CORE
Metadata, citation and similar papers at core.ac.uk
Provided by HAL Mines Nantes

Extending ATL for Native UML Profile Support: An Experience Report*

Andrea Randak[1], Salvador Martínez[2], and Manuel Wimmer[1]

[1] Business Informatics Group
Vienna University of Technology, Austria
{randak,wimmer}@big.tuwien.ac.at
[2] AtlanMod
INRIA & École des Mines de Nantes, France
salvador.martinez_perez@inria.fr

Abstract. With the rise of Model-driven Engineering (MDE) the application field of model transformations broadens drastically. Current model transformation languages provide appropriate support for standard MDE scenarios such as model-to-model transformations specified between metamodels. However, for other transformation scenarios often the escape to predefined APIs for handling specific model manipulations is required such as is the case for supporting UML profiles in transformations. Thus, the need arises to extend current transformation languages for natively supporting such additional model manipulations.
In this paper we report on extending ATL for natively supporting UML profiles in transformations. The extension is realized by providing an extended ATL syntax comprising keywords for handling UML profiles which is reduced by a preprocessor based on a Higher-Order Transformation (HOT) again to the standard ATL syntax. In particular, we elaborate on our methodology of extending ATL by presenting the extension process step-by-step as well as reporting on lessons learned. With this experience report we aim at providing design guidelines for extending ATL as well as stimulating the research of providing further extensions for ATL.

Keywords: ATL, UML profiles, Transformation Language Extensions

1 Introduction

Model transformation languages are an integral part of Model-driven Engineering (MDE) [1, 12]. Originally, they have been used as the key concept for bridging design and implementation of software systems. The Atlas Transformation Language (ATL) [4] is currently the de-facto standard transformation language in the Eclipse Modeling Framework[3] (EMF). Transformations written in ATL are specified between Ecore-based metamodels and executed to produce an output

* This work has been partly funded by the FWF under grant P21374-N13.
[3] http://www.eclipse.org/modeling/emf/

model conforming to its corresponding output metamodel from a given input model conforming to its corresponding input metamodel.

With the rise of MDE, also the application field of model transformations broadens drastically. Not only transformations between metamodels for bridging the gap between design and implementation of software systems are needed, but different model management tasks, model formats, metamodeling techniques, and model extension techniques have to be supported. Thus, it is inevitable for transformation languages such as ATL to align to these new challenges. Consider for example UML profiles [2] for tailoring the modeling concepts of the Unified Modeling Language (UML) [10] for specific domains and technologies. Although UML profiles may be used in an ATL transformation by escaping to APIs realized in Java by using imperative ATL code, their application within transformations is not a simple task and leads to verbose code. What is needed to make the use of UML profiles more intuitive and straightforward for the transformation engineer is a native support of UML profiles as is currently available for metamodels. This support may be achieved by extending ATL as is conceptually discussed in [17] by providing an extended ATL syntax. In addition, other transformation scenarios also require transformation language extensions as has been reported in several publications (cf. [3, 9, 13] for concrete extension examples and [15] for a survey).

In this paper we report on the possibilities, challenges, and limitations of extending ATL for natively supporting UML profiles. The extension is called ATL4pros and is realized by providing an extended ATL syntax comprising keywords for handling UML profiles which is reduced by a preprocessor based on a Higher-Order Transformation (HOT) again to the standard ATL syntax. In particular, we elaborate on our methodology of extending ATL by presenting the extension process step-by-step as well as reporting on lessons learned. With this experience report we aim at providing design guidelines for extending ATL as well as stimulating the research of providing further extensions for ATL.

This paper is structured as follows. Section 2 gives a motivating example and the arising challenges for extending ATL. In Section 3 the transformation language extension process is depicted in detail by explaining how to provide the abstract and concrete syntax extensions as well as how to define the operational semantics of the extensions in terms of a HOT for ATL4pros. An overview of lessons learned is given in Section 4. Finally, Section 5 concludes with an outlook on future work.

2 Motivating Example and Challenges

In this section, we motivate the extension of ATL for natively supporting UML profiles by showing a short example transformation using an EJB profile for annotating the output model of the transformation. First the transformation is illustrated in standard ATL code, and subsequently, we show how this code may be made more concise by integrating new language constructs into the standard ATL language. Finally, we elaborate on the challenges one has to face when extending ATL.

