Measuring ATL Transformations[*]

Andrés Vignaga

MaTE, Department of Computer Science, Universidad de Chile

avignaga@dcc.uchile.cl

Abstract

Model transformations are a key element in Model Driven Engineering because they are the primary means for model manipulation. Assessing the quality of model transformations enables the improvement of those assets, and in consequence affects the quality of the MDE-based process in which they are applied. In this work we address the implementation of metrics for model transformations. ATL is currently one of the most popular model transformation languages and thus we focus on measuring transformations defined in ATL. Our solution is implemented as a number of (higher-order) ATL transformations. The complete definition of ATL transformations may involve a variable number of libraries, for this reason we realized our approach by a family of measuring transformations, each of which is automatically producer by a generator. This use case posed a challenge from a typing point of view, which we discuss in detail, for which we propose a mechanism for addressing it.

1 Introduction

Model transformations are a primary means for manipulating models and are thus a key element in Model Driven Engineering (MDE). The quality of transformations directly affects the quality of MDE-based processes which rely on them. The ability of assessing the quality of model transformations enables the improvement not only of individual transformations, but also of the processes in which they are applied.

The quality of model transformations may be assessed by computing metrics on them and analyzing the resulting measures. Concrete metrics for measuring transformations defined in different languages were proposed elsewhere, such as in [4] for the ASF+SDF system. In this work we discuss the implementation of metrics for model transformations, focusing on those proposed for ATL in [6]. We implement metrics by means of an ATL (higher-order) transformation, which accepts as its source the transformation to be measured, then computes from it the metrics according to their definition, and finally produces a model containing the obtained measures.

Quality issues related to ATL transformations were addressed in the *ATL2Problem* use case [1]. *ATL2Problem* is an ATL transformation that checks a set of non-structural constraints, such as rule name uniqueness, on the definition of other ATL transformations. Although the problems reported by this transformation, which represent constraint violations, may be regarded as some form of metrics, they focus on other aspects of ATL transformations compared to those in [6]. Model transformations have already been used for measuring software. ATL transformations were used for measuring model repositories [5], and QVT transformations were used for measuring different aspects of information systems [3].

[*]This work was carried out in collaboration with the AtlanMod team (INRIA and EMN), and was funded by CONICYT Chile and Universidad de Chile.

In this work we address the *complete* definition of ATL transformations, according to the terms introduced in [6]. The complete definition of an ATL transformation consists of the module which includes the transformation rules and the transitive closure of imported libraries (i.e., libraries which are imported by the module and/or among them). Since modules and libraries are separate models, the implication of such a decision is that the number of source models of the measuring transformation is variable. Additionally, the definition of ATL transformations (the measuring transformation we present in this work is a particular case) require an explicit enumeration of both source and target models. As a consequence, it is not possible to measure the complete definition of *any* ATL transformation using a single measuring transformation. As discussed next, we shall define a family of measuring transformations, where each member handles a specific number of source models.

In this document we present a number of ATL transformations which realizes the scheme discussed above. Based on a "template" transformation we use another transformation for automatically generating the members of the family of measuring transformations. Each member produces a model of measures which can be further transformed to other representations, such as tables or charts. Our results may be used for applying the metrics defined in [6] in practical contexts. Furthermore, other metrics may easily be incorporated to our infrastructure as well.

The rest of this work is structured as follows. Section 2 provides an overview of the proposed solution. Section 3 describes the metamodels involved in the transformations. The details of the most important components are presented in Sect. 4. Section 5 discusses some issues concerning the typing of the transformations. Section 6 refers to the prototypical implementation we developed. Section 6 concludes.

2 Overview

Our tool for measuring ATL transformations is expected to handle complete transformation definitions. The complete definition of an arbitrary ATL transformation is composed of a mandatory module and a possibly empty set of libraries. As these assets are separate models, the number of the involved models is positive however arbitrary. This constitutes a problem. We chose to realize the measuring process through an ATL transformation, and since both source and target models must be statically enumerated in ATL transformations, it is not possible to define one single transformation which is capable of measuring an arbitrary transformation. As an example, let us consider two ATL transformations from [1]: *Class to Relational* and *Models Measurement*. For the first transformation, its complete definition is composed of module *Class2Relational* (of type *Class→Relational*) and library *strings* (of type $Lib(ATL)$). Therefore, for the type of the measuring transformation would be:

$$\forall A,B\colon Type;\ C\colon ReferenceModel.\ (A\to B)\times Lib(C) \to Measure$$

where parameters A, B and C should be appropriately instantiated with types *Class*, *Relational* and *ATL* respectively when applying the measurer transformation to models *Class2Relational* and *strings*. For the second transformation, the complete definition is composed of module *KM32Measure* (of type *KM3→Measure*) and libraries *EMOOSE4KM3*, *FLAME4KM3*, *MOOD4KM3* and *QMOOD4KM3* (which are all typed by $Lib(KM3)$). In this case, the type of the measuring transformation would be:

$$\begin{aligned}&\forall A,B\colon Type; C_1,C_2,C_3,C_4\colon ReferenceModel.\\ &\qquad (A\to B)\times Lib(C_1)\times Lib(C_2)\times Lib(C_3)\times Lib(C_4) \to Measure\end{aligned}$$

where parameters A and B should be instantiated with types *KM3* and *Measure* re-

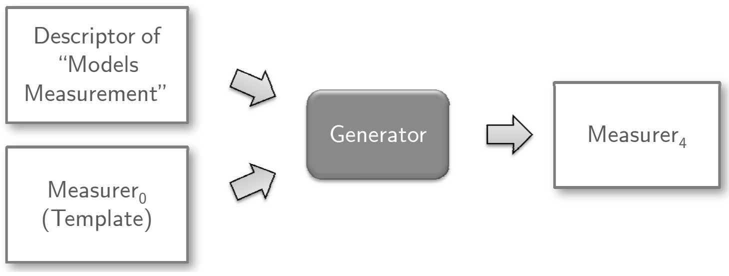

Figure 1: Automatic generation of a measurer transformation from a transformation descriptor and the template

spectively, and parameters C_1 through C_4 with type *KM3*, when applying the measurer transformation to models *KM32Measure*, *EMOOSE4KM3*, *FLAME4KM3*, *MOOD4KM3* and *QMOOD4KM3*. The type of the measuring transformation depends on the number of models involved in the complete definition of the transformation to be measured, and specifically, on the number of libraries. We define a family of measuring transformations of type:

$$\begin{aligned}
&\mathit{Measurer}_0 : \forall A,B\!:\!\mathit{Type}.\ (A\!\rightarrow\! B) \rightarrow \mathit{Measure}\\
&\vdots\\
&\mathit{Measurer}_i : \forall A,B\!:\!\mathit{Type};C_1,\ldots,C_i\!:\!\mathit{ReferenceModel}.\\
&\qquad\qquad (A\!\rightarrow\! B)\!\times\!\mathit{Lib}(C_1)\!\times\ldots\times\!\mathit{Lib}(C_i) \rightarrow \mathit{Measure} \qquad (i\!>\!0)
\end{aligned}$$

In this way, considering the class of transformations which are completely defined by a module only (i.e., they involve no libraries), then $\mathit{Measurer}_0$ is a measuring transformation which is capable of measuring any transformation of that class. Analogously, $\mathit{Measurer}_i$ measures any transformation involving exactly i libraries, for all $i>0$. Note that the operation of each measuring transformation is essentially the same. In fact, every variant may seamlessly manipulate two elements: (a) the module and (b) a collection of libraries, which will be empty for $\mathit{Measurer}_0$. Then the actual difference among all measuring transformations reduces to the part of the ATL header where source models are enumerated, and the code that inserts all source models (except for the module) into the collection of libraries.

The similarities among the measurer transformations may be exploited by a special transformation that inserts specific chunks of code into the code of $\mathit{Measurer}_0$ for generating $\mathit{Measurer}_i$, for any $i>0$. We call such a transformation a *Generator*. We illustrate this idea in Fig. 1 for the case of the transformation *Models Measurement* discussed before. The descriptor model contains information about the complete definition of the transformation. Its metamodel is discussed in the next section. Then the *Generator* generates from the template the transformation $\mathit{Measurer}_4$, as *Models Measurement* involves *four* libraries.

Every measuring transformation, either generated or even the template, has access to the module of the source transformation, the collection of the involved libraries, and the collection of libraries imported by a given unit (i.e., the module or any library). With this information, the transformation computes the values for the different metrics. Each metric is implemented as a separate helper.

