOF,MOF,XML	ATL	24	18	631	0	631	**0**	15	526	**16.64**	0	631	**0**	526	**16.64**	
6	AMWtoATL	Synthesis	normal	AMW	ATL	17	1	388	0	388	**0**	11	283	**27.06**	0	388	**0**	283	**27.06**	
7	AMWtoATL_Key2Nested	Synthesis	normal	AMW, MOF, MOF	ATL	17	2	373	0	373	**0**	9	294	**21.18**	0	373	**0**	294	**21.18**	
8	AMWtoATL_KM32SQL	Synthesis	normal	AMW	ATL	40	8	1149	0	1149	**0**	26	744	**35.25**	0	1149	**0**	744	**35.25**	
9	AMWtoATL_MantisBug	Synthesis	normal	AMW	ATL	22	3	489	0	489	**0**	11	380	**22.29**	0	489	**0**	380	**22.29**	
10	DUALLyLeft2Right	Synthesis	normal	AMW, MOF, MOF	ATL	30	42	1707	0	1707	**0**	12	1348	**21.03**	0	1707	**0**	1348	**21.03**	
11	GenDiff	Synthesis	normal	MOF	ATL	9	0	1056	0	1056	**0**	6	209	**80.21**	0	1056	**0**	209	**80.21**	
12	GenMatchBottomUp	Synthesis	normal	MOF,MMParams	ATL	11	0	944	0	944	**0**	9	285	**69.81**	0	944	**0**	285	**69.81**	
13	GenMatchFilter	Synthesis	normal	MOF,MMParams	ATL	5	0	709	0	709	**0**	4	157	**77.86**	0	709	**0**	157	**77.86**	
14	GenMatchTopDown	Synthesis	normal	MOF,MMParams	ATL	10	0	1086	0	1086	**0**	7	311	**71.36**	0	1086	**0**	311	**71.36**	
15	GenPatch	Synthesis	normal	DiffMM,MOF,LMM,RMM,MatchMM	ATL	6	6	499	0	499	**0**	8	425	**14.83**	0	499	**0**	425	**14.83**	
16	HOT_match	Synthesis	normal	EqualMM,MOF,MOF	ATL	7	1	157	0	157	**0**	3	132	**15.92**	0	157	**0**	132	**15.92**	
17	inverseAMWtoATL	Synthesis	normal	AMW, MOF, MOF	ATL	18	2	466	0	466	**0**	10	334	**28.33**	0	466	**0**	334	**28.33**	
18	KM32ATLCopier	Synthesis	normal	KM3	ATL	2	4	144	0	144	**0**	2	110	**23.61**	0	144	**0**	110	**23.61**	
19	KM32CONFATL	Synthesis	normal	KM3	ATL	7	1	1057	0	1057	**0**	7	225	**78.71**	0	1057	**0**	225	**78.71**	
20	meo2atl4model2rdf	Synthesis	normal	MOF, OWL, MEO	ATL	66	19	1018	0	1018	**0**	10	362	**64.44**	0	1018	**0**	362	**64.44**	
21	meo2atl4rdf2model	Synthesis	normal	MOF, OWL, MEO	ATL	63	17	802	0	802	**0**	7	725	**9.6**	0	802	**0**	725	**9.6**	
22	MM2Profile	Synthesis	normal	AMW, MOF, MOF, Profile	ATL	23	16	959	0	959	**0**	7	861	**10.22**	0	959	**0**	861	**10.22**	
23	MMD2ATL	Synthesis	normal	KM3	ATL	8	4	1484	0	1484	**0**	8	494	**66.71**	0	1484	**0**	494	**66.71**	
24	Profile2MM	Synthesis	normal	AMW, MOF, MOF, Profile	ATL	30	12	1117	0	1117	**0**	13	799	**28.47**	0	1117	**0**	799	**28.47**	
25	SimpleATLtoATL	Synthesis	normal	AMW, MOF, MOF	ATL	14	8	277	0	277	**0**	1	274	**1.08**	0	277	**0**	274	**1.08**	
	TOTAL Synthesis					**429**	**164**	**16512**	**0**	**16512**		**186**	**9278**		**0**	**16512**		**9278**		
	AVERAGE Synthesis					**20.43**	**7.81**	**786.29**	**0**	**786.29**	**0**	**8.86**	**441.81**	**37.36**	**0**	**786.29**	**0**	**441.81**	**37.36**	**43.81**
