Enabling the reuse of stored model transformations through annotations

Javier Criado[1], Salvador Martínez[2], Luis Iribarne[1] and Jordi Cabot[2]

[1] Applied Computing Group, University of Almeria, Spain
{javi.criado@ual.es,luis.iribarne}@ual.es
[2] AtlanMod Team (Inria, Mines Nantes, LINA) Nantes, France
{salvador.martinez_perez,jordi.cabot}@inria.fr

Abstract. With the increasing adoption of MDE, model transformations, one of its core concepts together with metamodeling, stand out as a valuable asset. Therefore, a mechanism to annotate and store existing model transformations appears as a critical need for their efficient exploitation and reuse. Unfortunately, although several reuse mechanisms have been proposed for software artifacts in general and models in particular, none of them is specially tailored to the domain of model transformations. In order to fill this gap, we present here such a mechanism. Our approach is composed by two elements 1) a new DSL specially conceived for describing model transformations in terms of their functional and non-functional properties 2) a semi-automatic process for annotating and querying (repositories of) model transformations using as criteria the properties of our DSL. We validate the feasibility of our approach through a prototype implementation that integrates our approach in a GitHub repository.

1 Introduction

Model-to-model (M2M) transformations play a key role in Model-Driven Engineering (MDE) by providing the means to automatically derive new modeling artifacts from existing ones. With the increasing adoption of MDE, these model transformations, difficult to produce as they require not only mastering the transformation tools but also domain specific knowledge, become valuable assets. Consequently, M2M transformations should be described, defined, constructed and then stored in the richest possible manner so that the functional and non-functional properties of each of the implemented transformation operations are easier to identify and query. This is a critical requirement for an efficient exploitation and reuse of the model transformations assets (or some parts of them) when facing similar manipulation tasks.

Unfortunately, although some transformation languages and frameworks provide some reuse facilities like inheritance, imports or Higher-Order Transformations (HOTs) [19] (even if largely unused [13]), they lack mechanisms for describing and/or storing information about the inherent properties of model transformations. This makes it difficult to find later the right transformation

for the problem at hand unless we dig into the transformation code ourselves to carefully analyze what it does and how it does it. This is specially true considering there are few public M2M transformation repositories (exceptions would be the ATL model transformation ZOO[3] or ReMoDD [4]).

As an example, a very common transformation use case is the translation from *class diagram* models to *relational* models. Being so popular, anybody requiring a transformation between these two domains should easily find an existing transformation to reuse. Even for the concrete case of ATL, a search for a *class to relational* transformation on the Internet yields thousands of results ranging from very minimal ones to complex versions using inheritance between transformations rules. Nevertheless, each variation implies a different trade-off on the properties of the generated relational model, e.g. different transformation strategies can be followed to deal with inheritance (see Figure 1). While the first strategy could be better for space optimization requirements, the second and third versions improve the maintainability in different degrees. Therefore, beyond its functionality, specific requirements for the task at hand (e.g. having the goal of space optimization) must be considered when choosing the transformation.

Therefore, we believe that a mechanism to facilitate the annotation and search of the transformations in a public repository would be an important step forward towards the reuse of model transformations. Once these annotated repositories are available, a user different from the original transformation developer would be able to select and reuse a transformation (or reuse parts of it) based on its requirements or objectives.

In this paper we propose such mechanism. It is composed by two main elements: 1) a Domain-Specific Language (DSL) to describe functional but also non-functional properties of M2M transformations; 2) a process to semi-automatically tag model transformation with information conforming to our DSL and to query repositories storing this annotated transformations. Functional properties can be calculated in many cases through an static analysis of the transformation code but non-functional properties may require subjective quality metrics or manual analysis in order to be determined.

We demonstrate the feasibility of our approach by developing a prototype implementation specially tailored for the ATL [7] transformation language, including a process to store and query transformations annotated with our DSL in a public GitHub repository. However, we would like to remark that this prototype could be easily extended to deal with other similar rule-based transformation languages like QVT [16], ETL [12] or RubyTL [3] as our approach remains language-independent.

The rest of the paper is structured as follows. Section 2 describes our solution approach. In Section 3, our DSL for describing model transformations is detailed while Section 4 defines the process to annotate existing transformations and constitute repositories with rich search capabilities. Section 5 provides details about our prototype implementation and Section 6 discusses related work. Finally, Section 7 presents conclusions and future work.