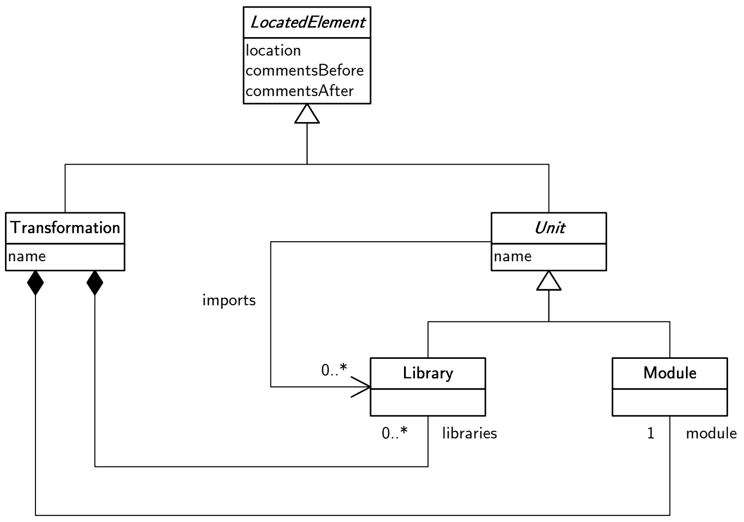

Figure 2: *TransformationDescriptor* metamodel

3 Metamodels

In this section we discuss the metamodels which are involved in all the transformations that embodies our solution. For the case of *Generator*, sources conform to *TransformationDescriptor* and *ATL*, while the target conforms to *ATL*. For the case of the measuring transformations, the sources are *ATL* models (one module and zero or more libraries), and the target conforms to *Measure*. For processing the measures, the definition of the associated metrics is required. Such information is present in models conforming to *Metric*.

3.1 Transformation Descriptor

A transformation descriptor model describes the dependencies among the units that compose a complete transformation. The *TransformationDescriptor* metamodel is illustrated in Fig. 2. A complete transformation is composed of one module and many libraries. Since both modules and libraries are units, they in turn may import many libraries.

The *Generator* transformation just counts the number of libraries that compose the described transformation and uses that value for inserting the specific code the template requires for becoming a proper measuring transformation. More concretely, that value is the index of such a transformation. For the case of *Models Measurement* transformation, that value is 4.

3.2 ATL

The *ATL* metamodel used for *Generator* and all measuring transformations is the standard one [2] and will not be further discussed in this document. We refer the reader to [7] for a short introduction to that metamodel.

3.3 Metric

The *Metric* metamodel is illustrated in Fig. 3. It is based on a part of the *Measure* metamodel defined for the *Models Measurement* transformation [1]. Such a metamodel can be understood as the union of the metamodel presented in this section, for representing information about metrics, and the metamodel presented in Sect. 3.4, for representing