2.1 Motivating Example

The concept of UML profiles serves as a lightweight extension mechanism for UML. Arbitrary stereotypes and tagged values can be defined within a UML profile which can be subsequently applied to UML model elements. Using stereotypes within an ATL transformation is feasible, but this leads to verbose transformation code. Listing 1.1 presents an exemplary ATL code snippet for transforming EJB archives into UML models. The workaround for using UML profiles in ATL transformations is based on providing the profile as an additional input model for the transformation as well as making calls to the Java UML2 API for assigning profiles and stereotypes as well as setting tagged values in the imperative *ActionBlock* (cf. line 10 to 20 in Listing 1.1).

Listing 1.1. ATL code excerpt for using UML profiles in standard ATL

```
1   helper def: stereo : uml!Stereotype = OclUndefined;
2
3   rule EJBArchive_2_Model  {
4    from
5     s : EJB!EJBArchive
6    to
7     t : UML!Model (
8      name <- s.name
9     )
10   do {
11     t.applyProfile(profile!Profile.allInstances().asSequence().first());
12     thisModule.stereo <- profile!Stereotype.allInstances()
13      -> any( e | e.name = 'EJBArchive');
14     t.applyStereotype(thisModule.stereo);
15
16     if(not s.version.oclIsUndefined()){
17      t.setValue(thisModule.stereo, 'version',s.version);
18     }
19   }
20  }
```

Line 1 of this code example shows the definition of a helper called *stereo*. As we will see later, this helper is necessary for reusing the currently applied stereotype for setting the different tagged values without retrieving the stereotype from the additional input model again and again. In lines 3 to 9 the mapping between the source element *EJBArchive* and the target element UML *Model* is specified. In line 8 it is shown that the name of the *EJBArchive* can be set directly as the name of the UML *Model*. Lines 10 to 20 comprise the imperative *ActionBlock* of the presented rule, indicated by the keyword **do**. In line 11, the UML profile is applied to the UML *Model* for enabling stereotype applications. The code in lines 12 to 13 queries the stereotype with name *EJBArchive* from the additional input model and is saved in the previously mentioned helper *stereo*. Line 14 is needed for applying the recently saved stereotype on the target element. Setting the tagged value called *version* of the *EJBArchive* stereotype is achieved on lines 16 to 18. As the *setValue* operation on line 17 is a call to the Java UML2 API which would in the absence of a concrete value result in an exception, it is necessary to provide an additional check before calling the operation. Please note that lines 11 and 14 also denote calls to the Java UML2 API using the operations *applyProfile* and *applyStereotype*.

The things that are striking about this verbose ATL code are the following: Profile related model manipulation tasks are implicitly established by means of calling Java operations. In particular, the transformation engineer needs to have knowledge concerning the Java UML2 API in order to correctly invoke operations for applying profiles, applying stereotypes, and for setting tagged values. Furthermore, all the operations have to be defined in the imperative *ActionBlocks*. Even though ATL is a hybrid language supporting declarative and imperative constructs, imperative ATL code should be avoided or at least be reduced to a minimum. With the help of UML profile specific keywords, e.g., a keyword for applying stereotypes, the same ATL transformation may be defined more concisely as shown in Listing 1.2.

In [17] we have outlined three different conceptual approaches how to improve the current UML profile support in ATL. The first approach is based on merging the UML metamodel and the UML profiles into one unified metamodel as well as converting the profiled UML models into models conforming to the unified metamodel. The second approach is using a preprocessor for transforming model transformations specified with an extended ATL syntax to model transformations specified with the standard ATL syntax. Finally, the third approach is defined as a direct extension of the standard ATL syntax and ATL compiler. The pros and cons of these three mentioned approaches have been extensively discussed in [17].

In this paper, we pick up the second approach of [17] and report on the realization of this approach by implementing the ATL version named ATL4pros. With ATL4pros, we aim for an ATL extension which is tailored for the usage of UML profiles within ATL transformations. The benefits of such an extension for handling UML profiles are evident: The number of lines of code can be reduced due to the absence of complex statements for querying profile information from an additional input model representing the UML profile. What helps increasing the readability of the transformation even more is the fact that the transformation engineer can apply UML profiles without using imperative code. The imperative parts that are needed for the transformation to execute are hidden from the engineer completely. Furthermore, ATL4pros eases the UML2 API handling because the engineer no longer has to know about the intricacies of the underlying Java UML2 API. All required statements that trigger an API call are automatically created by the HOT.