[3] http://www.eclipse.org/atl/atlTransformations/

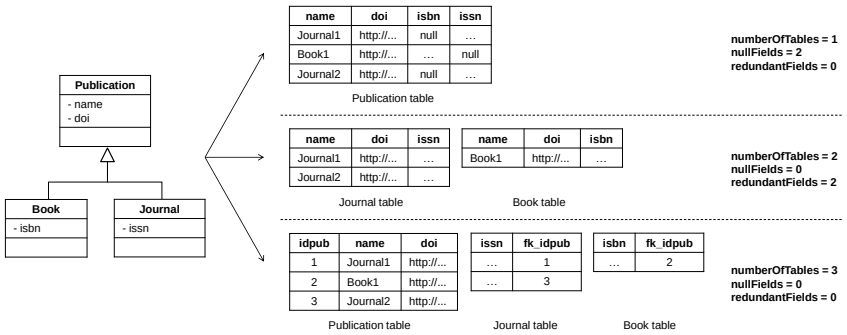

Fig. 1. An example of domain-dependent properties

2 Approach

In order to tackle the aforementioned problems, we propose an approach composed by two main steps (see Figure 2):

1. A Domain-Specific Language for the description of functional and non functional properties of implemented model transformation. This DSL, which will be further detailed in Section 3, is independent from the concrete transformation language. Therefore, it can be used to annotate transformations written in different transformation languages. Along with its abstract syntax, we propose a default catalogue of properties ready-to-use for rule-based model transformations and textual and graphical concrete syntaxes.

2. A semi-automatic process for annotating and reusing existing transformation. This step starts by annotating a given transformation (to be stored) with attributes from a model instance of our proposed DSL. Then, the transformation is stored in a repository of choice. Finally, and transparently to the user, a search engine provides the user with the capability of using the

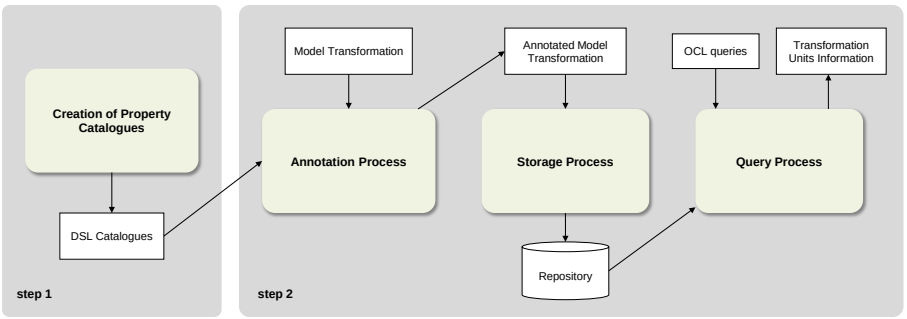

Fig. 2. Annotation and retrieval approach

OCL query language to search for model transformations fulfilling a set of given requirements.

Transformations annotated in this way will allow us to constitute repositories of transformations with rich search capabilities. To demonstrate this extreme we provide a prototype for annotating, storing and querying ATL model transformations and repositories.

3 A DSL to describe model transformations

Explicitly representing the functional and non-functional properties of a transformation helps to identify a suitable transformation (or part of it) for reuse in a given new transformation task.

In order to allow a precise definition of those properties we have developed a new DSL that allows us to describe properties about a *Transformation unit*, i.e., about a `Module`, or about the `Rules` composing it from a predefined `Catalogue` of properties that can be evolved depending on the transformation domain.

In the following we will provide a detailed description of the abstract syntax of our language, a default catalogue to be used for starting annotate rule-based model transformations out of the box and a concrete syntax for 1) visualizing the annotations and 2) integrate them in transformations languages with textual concrete syntax.

3.1 DSL specificaton: Abstract Syntax

The metamodel of our DSL is shown in Figure 3. The main metaclasses are:

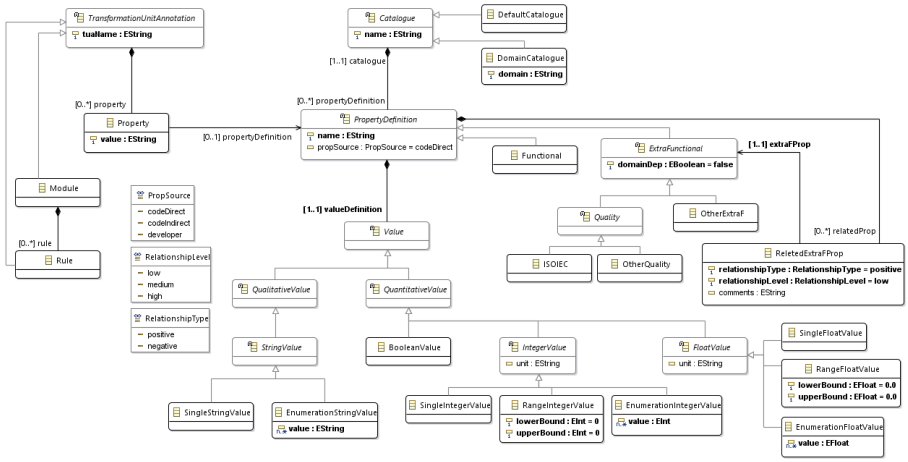

Fig. 3. DSL for describing model transformation properties

Catalogue metaclass: With the aim of giving more flexibility to the property description and instantiation, we propose to define the properties making use of *Catalogues* instead of hardwiring a fixed set of properties in the metamodel itself. Therefore, the DSL allows us to create domain-independent or dependent catalogues. The first type is used to describe the properties common to all domains while the second type is intended to describe the properties that are only relevant to a particular domain.

For instance, in a typical example scenario of transformation from a *Class Diagram* to a *Relational* model, there are different transformation strategies for the inheritance relationship (as we can see in Figure 1). In this sense, it could be useful to define some functional properties specific of this domain, such as the number of tables, the number of "null" fields, or the number of redundant fields generated by the transformation units and some non-functional properties such as maintainability.

Properties: Each catalogue contains a number of *Property* definitions. The *propSource* attribute defines who is responsible for creating the property. A property may be instantiated by an automatic process (calculating its value directly or indirectly from the code), or manually by the developer.

Each property definition is associated with a *Value* definition, which can be qualitative (*QualitativeValue*) or quantitative (*QuantitativeValue*). Qualitative definitions can be a single value or an enumeration of string values. Quantitative definitions can be boolean, integer or float. Integer and float types can be instantiated as a single value, a range of values, or an enumeration of values.

Once we have created the catalogues with the property definitions, we can define annotations for transformation modules or rules. Note that each annotation can be related to a property value, but this value must be established according to the property definition of a catalogue.

Non-functional properties: Our DSL differentiates between *Functional* (e.g. number of input models, number of helpers, or coverage of target metamodels) and *Non-functional* properties.

As an example, Table 1 lists some non-functional properties defined for ATL transformations but any other property could be adapted as well, e.g. "testability" and "installability" can also be added to our DSL. The former could be used to describe if there exist test models associated with a transformation whereas the "installability" quality attribute could be used to identify if the transformation is implemented in a stable version of the transformation language, if the package references are related with integrated URIs or some packages should be registered previously, etc.

Additionally, we classify *non-functional* properties in two different subtypes. *Quality* related properties or *Other* non-functional properties (e.g. developer name, developer affiliation, or last update) information. Note that within the set of possible *Quality* attributes, we also distinguish between ISO/IEC 25000 (e.g. understandability, reusability, or modifiability) properties, i.e., properties

Table 1. Examples of quality attributes (extracted from [21] and [20])

Property	Description
Understandability	Defines how easy or difficult is to comprehend a model transformation. Negative relationship with the number of input models, output models, unused helpers, or elements per output pattern.
Modifiability	Describes how much effort is needed to change a model transformation. Negative relationship with the number of input models, output models, unused helpers, or calls to `resolveTemp()` operations.
Completeness	Indicates if the transformation covers all the elements of the input and output models. Positive relationship with coverage of input/output metamodels. Negative relationship with the number of input/output models, unused helpers, or parameters per called rule.
Consistency	Describes how coherent and stable is the transformation. Positive relationship with coding convention, number of helpers, or calls to `oclIsUndefined()`. Negative relationship with the number of called rules, calls to helpers, or calls to `oclIsTypeOf()`.
Conciseness	Indicates if the transformation is brief and directed to the solution. Positive relationship with number of helpers, rule inheritance, or imported libraries. Negative relationship with the number of input/output models, unused helpers, or the number of called rules.
Reusability	Defines if the transformation or some rules could be reused. Positive relationship with number of helpers or imported libraries. Negative relationship with non-lazy matched rules, called rules, or rules with filter.

defined in the standard, and *Other quality* properties (e.g. stability, reliability of the developer, or level of updating) not belonging to it.