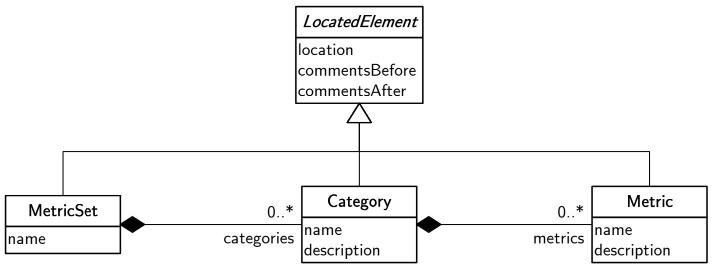

Figure 3: *Metric* metamodel

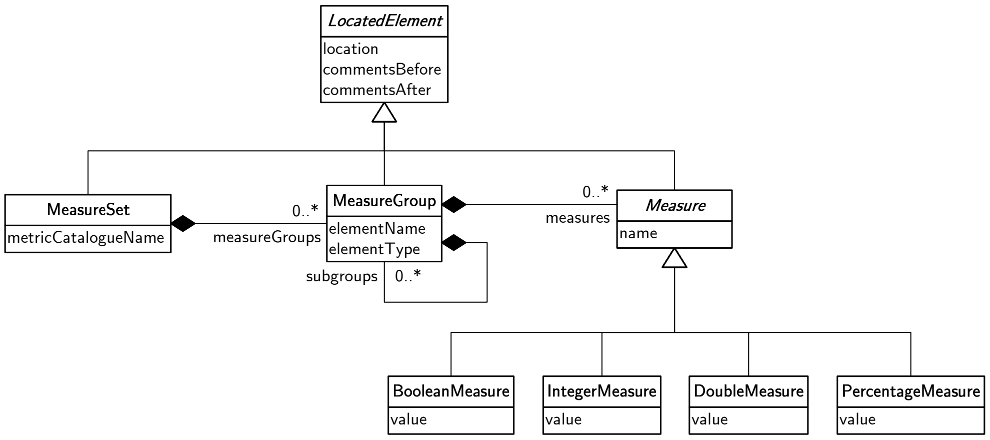

Figure 4: *Measure* metamodel

measures. Using such an "unified" metamodel, the information about the metrics is included in every model of measures. This means that several models of measures repeat the same description of the metrics they refer to. Separating measures from metric definitions in their own metamodels, metrics can be defined once, and that model can be associated to multiple *Measure* models. Additionally, this approach does not couple the metrics definition to the existence of measures for them.

A metric set has a name (e.g., "Metrics for ATL Transformations"), and a set of metric categories. A category has a name too, a description and a set of metrics. A metric in turn also has a name and a description. A category is used to group metrics. For example, it may contain all metrics which apply to the same element (e.g. a Rule, a Helper, etc.).

3.4 Measure

The *Measure* metamodel is illustrated in Fig. 4. A measure represents the value computed for a metric. The root of this metamodel is a measure set which indicates the name of the catalogue containing the definition of the associated metrics (typically, the value of this attribute is the value of attribute name of class MetricSet in Fig. 3). A measure set also has a set of groups of measures. Each group contains measures of some element; therefore the group registers the name and the type of such an element. Measurable elements may be composite, for that reason groups may have subgroups. A measure has only a value, which may be of a variety of types (Boolean, Integer, and so on). That value is meaningless on its own. For that reason, the measure registers the name of the associated metric. Such a name must match the value of the name attribute (which is unique) of an instance of class Metric within a model conforming to *Metric*. In this way, the *Measure* metamodel has a dependency on the *Metric* metamodel. Such a dependency is implemented by attribute values, which is the same mechanism by which an ATL Unit registers the dependencies on Libraries, through the value of attribute name of LibraryRef class. However, a weaving model could also be produced along with the measure model, establishing the required links between the appropriate measure and metric elements.

4 Measuring Transformations

In this section we discuss in detail our approach to measuring complete ATL transformations using ATL transformations. We start by describing the main ideas behind the template used for generating the actual measuring transformations. We then present the details of the *Generator* transformation which transforms such a template to concrete ATL measurers.

4.1 Template

Measuring transformations are produced by inserting specific pieces of code into specific locations of a template transformation. In what follows we describe the structure of such a template and the operation of measuring transformations during the measurement process.

4.1.1 Design of the Template

The template we use for generating the different measuring transformations is actually a fully functional measuring transformation itself. On the one hand, it is capable of measuring transformations composed of just one module, and for that reason it is the implementation of the $Measurer_0$ transformation. On the other hand, it is defined in a way that enables the insertion of specific code which turns it into some $Measurer_i$, for some i>0. The details of such a process are presented in the next subsection. The ATL header of the template is therefore:

```
create OUT : Measure from ModIN : ATL;
```

which indicates that the template produces a Measure model OUT from an ATL model (i.e., the module) ModIN. The template also defines two utility helpers; one attribute and one function. The attribute is defined in the context of the module implementing the template. It is used for accessing the complete set of libraries involved in the transformation to be measured. Its name is libraries and is defined as follows:

```
helper def: libraries : Set(ATL!Library) =
    Sequence{library names}->
        iterate(e; res : Set(ATL!Library) = Set{} |
            res.including(ATL!Library.allInstancesFrom(e)->asSequence()->first())
        )
;
```

This helper essentially converts a sequence of library names to a set of libraries. In such a definition, *library names* is a collection of strings where each name is the name of a formal parameter corresponding to a library. In the case where the template is considered as a measuring transformation, such a collection is empty, since it is assumed that no library is involved, and the attribute returns an empty set. In other cases, the resulting set contains all libraries received as source models. Note that ModIn only provides access to the names of the libraries imported by it and thus libraries is required. Additionally, helper getLibraries() may be applied on any ATL unit (i.e., a module or a library) and returns the (possibly empty) set of libraries imported by that unit. It is defined in terms of libraries as follows:

```
helper context ATL!Unit def: getLibraries() : Set(ATL!Library) =
    let myLibs : Set(String) = self.libraries->collect(n | n.name)->asSet() in
        thisModule.libraries->select(l | myLibs->includes(l.name))
;
```