Listing 1.2. ATL code excerpt for using UML profiles in ATL4pros

```
1  rule EJBArchive_2_Model  {
2   from
3    s : EJB!EJBArchive
4   to
5    t : UML!Model (
6     name <- x.name
7    ) apply PRO!EJBArchive (
8      version <- s.version
9    )
10 }
```

When looking closer at the transformation code shown in Listing 1.2 (which is equivalent to Listing 1.1), two extensions of the standard ATL syntax have to be introduced by ATL4pros for allowing for a more concise definition:

1. The keyword **apply** (cf. line 7 in Listing 1.2) is incorporated into ATL and can be used for applying a stereotype on a UML model element. Therefore, a new construct has to be added to the existing ATL metamodel. For this construct a reasonable container element has to be identified within the existing ATL language element hierarchy.
2. The tagged values of a stereotype may be set just like normal features (cf. line 8 in Listing 1.2), for avoiding the explicit writing of imperative code. This is achieved by simply reusing the existing *Binding* construct of ATL and embed it into the new context for applying stereotypes.

2.2 Challenges

There is a number of challenges that have to be met when extending ATL for supporting new language features.

Abstract syntax. The abstract syntax is defined by an Ecore-based metamodel which has to be extended with new elements. Since the metamodel is containing a large number of elements, it is crucial to find an appropriate location for the new ones. In addition, existing elements have to be altered in order to make the newly introduced elements usable within the standard transformation context.

Concrete Syntax. The extension of the concrete syntax of ATL is the second challenge. Not only is there the need to define new keywords for newly introduced elements, but also to revise the concrete syntax definitions for already existing elements by inserting references from/to the newly introduced ones.

Operational Semantics. The operational semantics determines how to transform new language features into constructs of the standard ATL language via a HOT. This means for each newly introduced element, rewriting rules have to be given to produce standard ATL code and to eliminate the extended syntax elements.

3 Realizing ATL4pros

This section is dedicated to the ATL extension methodology used for building the ATL4pros extension and describes how the aforementioned challenges are tackled. In particular, we first present the methodology at a glance and subsequently each step is explained in more detail. To make the methodology description more concrete, we exemplify each step by elaborating on the main artifacts of the ATL4pros extension.

3.1 ATL Extension Methodology at a Glance

The general approach for building the ATL4pros extension is as follows. The syntactically extended ATL version ATL4pros is transformed into standard ATL via a HOT. Realizing extensions following this preprocessor approach requires for three successive steps: (i) the abstract as well as (ii) the concrete syntax of standard ATL needs to be extended by new elements, and (iii), an operational semantics needs to be defined for the syntax extensions to determine how the language constructs of the extended ATL version, e.g., ATL4pros, are translated into the standard ATL constructs. Please note that this preprocessor approach is not limited to the presented example but may be applied for other domain-specific extensions as well [15]. This approach leads to an extension process as depicted in Figure 1. Extending the abstract syntax is the first activity which has a direct dependency to the ATL.ecore artifact. As a next step, the concrete syntax has to be modified to reflect the performed extension of the abstract syntax. The concrete syntax of ATL is defined in the ATL.tcs artifact which has to be modified in order to use the new syntax elements in the ATL editor. In the last step, an operational semantics needs to be defined in terms of a HOT which has to be developed from scratch. In the following subsections, each step is explained in more detail.

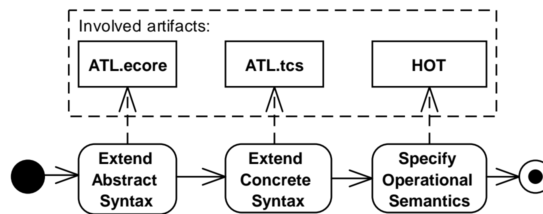

Fig. 1. Extension process and involved artifacts

3.2 Step 1: Extending the abstract syntax

The abstract syntax of the ATL language is provided as an Ecore-based metamodel. Apart from the essential ATL elements, it also contains a vast number of OCL model elements that are needed in a transformation. The location of new elements should fit into the overall structure of ATL transformations to ensure a smooth integration with already existing language elements.

Before we proceed with explaining the ATL4pros extension, it has to be clarified what the term *extension* in our context actually means by stating which actions are valid for producing the extended syntax, i.e., which actions are allowed for modifying the ATL metamodel during the extension process.