Relation between properties: Functional and Non-functional *properties* can affect the value of related *Non-Functional* properties. In order to represent this relation, a *PropertyDefinition* can contain a collection of *RelatedExtraProperty* definitions describing to which extra-functional *Properties* is related with.

This description can specify a *Type* (positive or negative), a *Level* (low, medium or high), and some *Comments* to this relationship. For example, we can define a functional property named as "ratioOfHelpers" with type "positive" linked (with level "low") to an extra-functional property representing "reusability". This link indicates that this value of the first property has a positive and low effect on the second one. We can also define a extra-functional property named as "understandability" with type "positive" linked (with level "high") to another extra-functional property named as "modifiability", indicating the positive high effect of the first property on the second one.

3.2 DSL specification: Domain-independent Catalogue

In order to facilitate the adoption of our DSL, we provide a ready-to-use default domain-independent catalogue. This catalogue can be imported when creating a new annotation model for a given transformation. Note that although this catalogue is based on properties and metrics defined for the ATL transformation language [20], it can be reused for other transformation languages just by adapting the metric calculation process to each specific case.

In this sense, we provide the following list of functional and non-functional properties for ATL transformations: *number of input models*, *number of output*

models, *number of input patterns*, *ratio of helpers*, *number of calls to resolveTemp*, *ocl expression complexity*, *understandability*, *modifiability*, *reusability*, *completeness*, *performance*, *author* and *last update*. As we mentioned, our DSL allows us to establish the value definition for each property. In addition, we can also represent relations between properties. For example, the value definition of the functional property *numberOfInputModels* has been created as a single integer value, and this property is related to three non-functional properties (*understandability*, *modifiability* and *reusability*) with type *negative* and level *high*.

Note that, as described above, we can also build reusable catalogues for concrete transformation domains. As an example, for the transformation domain of *Class Diagram* to *Relational* models, we have selected three functional properties: *ratio of tables*, *ratio of null fields* and *ratio of redundant fields*; and 2 non-functional properties: *maintainability* and *storage performance*. These properties arise from the three different transformation strategies depicted in Figure 1. For example, the second transformation strategy of Figure 1 has a *medium* value for *ratioOfTables* property, a *low* value for *ratioOfNullFields* and a *high* value for *ratioOfRedundantFields*.

3.3 DSL specification: Concrete Syntax

Our DSL is intended to be used as an annotation language integrated with existing model transformation languages. As in the vast majority of cases, model transformation languages use textual syntaxes as concrete syntax, we propose here a simple textual syntax for our DSL. The grammar of our proposed textual syntax is provided in Listing 1.1.

Basically, our textual syntax allows us to produce annotations that identify the transformation module or rule by name and assign to it couples of properties and the corresponding values (identifying also the catalogue containing the definition of the property as it will help the understandability of the annotation).

As an example, we show an ATL transformation module and a contained rule in Listing 1.2. The rule is annotated with two properties, the ratio of tables functional property with the value of normal (and identifying it as a domain-specific property defined in the catalogue of the Class2Relational domain) and the understandability non-functional property with the value of medium.

Listing 1.1. DSL Grammar

```
Module returns Module:
   rule += Rule*
   '@module' tuaName=EString
   property+=Property*;

Rule returns Rule:
   '@rule' tuaName=EString
   property+=Property*;

Property returns Property:
   '@property' propertyName=EString '=' value=EString
   '(catalogue = ' catalogueName=EString ')';

EString returns ecore::EString:
   STRING | ID;
```

Note that, for simplicity, our annotation language has been integrated in the ATL transformation language by using tags inside comments, which allows us to perform the integration without changing the grammar of the host language. Nevertheless, it would be possible to integrate our annotation language as native tags of the language, which could provide some advantages like syntax highlighting, etc.

Listing 1.2. ATL Class2Relational rule

```
--@module Class2Relational
--@property understandability = medium (catalogue = DefaultCatalogue)
--@property ratioOfTables = normal (catalogue =
    ↪Class2RelationalCatalogue)
--@property reusability = low (catalogue = DefaultCatalogue)
module Class2Relational;
create OUT : Relational from IN : Class;

--@rule Class2Table
--@property understandability = medium (catalogue = DefaultCatalogue)
--@property ratioOfTables = normal (catalogue =
    ↪Class2RelationalCatalogue)
rule Class2Table {
  from
    c : Class!Class
  to
    out : Relational!Table (
      name <- c.name,
      col <- Sequence {key}->union(c.attr->
        select(e | not e.multiValued)),
      key <- Set {key}
    ),
    key : Relational!Column (
      name <- 'objectId',
      type <- thisModule.objectIdType
    )
}
```

Additionally, as graphical information is often easier to grasp at a glance than textual one, we also provide a graphical syntax for our language. In Section 5, we show this concrete graphical syntax.