Each metric is implemented as a separate helper function. Such code is not affected by the *Generator*, and as such is shared by all measuring transformations. Furthermore, we exploit the categorization of metrics used in [6] for determining the context of each helper. As an example, metric Total number of Imported Libraries (TIL) counts the total number of imported libraries, either directly or indirectly, by an ATL unit. Such a metric is applicable to any ATL unit and therefore the context of the helper that implements it is ATL!Unit. We implemented such a metric recursively as follows:

```
helper context ATL!Unit def: computeMetric_TIL() : Integer =
    let libs : Set(ATL!Library) = self.getLibraries() in
        let s : Integer = libs->size() in
            if s = 0 then
                s
            else
                s + libs->iterate(e; res : Integer = 0 | res + e.computeMetric_TIL())
            endif
;
```

This definition uses the getLibraries() helper defined above for accessing the libraries imported from self. In addition, the type of the result indicates which variant of Measure, from Fig. 4, should be instantiated for storing the resulting measure. In this case the result is an Integer value, thus IntegerMeasure needs to be instantiated.

4.1.2 The Measurement Process

The measurement process is defined as follows. We define a MeasureSet for the complete set of measures computed for a source transformation. Such an element is organized in a number of MeasureGroup elements. Top level groups correspond to the source module and to the source libraries, if any. For example, if a transformation to be measured is defined in a module and n libraries, then the resulting measure model will have n+1 top level measure groups.

The measure group corresponding to the source module comprises a number of measures on the module itself, and a number of subgroups which correspond to the different elements associated to the module. For example, a module may have helpers and rules. Then, for each of these elements, a measure group, which is a subgroup of the group corresponding to the owner module, is generated. Each (sub)group contains the measures obtained from the corresponding helper or rule. In turn, a measure group corresponding to a library analogously comprises measures on the library, and a number of subgroups that correspond to the helpers defined in the library. In turn, each subgroup contains measures obtained from the corresponding helper. Figure 5 illustrates this scheme applied to the *Class to Relational* transformation [1]. Such a transformation is defined by module *Class2Relational* and library *strings*, therefore two top level measure groups were created.

The template defines two main rules, one for creating the measure group for a module (rule Module2MeasureGroup) and the other for creating measure groups for libraries (rule