- Insertion of new classes: Inserting new classes into the existing metamodel is of most importance. These new classes may have arbitrary features and may inherit from as well as reference to already existing and new classes.
- Extending existing classes: For providing extensions, also existing elements may be extended by adding additional features. In particular, this is necessary for defining the container of newly introduced elements.

The elements needed for the ATL4pros extension are depicted with blue colour in Figure 2 while predefined elements are coloured in black. The main element of the extension is the *ApplyPattern* class which is associated with the *OutPatternElement* class. This assures that each *OutPatternElement* in the **to** block of a transformation rule may have several stereotypes applied, because an *ApplyPattern* may have several *ApplyPatternElements* which stand for the actual stereotype applications. The classes *ApplyPatternElement* and *SimpleApplyPatternElement* are inspired by the hierarchical structure of the classes *OutPattern* and *InPattern* of the standard ATL metamodel. Furthermore, the class *ApplyPatternElement* has a containment reference to the standard *Binding* class. By this, *Bindings* may be reused to define assignments for tagged values similar as assignments for metamodel features are defined. The class *LocatedElement* has a significant role in the ATL metamodel as every other class directly or indirectly inherits from this element. The *LocatedElement* with its location feature gives every subclass the opportunity to have a location within the actual transformation, stating the line number(s) as well as the position within the line(s). All of our introduced elements also have the *LocatedElement* as their superclass in order to maintain the predetermined metamodel structure.

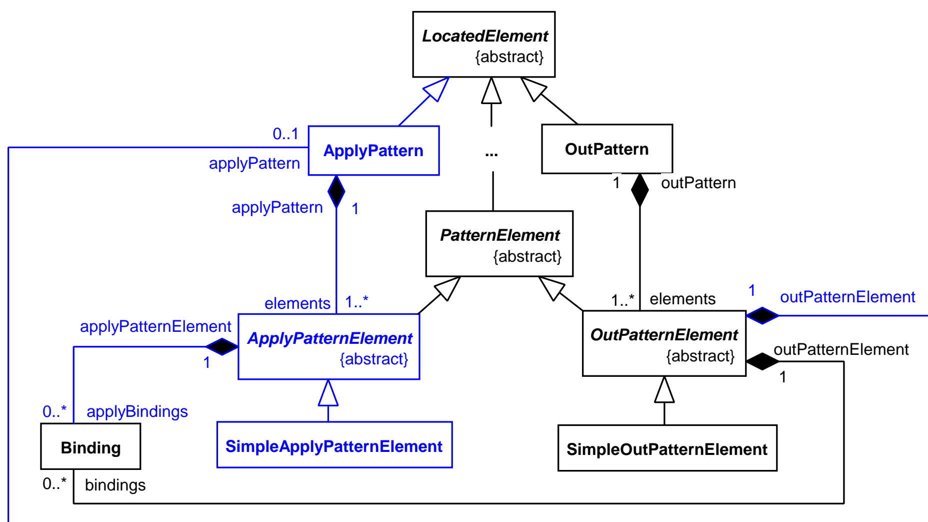

Fig. 2. Excerpt of the ATL metamodel extended by new elements

3.3 Step 2: Extending the concrete syntax

After having introduced the additional abstract syntax elements, the concrete syntax for these elements has to be defined. The concrete syntax of ATL is available as a textual representation, also referred to as Textual Concrete Syntax (TCS) [6]. The TCS builds on the metamodel and consists of templates which define the textual structure of the transformation. Each template corresponds to an element of the metamodel, i.e., for each new element in the abstract syntax, a new template in the concrete syntax has to be inserted. Listing 1.3 shows the implemented templates for the ATL4pros extension as well as an adapted predefined template.