4 Annotating and Searching model transformations

We describe in this section the process of annotating existing transformation with models conforming to our DSL and the process of then querying already annotated model transformations. This process is summarized in Figure 4. Note that the steps 1.1 to 1.4 depend on the transformation language at hand (although the process for others languages will be similar) while the steps 2 to 3.3 are independent of the language.

4.1 Semi-automatic annotation

The annotation process that we describe here is semi-automatic: (1) functional and non-functional properties that can be derived/extracted from the code itself (including the environment information, like metamodels, etc.) are calculated in

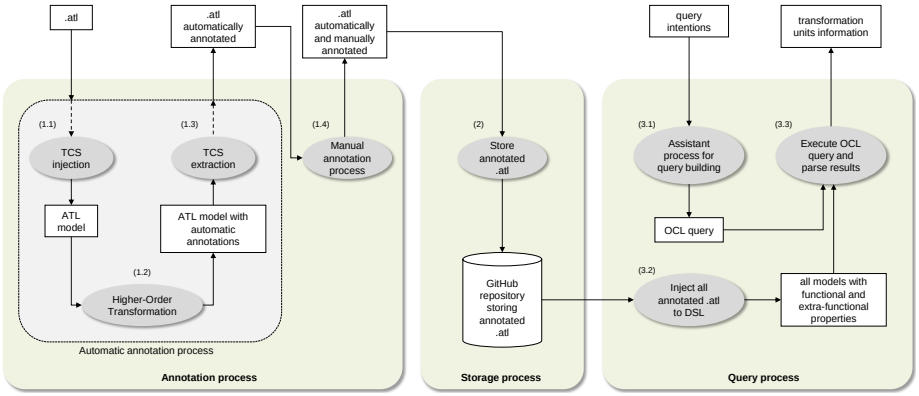

Fig. 4. Process for annotating model transformations and its applications

an automatic way, and (2) properties that need to be evaluated by a developer are filled manually.

In the case of properties that can be directly derived or extracted from the source code of the transformation (including information about input/output metamodels and models, or about the internal structure of each rule) we have chosen to use Higher-Order Transformations. The uniformity and flexibility of the model-driven paradigm allow us to make use of the same transformation infrastructure to develop the model transformation and the annotation process, since model transformations can be translated into transformation models and be given as objects to a different class of model transformations [19]. The calculation of these properties is based on metrics defined in previous work [21, 20].

Note that this process requires having access to the internal structure of the model transformation. Consequently, the concepts of *Module* and *Rule* in our DSL are meant to be linked to the corresponding elements of the metamodel of the transformation language in hand. In the case of ATL, we have linked these concepts to the *Module* and *Rule* concepts of the ATL metamodel so that we are able to inspect all the functional features of the ATL transformation.

Basically, the process of automatically annotating an ATL model transformation follows three steps (see Figure 4): 1.1) injecting the transformation code to a transformation model by using TCS [8]; 1.2) using a HOT transformation to calculate metrics and generate the annotations; 1.3) extracting the transformation model to an ATL transformation with textual syntax by using TCS.

The definition of properties for any given catalogue would follow this process. Here, we have performed it for the properties defined in our default domain-independent catalogue. Some examples are shown in Listing 1.3.

Listing 1.3. automatic calculation of properties

```
helper context ATLMM!Rule def:numberOfInputPatterns():Integer =
   if self.oclIsKindOf(ATLMM!MatchedRule) then
      self.inPattern.elements->size()
```

```
    else
        0
    endif;

helper context ATLMM!Module def:numberOfCallsToResolveTemp():Integer =
    ATLMM!OperationCallExp->allInstances()->select(oce |
    oce.operationName = 'resolveTemp')->size();

helper context ATLMM!OclExpression def:oclExpComplexity():Integer =
    if(self.oclIsTypeOf(ATLMM!OperatorCallExp))then
        self.oclOperatorCallExpComplexity()
    else
        if(self.oclIsTypeOf(ATLMM!IfExp)) then
            self.oclIfExpComplexity()
        else
            if(self.oclIsTypeOf(ATLMM!LoopExp)) then
                self.oclLoopExpComplexity()
            else
                0
            endif
        endif
    endif;
```

Note that, although some non-functional properties can be derived from the functional information, the intervention of a developer is still necessary for fully documenting model transformations. In this sense, a manual annotation process can be performed by using the textual and the graphical concrete syntax, so that a developer can inspect existing properties and add new ones.