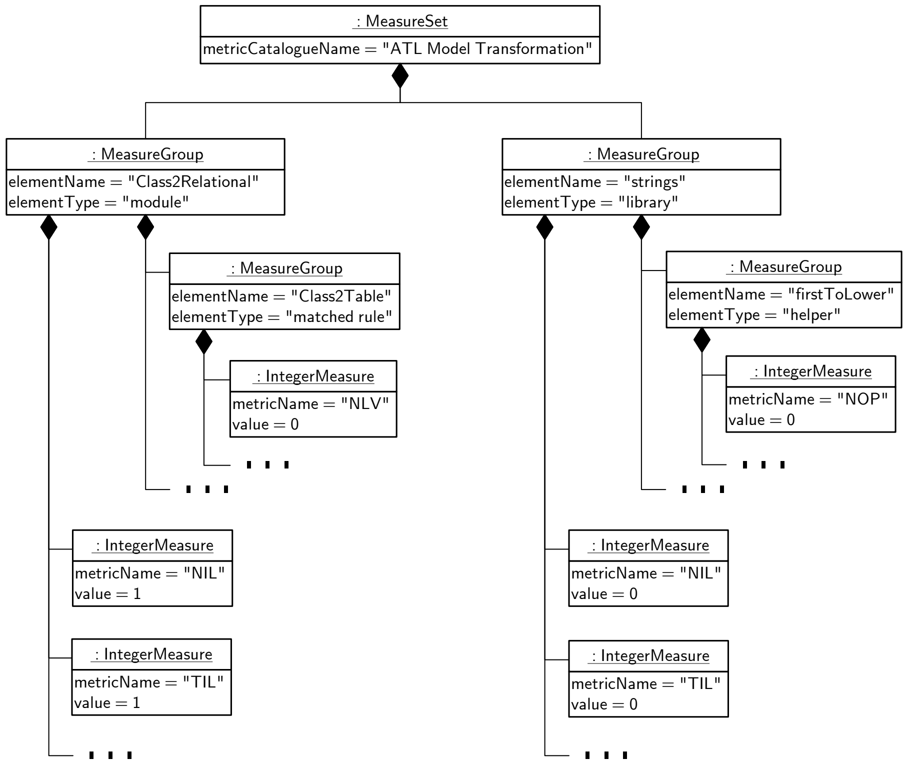

Figure 5: Partial measure model for *Class to Relational*

Library2MeasureGroup). These rules create the corresponding measure group, computes the metrics on the source unit (e.g., measures for metrics NIL and TIL in Fig. 5), and indicates which subgroups need to be created. Other rules include MatchedRule2MeasureGroup and Helper2MeasureGroup. The resulting elements of such rules are in fact included in the target model as subgroups of other groups, as is the case of groups corresponding to elements Class2Table or firstToLower in Fig. 5.

Since any transformation to be measured is assumed to be composed at least by one single module, then rule Module2MeasureGroup is always matched exactly once. In turn, rule Library2MeasureGroup will be matched in $Measurer_i$ exactly i times, for all $i \geq 0$. For this reason, we decided to include the creation of the instance of MeasureSet within the out pattern of Module2MeasureGroup.

As a final remark, the approach described above is implemented using declarative code only. This contrasts with the implementation of the measuring transformation of [5], which even though it is based on a similar measure metamodel and shares a similar purpose, it makes intensive use of called rules and imperative code.

4.2 Generator

The *Generator* transformation transforms the template (i.e., $Measurer_0$) to a $Measurer_i$ transformation, for any $i>0$. The concrete value of i is extracted from the descriptor of the transformation to be measured. The *Generator* produces a transformation whose implementation is essentially the implementation of the template with the inclusion of specific fragments of code in some appropriate locations. In other words, the Generator may be regarded as a "copier" (in the terms of the *KM32ATLCopier* transformation [1]), which interleaves additional code. Such additional code is:

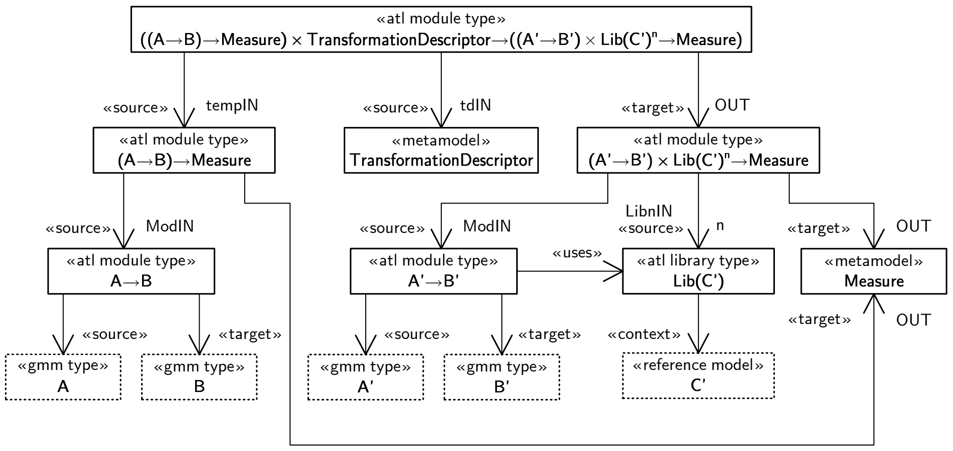

Figure 6: Typing of *Generator* and measuring transformations

(1) As many formal parameters in the header as libraries are involved as source models.

(2) The names of those formal parameters, as strings, in the *library names* area of the definition of the libraries helper discussed in the previous subsection.

As an example, let us consider again the *Models Measurement* transformation, which comprises a module and four libraries. Such a transformation is to measured by *Measurer*$_4$. According to (1), *Generator* produces its header as follows:

```
create OUT:Measure from ModIN:ATL, Lib1IN:ATL, Lib2IN:ATL, Lib3IN:ATL, Lib4IN:ATL;
```

In turn, according to (2), the definition of libraries produced for *Measurer*$_4$ is the following:

```
helper def: libraries : Set(ATL!Library) =
    Sequence{‘Lib1IN’, ‘Lib2IN’, ‘Lib3IN’, ‘Lib4IN’}->
        iterate(e; res : Set(ATL!Library) = Set{} |
            res.including(ATL!Library.allInstancesFrom(e)->asSequence()->first())
        )
;
```

5 Discussion

In this section we discuss some issues concerning the typing of the transformations presented in the previous section. In particular, based on the typing approach presented in [8], we discuss how types are related for building the types of the *Generator* and the measuring transformations. This provides the context for analyzing a limitation in the typing of transformations like the *Generator*. We also show in detail the particular instantiation of this scheme when the *Models Measurement* transformation is measured.