Listing 1.3. Excerpt of TCS definition for ATL4pros

```
1   -- Defining the CS of a new element
2   template ApplyPattern
3    : "apply" [ elements{separator = ","} ] {endNL = false}
4    ;
5
6   template ApplyPatternElement abstract addToContext;
7
8   template SimpleApplyPatternElement
9    : type{separator = ","}
10    -- Reuse of the existing element Binding
11    (isDefined(applyBindings) ?
12     <space> "(" [
13      applyBindings{separator = ","}
14     ] ")"
15    )
16   ;
17
18     -- Enhancing the SimpleOutPatternElement to comprise ApplyPatterns
19  template SimpleOutPatternElement
20   : varName ":" type
21    ...
22    -- Embedding a new element into an existing element
23    (isDefined(applyPattern) ? applyPattern)
24   ;
```

The *ApplyPattern* template on lines 2 to 4 determines that the extension for applying a stereotype to an *OutPatternElement* of a transformation rule has to start with the keyword **apply**. This keyword may subsequently be followed by comma-separated *elements*, whereby these elements refer to the *ApplyPatternElement* of the extended metamodel (cf. Figure 2). The *type* (see line 9) of the *SimpleApplyPatternElement* template is representing the stereotype to be applied, e.g., *EJBArchive* in our example. Moreover, on lines 11 to 15 the template concerning the *SimpleApplyPatternElement* states that there may be an arbitrary amount of *Bindings* defined for setting the tagged values of the stereotype. Please note that we are reusing the abstract and concrete syntax defined for *Bindings* for setting tagged values which is simply possible by copy-and-paste of fragments of standard ATL TCS definition. Finally, the *SimpleOutPatternElement* template is extended in order to posses *ApplyPatterns* (see line 23) so to say to define the anchor for *ApplyPatterns* in the textual concrete syntax.

3.4 Step 3: Defining the Operational Semantics

Overview. A transformation defined using an extended syntax cannot be processed by the existing compiler and virtual machine without modifying these components to support the extended syntax also in the runtime. This would require for a heavyweight extension impacting practically all ATL runtime components which would lead to a separated runtime which has to be separately maintained from the standard runtime. However, in our case, the desired behaviour can be, although in a far more verbose way as shown by the motivating example (cf. Listing 1.1), addressed using the standard ATL syntax. Therefore, we can implement a more lightweight extension mechanism just by preprocessing the transformations defined in the extended syntax to produce standard ATL code.

As ATL transformations are themselves models, they may be preprocessed by using HOTs. The requirements of such a HOT for our running example are as follows:

1. If there is an occurrence of the newly introduced *ApplyPattern* element in a given rule, an *ActionBlock* has to be generated if there is not already one, and subsequently, inside this *ActionBlock*, the statements for applying the stereotype as well as setting the tagged values have to be generated (cf. Listing 1.4).
2. For the rule matching the UML element *Model*, generate an *ActionBlock* if there is not already one, and add the statement for applying the UML profile as is shown in Listing 1.5.
3. The *ApplyPattern* elements have to be removed from the transformation in order to fulfill the standard ATL grammar.

Listing 1.4. Statements to be added to the ActionBlock

```
1  thisModule.stereo <- profile!Stereotype.allInstances()
2     -> any( e | e.name = 'EJBArchive');
3  t.applyStereotype(thisModule.stereo);
```

Listing 1.5. ApplyProfile statement

```
1
2  t.applyProfile(profile!Profile.allInstances().asSequence().first());
```

Implementing HOTs with ATL Refining Mode. As we can see in the example transformations, most of the model representing our original transformation will remain without changes. However, in a model-to-model transformation this won't be an advantage because we would have to create rules not only to apply the desired changes but to copy all the unmodified elements. As the metamodel of ATL is rather complex, the task of creating the copy rules is quite time consuming and even worse, it is error prone. Instead, this is the typical scenario where using the ATL refining mode [16] is appealing.

A transformation in refining mode will be performed in-place. This means that the changes will be directly applied to the input model for producing the target model incrementally. Using this mode, we only have to write rules for the elements that are actually changing. All the model elements not matched by the transformation rules will be kept as they are without the necessity of copying them. It is important to note that to avoid rule interaction problems, a transformation in refining mode is performed in two steps. The changes promoted by the rules are calculated in a first step without modifying the input model and applied afterwards as a second step.

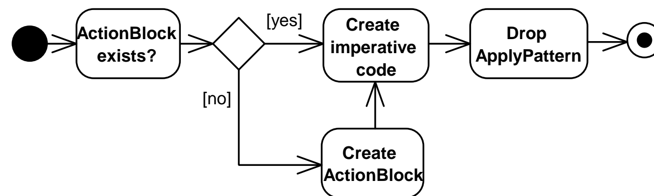

Fig. 3. Rewriting process for transformation rules with ApplyPatterns

From ATL4pros to Standard ATL. Due to the mentioned advantages, the preprocessor HOT has been implemented in refining mode. To simplify this transformation, we have decided to split it in several steps. In the first step, all the required *ActionBlocks* will be created whereas in the second step, these *ActionBlocks* will be matched and filled with the proper statements. In the third and last step, the *ApplyPattern* elements will be eliminated.