26	ATL2BindingDebugger	Modification	refining	ATL	ATL	2	0	41	0	41	**0**	1	24	**41.46**	0	41	**0**	24	**41.46**	
27	ATL2Tracer	Modification	refining	ATL	ATL	2	0	96	0	96	**0**	1	58	**39.58**	1	84	**12.5**	46	**52.08**	
28	ATL2WTracer	Modification	refining	ATL	ATL	8	2	333	2	328	**1.5**	7	166	**50.15**	1	324	**2.7**	152	**54.35**	
29	CACA	Modification	refining	ATL, Trace	ATL, Trace	6	2	178	0	178	**0**	0	178	**0**	3	133	**25.28**	133	**25.28**	
30	CACD	Modification	refining	ATL, Trace	ATL, Trace	6	5	224	1	210	**6.25**	0	224	**0**	3	154	**31.25**	140	**37.5**	
31	CCCR	Modification	refining	ATL, Trace	ATL, Trace	6	2	178	0	178	**0**	0	178	**0**	3	133	**25.28**	133	**25.28**	
32	CFCA	Modification	refining	ATL, Trace	ATL, Trace	6	2	180	0	180	**0**	0	180	**0**	3	130	**27.78**	130	**27.78**	
33	CFCD	Modification	refining	ATL, Trace	ATL, Trace	6	2	122	0	122	**0**	0	122	**0**	3	98	**19.67**	98	**19.67**	
34	CFCP	Modification	refining	ATL, Trace	ATL, Trace	6	2	178	0	178	**0**	0	178	**0**	3	131	**26.4**	131	**26.4**	
35	MergeHOT	Composition	normal	ATL,ATL	ATL	11	3	185	2	175	**5.41**	0	185	**0**	2	70	**62.16**	60	**67.57**	
36	MMTtoMT	Modification	normal	ATL, MOF	ATL	4	0	260	0	260	**0**	4	93	**64.23**	1	249	**4.23**	82	**68.46**	
37	RefactorATL	Modification	refining	ATL	ATL	1	1	37	0	37	**0**	0	37	**0**	0	37	**0**	37	**0**	
38	ROCC	Modification	refining	ATL, Trace	ATL, Trace	6	2	158	0	158	**0**	0	158	**0**	3	110	**30.38**	110	**30.38**	
39	RSCC	Modification	refining	ATL, Trace	ATL, Trace	6	2	179	0	179	**0**	0	179	**0**	3	132	**26.26**	132	**26.26**	
40	RSMA	Modification	refining	ATL, Trace	ATL, Trace	6	5	228	1	214	**6.14**	0	228	**0**	3	147	**35.53**	133	**41.67**	
41	RSMD	Modification	refining	ATL, Trace	ATL, Trace	6	4	171	0	171	**0**	0	171	**0**	3	123	**28.07**	123	**28.07**	
42	Superimpose	Composition	normal	ATL, ATL	ATL, Trace	6	16	267	1	264	**1.12**	0	267	**0**	6	217	**18.73**	214	**19.85**	
	TOTAL Modification + Composition					**94**	**50**	**3015**	**7**	**2969**		**13**	**2626**		**41**	**2313**		**1878**	**34.83**	
	AVERAGE Modification + Composition					**5.53**	**2.94**	**177.35**	**0.41**	**174.65**	**1.2**	**0.76**	**154.47**	**11.5**	**2.41**	**136.06**	**22.13**	**110.47**	**34.83**	**37.71**
	TOTAL					**551**	**268**	**21022**	**28**	**20433**		**199**	**13399**		**41**	**20320**		**12108**		
	AVERAGE					**13.12**	**6.38**	**500.52**	**0.67**	**486.5**	**3.21**	**4.74**	**319.02**	**23.33**	**0.98**	**483.81**	**8.96**	**288.29**	**35.5**	**42.4**