4.2 Queries

In this subsection, we show how our DSL annotations enable rich searching. Our main goal is to be able to query the information from the metadata that have been included into the annotated model transformations (step 3.3. of Figure 4). This part of the process is completely independent of the transformation language since it relies only on the property annotations and general information about the transformation.

Querying individual transformations: Given a single transformation, the process of querying it to check its functional and non functional properties requires injecting the textual representation of the transformation into a model corresponding to our DSL (with preimported and loaded instances of the catalogue/s used to annotate the transformation). Once this model is available, standard OCL queries can be used to retrieve the desired information. For example, the query shown below corresponds to an operation performed on a *module* transformation unit (`self` in the code) that lists all the properties with all their values of a specific rule.

```
self.rule->select(r | r.tuaName = 'Rule1').property->
  collect(p | Tuple{name = p.propertyDefinition.name, value = p.value})
```

Querying repositories of transformations: Given a repository of annotated transformations, the process of querying it to retrieve transformation units with specific properties requires: 1) executing the previously described injection step for each transformation in the repository and 2) the construction of an index

model (this index model, which can be considered equivalent to a megamodel, contains links to instance models of our DSL).

Once the index model is available, we can use OCL queries over it in order to make rich searches over the repository (step 3.3 in Figure 4). We can also obtain some information about functional and non-functional properties along with information represented by the transformation models (*e.g.*, metamodels coverage or rule structure). Therefore, many different queries can be performed in order to obtain: rules with a specific value (or value range) of a requested property, modules that have some annotations related to an application domain, catalogue properties which are used more often than others, etc.

For example, the following OCL query could be used for obtaining the transformation units (modules in this example) in the index model(TUAIndex) that thansform UML class diagram models to relational database models.

```
TUAIndex!Index->select(t | t.oclIsTypeOf(tuaproperties::Module))
  ->select(m |m.moduleRef.oclAsType(atl::Module).inModels
  ->exists(inm | inm.metamodel.name = 'ClassDiagram') and
  m.moduleRef.oclAsType(atl::Module).outModels
  ->exists(outm | outm.metamodel.name = 'Relational'))
```

Then, over this collection, it is possible to find which of these selected transformation units have annotations about the *ratioOfTables* property with a *low* value and about the *understandability* property with a *high* value.

```
collection->select(t | t.property
  ->exists(p1, p2 | p1.propertyDefinition.name = 'ratioOfTables'
  and p1.value = 'low'
  and p2.propertyDefinition.name = 'understandability'
  and p2.value = 'high'))
```

Note that a library of frequently used OCL queries can be provided in top of our approach in order to simplify the search tasks of developers. Moreover, once the transformations are integrated in an index model, it would be possible to use other query facilities over it, or use other existing infrastructures for the management of megamodels as MoScript [10].

5 Tool Support

In order to validate the feasibility of our approach, here we describe an Eclipse-based prototype implementation that it includes the creation of textual and graphical editors for our DSL, the adaptation of our DSL to connect it to ATL transformations, the enhancement of the generated *ecore* editor, and facilities for the integration with GitHub and for query execution.

5.1 DSL and editors

The metamodel shown in Section 3 is adapted to the case of ATL in the following way: 1) `Module` elements are linked to the Global Model Management metamodel for ATL [2], in order to store information about input/output metamodels, input/output models, etc. 2) `Rule` elements are connected with the ATL

metamodel to represent the internal structure of each rule (type, input/output patterns, conditions, OCL expressions, etc.).

As discussed in Section 3, we have provided textual and graphical syntaxes for our DSL. Editors for these syntaxes are provided by using the Eclipse Xtext[4] and Sirius[5] tools, respectively.

The default generated editors have been modified to assist in the definition of property and annotation values. This helps to create and visualize together the catalogue property definitions, the property annotations for modules and rules, and the relations between properties and property definitions (see Figure 5). "Recommended" values are automatically represented in green whereas the "not recommended" values are represented in red, and the neutral ones in blue.