In Figure 3 an overview of the process followed by the preprocessor is illustrated and Listing 1.6 shows the rule for creating the *applyStereotype* statement (cf. line 3 in Listing 1.4) that is added to the *ActionBlock* element as well as the rule for eliminating the *ApplyPattern*. Please note that deleting the *ApplyPattern* also ensures the deletion of all contained elements such as the *SimpleApplyPatternElements*.

Listing 1.6. Excerpt of the ATL4pros2ATL HOT

```
1   -- Add stereotype application to ActionBlock
2   rule CreateStereotypeApplication {
3    from
4     s : ATL!SimpleApplyPatternElement
5    to
6     t : ATL!SimpleApplyPatternElement,
7          ...
8     expressionStat : ATL!ExpressionStat (
9      expression <- applySt
10    ),
11    applySt : ATL!OperationCallExp (
12     operationName <- 'applyStereotype',
13     source <- varExpression,
14     arguments <- Sequence { navStereo }
```

```
15     ),
16     varExpression : ATL!VariableExp (
17      appliedProperty <- applySt,
18      referredVariable <- s.applyPattern.outPatternElement
19     ),
20     navStereo : ATL!NavigationOrAttributeCallExp (
21      name <- 'stereo',
22      parentOperation <- applySt,
23      source <- variableExp
24     ),
25     variableExp : ATL!VariableExp (
26      appliedProperty <- navStereo,
27      referredVariable <- varDecl
28     ),
29     varDecl : ATL!VariableDeclaration (
30      varName <- 'thisModule',
31      variableExp <- variableExp
32     )
33   }
34
35   -- drop ApplyPattern
36   rule deleteApply{
37    from
38     s:ATL!ApplyPattern
39    to
40     drop
41   }
```

4 Lessons Learned

This subsection presents lessons learned from extending ATL for natively supporting UML profiles which may lead to a catalogue of design guidelines for building ATL extensions in the future.

Extension points in the ATL metamodel. For the new metamodel elements it is important to find appropriate places in the ATL metamodel. When investigating the ATL metamodel it becomes clear that the abstract class *LocatedElement* is a natural superclass for all new elements. Thus, every new element should directly or indirectly inherit from this metamodel class. The immediate composition relations as well as the directly associated classes depend on the purpose of the extension and have to be determined by the extension engineer from case to case.

HOTs as in-place transformations. As mentioned in Section 3.4, the implemented HOT is designed as an in-place transformation using the ATL refining mode. As every single ATL code fragment that is needed in the standard ATL language to represent the introduced keywords has to be created by instantiating the corresponding metamodel element this is a quite complex and time-consuming task. Just to give the reader an idea about the transformation size, the entire HOT involves approximately 600 lines of code for supporting only small language extensions. As a consequence, before building the HOT, the aim should be to out-source as much reusable parts from the standard ATL transformations as possible into an ATL library which is just included in the standard ATL transformation. This helps a lot in keeping the HOTs small which has a major impact on the development and maintainability efforts. Finally, further

improvements for developing in-place HOTs may be explored similar to what has been done in [14] for model-to-model HOTs.

Test-driven development. An important feature that needs to be investigated refers to the correctness of the implemented extension. Therefore, an appropriate test suite is required for having a test-driven development of ATL extensions. The workflow for ensuring the correctness of the extension is as follows. The standard ATL transformations which are used for abstracting the ATL extension are executed on sample input models. These transformations consequently create corresponding output models. As a starting point for the correctness test, the transformations expressed in the extended syntax should be transformed to standard ATL transformations based on the implemented HOT. Now we can execute the generated ATL transformations with the same sample input models as before resulting in corresponding output models. Assuming that the extension and the HOT are working correctly, these output models have to be equivalent to the afore generated output models by the initial transformations. This test-driven development also allows for building the extensions in an incremental and iterative process meaning that the additional language features are introduced and tested consecutively. For such a framework, previous work on model transformation testing [7, 8] seems to be an appropriate basis, but which has to be extended for testing HOTs. One important building block may form model transformation orchestration languages [11] for modeling the testing process.