Table 2. Semantic features of HOTs.

classical in the functional programming paradigm (e.g. *map*, *fold*), accepting XSLT templates as arguments. Our work is instead focused on facilitating the development of new, user-defined HOTs in the transformational paradigm.

6 Conclusions and Future Work

HOTs are a relatively novel topic in MDE and specialized tools are not yet available for them. In this paper we show how some modifications to a well-known transformation language could facilitate HOT development by allowing more concise transformations. We support experimentally our proposals, by measuring their impact on a set of real-world HOTs.

Some of the proposals we introduce, namely Concrete Syntax in Output Patterns and Refining Rules, have also a wider contribution scope, since they could have a positive outcome also outside HOT classes.

There are several possible directions for future work. We plan to further exploit the homogeneity of HOT classes by defining transformation patterns and introducing a pattern-based design methodology for them. We want to embed a deeper support of HOTs in the AmmA environment, allowing the user to easily define composite HOT patterns and to execute them seamlessly at compilation time. Finally we plan to generalize our results to the other transformation languages, and create cross-environment tools for HOTs.

References

1. EXSLT. `http://www.exslt.org`, 2006.
2. ATLAS INRIA Research Group. ATL to Problem. `http://www.eclipse.org/m2m/atl/atlTransformations/#ATL2Problem`, 2005.
3. Z. Balogh and D. Varró. Model transformation by example using inductive logic programming. *Software and Systems Modeling*, 8(3):347364, 2009.
4. J Bézivin, F Buttner, M Gogolla, and F Jouault. Model Transformations? Transformation Models! *Lecture Notes in*, Model Driv:440–453, 2006.
5. R. Chenouard and F. Jouault. Automatically Discovering Hidden Transformation Chaining Constraints. In *Model Driven Engineering Languages and Systems: 12th International Conference, Models 2009, Denver, Co, USA, October 4-9, 2009, Proceedings*, page 92. Springer, 2009.
6. J.S. Cuadrado, J.G. Molina, and M.M. Tortosa. RubyTL: A practical, extensible transformation language. *Lecture Notes in Computer Science*, 4066:158, 2006.
7. K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. *Evolution*, 2:c2, 2009.
8. Apache Group. The Apache ANT Project. `http://ant.apache.org/`, 2010.
9. ATLAS INRIA Research Group. KM3 to ATL Copier. `http://www.eclipse.org/m2m/atl/atlTransformations/#KM32ATLCopier`, 2005.
10. M.E. Iacob, M.W.A. Steen, and L. Heerink. Reusable Model Transformation Patterns. In *Workshop on Models and Model-driven Methods for Enterprise Computing (3M4EC 2008))*, page 1, 2008.

11. F. Jouault, J. Bézivin, and I. Kurtev. TCS:: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, page 254. ACM, 2006.
12. Frédéric Jouault Kelly Garces, Wolfgang Kling. Automatizing the Evaluation of Model Matching Systems. In *Workshop on Matching and Meaning: Automated development, evolution and interpretation of ontologies, in AISB'10 Convention, Leicester, UK ¡submitted¿*, 2010.
13. I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based DSL frameworks. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, page 616, Portland, Oregon, USA, 2006. ACM.
14. D. Novatchev. Higher-order functional programming with XSLT 2.0 and FXSL. In *Extreme Markup Languages*, volume 7. Citeseer, 2006.
15. M. Strommer and M. Wimmer. A framework for model transformation by-example: Concepts and tool support. In *Proceedings of the 46th International Conference on Technology of Object-Oriented Languages and Systems, Zurich, Switzerland*, page 372391. Springer, 2008.
16. M. Tisi and F. Jouault. ATL HOT Library. `http://docatlanmod.emn.fr/HOT-library/`, 2010.
17. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *Proceedings of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA)*, page 1833. Springer, 2009.
18. Patrick Valduriez and Marcos Didonet Del Fabro. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and Systems Modeling*, 8(3):305–324, July 2009.