Finally, our tool allows the user to define OCL queries to search in the repository for transformations (or rules) based on their functional and non-functional properties.

5.2 Integration in GitHub

In order to facilitate the adoption of our annotation approach, we have decided to use a GitHub project as the repository for annotated model transformations. This way, annotated model transformations will be directly stored in GitHub (step 2 in Figure 4) while a service will be put in place in order to allow the utilization of the metadata. Concretely, we have used the existing Eclipse plugin for "git", which permits the synchronization of the repository with our workspace. Then, from the obtained ATL transformations, we execute an operation in charge of injecting the annotated transformations into the transformation and annota-

[4] https://eclipse.org/Xtext/
[5] http://eclipse.org/sirius/

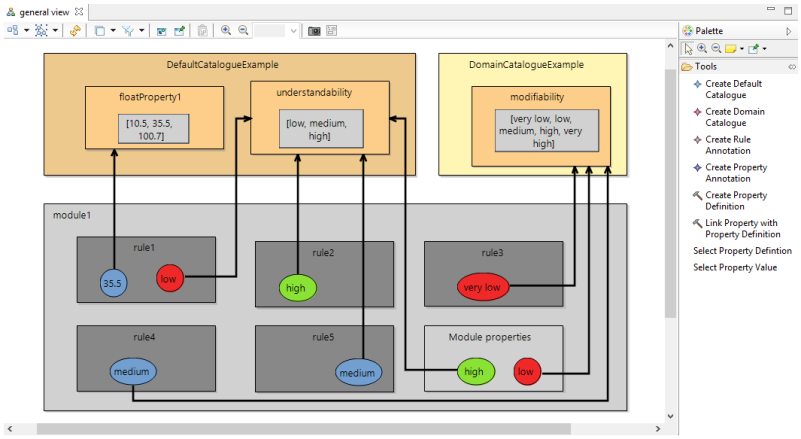

Fig. 5. Snapshot of the graphical editor for our DSL

tion models (Figure 6). The "git" plugin also allows us to upload new annotated transformations, commit modifications or perform update proposals.

Using a GitHub repository for storing the annotated model transformations has some remarkable benefits: (a) it offers a well-known environment that makes very easy to upload or modify transformations, independently of the transformation language, via pull requests; (b) it provides an API to execute basic queries about the stored files, about the contributing users or about other metadata; (c) it gives a tracking system of the problems that may arise in the development of model transformations (through the use of "issues"); (d) it includes the possibility of reviewing the code by adding annotations anywhere in the transformation files; (e) it offers a display of the branches to check the progress and versions of model transformations; and (f) it gathers a lot of information about each user's participation in the development and improvement of the transformations.

However, this kind of repository has some shortcomings. Our repository is intended to store only ATL files, so we must manage the upload operations and limit the repository tracking by using a ".gitignore" configuration file. In addition, GitHub does not implement a specific functionality for managing models or model transformations. Thus, if we want to perform some kind of merge or comparison operation (as our query operations), we have to implement it into a tool or a service outside the repository.

6 Related work

Storing and searching source code of general purpose languages for reuse is a subject largely studied in the software engineering community. Recent contributions include [17] where the authors present a search approach for retrieving code fragments based on code semantics, [14] where the search is specified by using test cases, or [15] focused on the relation between relevant retrieved functions.

Similar to them, our approach allows us to query the repository for appropriate transformation code fragments. However, we follow a different approach. By storing annotated transformations we take advantage of domain-specific knowledge to perform more complex and complete searches.

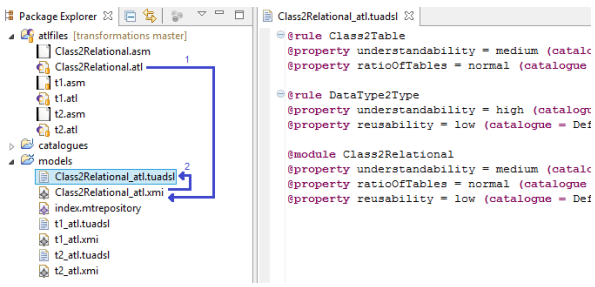

Fig. 6. Inject all annotated transformations from GitHub

Regarding the use of repositories for storing model transformations, most of the existing approaches are focused on the management and storage of models and usually they only allow the definition and storage of very basic structural metadata. AM3 [2], EMFStore [11], or MORSE [5] just store information about the model structure through metamodel references. Other approaches such as ModelCVS [9] and AMOR [1] extract automatic and predefined data from the metamodels to use it as a knowledge base for querying and merging operations.