ATL extension framework. While implementing the presented extension of ATL, it became apparent that an integrated ATL extension framework is needed. This is merely due to the fact that the tooling is quite time-consuming. To be more specific, different versions of ATL plugins are required for implementing the extension and a lot of copy-and-paste tasks must be performed. It turns out that a wizard-driven extension would be more appropriate to keep the develop/test cycles short which is planned as subject for future work. By such an extension framework we are sure that more researchers would start experimenting with introducing new language features in ATL and providing domain-specific preprocessors for different domains. Such experimental implementations of new language features would also provide valuable input for the general evolution of ATL.

5 Conclusion and Future Work

In this paper we have presented our approach for extending ATL for natively supporting UML profiles as well as elaborated on our experiences. The presented approach based on a preprocessor implemented as a HOT allows to reuse the standard ATL editor as well as the complete ATL runtime. However, we have to admit that the main limitation of the approach is that it is not possible to enhance the expressivity of standard ATL. Nevertheless, it is possible to ease the development of ATL transformations which would normally result in verbose transformation code.

As future work we plan to provide debugging capabilities for ATL4pros. The standard ATL already provides a set of such facilities, e.g., including step-by-step transformation execution, running a transformation to the next breakpoint, and introspection of variables. Debugging is supported for the preprocessed transformation, but not for the transformation expressed in UML4pros which is the specification the transformation engineer would prefer to debug. However, the ATL refining mode builds during its transformation process an internal change computation model. This model is used to store all the changes promoted by the matching rules like created elements, deleted elements, and modification on elements, relating the two versions of the user transformation. Thus, we plan to explore the possibility of using this change computation model, that may be serialized after running the HOT as an additional output model by applying techniques as presented in [5, 18] for model-to-model transformations, to enable debugging for transformations written in the extended ATL syntax. In particular, the debugging messages and further state information should be propagated from the standard ATL transformation specification to the ATL4pros transformation specification.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper.

References

1. Bézivin, J.: On the unification power of models. Software and System Modeling 4(2), 171–188 (2005)
2. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An Introduction to UML Profiles. European Journal for the Informatics Professional 5(2), 5–13 (2004)
3. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: A Domain Specific Language for Expressing Model Matching. In: Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM09) (2009)
4. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming 72(1-2), 31 – 39 (2008)
5. Jouault, F.: Loosely coupled traceability for ATL. In: Proceedings of ECMDA-TW'05: European Conference on Model Driven Architecture - Traceability Workshop (2005)
6. Jouault, F., Bézivin, J., Kurtev, I.: 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 (GPCE'06) (2006)
7. Kolovos, D.S., Paige, R.F., Polack, F.A.: Model comparison: a foundation for model composition and model transformation testing. In: Proceedings of the International Workshop on Global Integrated Model Management (GaMMa'06). pp. 13–20. ACM (2006)

8. Lin, Y., Zhang, J., Gray, J.: Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In: Proceedings of the OOPSLA Workshop on Best Practices for Model-Driven Software Development (2004)
9. Muliawan, O.: Extending a model transformation language using higher order transformations. In: Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08) (2008)
10. Object Management Group (OMG). Unified Modeling Language (UML), Superstructure Version 2.3: http://www.omg.org/spec/uml/2.3/superstructure/pdf/
11. Rivera, J.E., Ruiz-Gonzalez, D., Lopez-Romero, F., Bautista, J., Vallecillo, A.: Orchestrating ATL Model Transformations. In: Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL'09) (2009)
12. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer 39(2), 25–31 (2006)
13. Sijtema, M.: Introducing Variability Rules in ATL for Managing Variability in MDE-based Product Lines. In: Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL'10) (2010)
14. Tisi, M., Cabot, J., Jouault, F.: Improving Higher-Order Transformations Support in ATL. In: Proceedings of the Third International Conference on Theory and Practice of Model Transformations (ICMT'10) (2010)
15. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the Use of Higher-Order Model Transformations. In: ECMDA-FA. pp. 18–33 (2009)
16. Tisi, M., Martínez, S., Jouaoult, F., Cabot, J.: Refining models with rule-based model transformations. Tech. rep., AtlanMod, INRIA & École des Mines de Nantes (2011)
17. Wimmer, M., Seidl, M.: On Using UML Profiles in ATL Transformations. In: Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL'09) (2009)
18. Yie, A., Wagelaar, D.: Advanced Traceability for ATL. In: Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL'09) (2009)