As for the description of model transformations, in [18] the authors present an extension of the QVT-r language which is able to express alternatives (and their impact on non-functional properties) in the design of transformations. The concept of quality-driven model transformations is also addressed in [6] where design guides are proposed to define model transformations with "alternatives" based on non-functional properties. Our approach applies these ideas to the problem of model transformation reuse where the alternatives can come from different independent sources.

7 Conclusions and future work

We have presented a new DSL specially conceived for describing existing model transformation in terms of their functional and non-functional properties. This DSL along with a semi-automatic annotation process facilitates the reusability of model transformations by enabling the capability of searching for transformation artifacts fulfilling the requirements of a given developer.

As a future work, we would like to explore how our DSL can be used to search for combinations of transformations that may be chained to solve a transformation problem for which a direct transformation is not available. We also intend to reuse existing algorithms for qualitative analysis in goal-oriented requirements engineering to help choose the best possible transformation when none is a perfect match for the designer's goal. At the tool level, we plan to improve the edition and definition of the annotations including code-completion and syntax compilation features as well.

Acknowledgments

This work was funded by the EU ERDF and the Spanish Ministry of Economy and Competitiveness (MINECO) under Project TIN2013-41576-R, the Spanish Ministry of Education, Culture, and Sport (MECD) under a FPU grant (AP2010-3259), and the Andalusian Regional Government (Spain) under Project P10-TIC-6114.

References

1. K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. Amor–towards adaptable model versioning. In *1st International*

Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS, volume 8, pages 4–50, 2008.
2. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the large and modeling in the small. In *MDAFA'05*, pages 33–46. Springer, 2005.
3. J. S. Cuadrado, J. G. Molina, and M. M. Tortosa. Rubytl: A practical, extensible transformation language. In *ECMDA-FA'06*, pages 158–172. Springer, 2006.
4. R. France, J. Bieman, and B. H. Cheng. Repository for model driven development (remodd). In *Models in Software Engineering*, pages 311–317. Springer, 2007.
5. T. Holmes, U. Zdun, and S. Dustdar. Morse: A model-aware service environment. In *APSCC'09*, pages 470–477. IEEE, 2009.
6. E. Insfran, J. Gonzalez-Huerta, and S. Abrahão. Design guidelines for the development of quality-driven model transformations. In *MODELS'10*, pages 288–302. Springer, 2010.
7. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.
8. F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GCPE'06*, pages 249–254. ACM, 2006.
9. G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer. On models and ontologies - A semantic infrastructure supporting model integration. In *Modellierung 2006, 22.-24. März 2006, Innsbruck, Tirol, Austria, Proceedings*, pages 11–27, 2006.
10. W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. Moscript: A dsl for querying and manipulating model repositories. In *SLE'12*, pages 180–200. Springer, 2012.
11. M. Koegel and J. Helming. EMFStore: a model repository for EMF models. In *ICSE'10*, pages 307–308. ACM, 2010.
12. D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *Theory and practice of model transformations*, pages 46–60. Springer, 2008.
13. A. Kusel, J. Schönböck, M. Wimmer, W. Retschitzegger, W. Schwinger, and G. Kappel. Reality check for model transformation reuse: The atl transformation zoo case study. In *AMT@ MoDELS*, 2013.
14. O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: using test-cases to search and reuse source code. In *ASE'07*, pages 525–526. ACM, 2007.
15. C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *ICSE'11*, pages 111–120. IEEE, 2011.
16. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011.
17. S. P. Reiss. Semantics-based code search. In *ICSE'09*, pages 243–253. IEEE, 2009.
18. A. Solberg, J. Oldevik, and J. Ø. Aagedal. A framework for qos-aware model transformation, using a pattern-based approach. In *CoopIS, DOA, and ODBASE*, pages 1190–1207. Springer, 2004.
19. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In *ECMDA-FA'09*, pages 18–33. Springer, 2009.
20. M. F. van Amstel and M. van den Brand. Using metrics for assessing the quality of atl model transformations. In *MtATL'11*, volume 742, pages 20–34, 2011.
21. A. Vignaga. Measuring atl transformations. Technical report, Technical report, MaTE. Department of Computer Science, Universidad de Chile, 2009.