5.1 Typing the Transformations

In Fig. 6 show how types are combined for building a type for the Generator transformation and for the family of measuring transformations. In the lower left hand corner the type of an ATL module can be found. Such a type is built from A and B as source and target respectively, and for that reason it is denoted as A→B. Both A and B are denoted using a dashed box which means that they are parameters to be bound for obtaining a concrete function type. In this sense, type A→B is quantified on both A and B. In particular, stereotypes applied to both parameters indicate that they can be instantiated with *any* type, from a simple metamodel to a more complex function type. Then any function type can be obtained from a proper instantiation of this type, and thus it is the type of an arbitrary ATL transformation. Right above in the figure, another function type can be seen. It is built from the type just discussed as source and Measure metamodel as target. This type specifies a transformation that accepts an arbitrary transformation as source and produces a measure model as a result. This is the type of the template transformation, which is $Measurer_0$, as we discussed in Sect. 2. At the center of Fig. 6 another type of an arbitrary ATL transformation uses parameters A' and B'. Although parameters are local to the type that declares them, we used different parameter names for avoiding name conflicts. Above and to the right, another function type is shown. Such a type indicates that source models are one arbitrary ATL transformation and n ATL libraries defined in the context of C', and the target model is a Measure model. A library typed by Lib(C') is a library that contains helpers defined in the context of some element within C'. Type $(A'\rightarrow B')\times Lib(C')^n \rightarrow$ Measure is a dependent type which depends on n. It actually represents a family of types where each member of the family corresponds to a value of n, for any n>0. Such a type generically types $Measurer_i$, but most importantly, each member of the family types a concrete measuring transformation. Note that $Lib(C)^n$ is a shorthand notation for $Lib(C_1)\times\ldots\times Lib(C_n)$ used in Sect. 2. The association stereotyped by uses indicates that the module uses the libraries. This enables some additional checks. For example, it is expected that the helpers within a library are defined in the context of some elements contained in the source metamodel of the module that uses the library. As a consequence, we could require that C' should be instantiated with the same type as A'. In the case A' is instantiated with a product, C' should be instantiated with a component of A'. In the particular case where the library defines helpers operating on ATL datatypes (e.g., Strings), C' should be instantiated with ATL.

At the top of Fig. 6 the dependent type for the *Generator* transformation is shown. Since this type involves the dependent type just discussed, it also depends on n. The type specifies that the *Generator* accepts the template and a transformation descriptor as sources, and produces a member of the family of measuring transformations. Note that, using classical function type notation, the header of the *Generator* is just $ATL\times TransformationDescriptor\rightarrow ATL$, that is, a transformation is produced from a transformation and a descriptor. This is an underspecification for *Generator*, since it is not true that any transformation is expected as input, and it is not possible to infer a concrete function type for the result. In what follows, we show using an example how this problem is solved in our approach.

5.2 Example

In the remainder of this section we show how our typing approach is applied to a concrete case. Figure 7 shows the types involved in the *Models Measurement* transformation [1]. We have an ATL module of type KM3→Measure, that is, it produces a Measure model from a KM3 model. Such a module also uses four libraries that contain helpers defined in the context of some KM3 metaclass. For that reason, the type of all four libraries is Lib(KM3). Since this transformation involves four libraries n needs to be bound to 4. As a result, the type of the measuring transformation is $(A'\rightarrow B')\times Lib(C'_1)\times Lib(C'_2)\times Lib(C'_3)\times Lib(C'_4) \rightarrow$ Measure, where A' and B' can be bound to any type, and C'_1 to C'_4 can be independently bound to any reference model. Then binding A' to KM3, B' to Measure, and C'_1, C'_2, C'_3 and C'_4 to KM3 in the appropriate part of Fig. 6, the structure of Fig. 7 can be obtained. This means that Measurer4 may be safely applied to the module and the four libraries that compose *Models Measurement*.

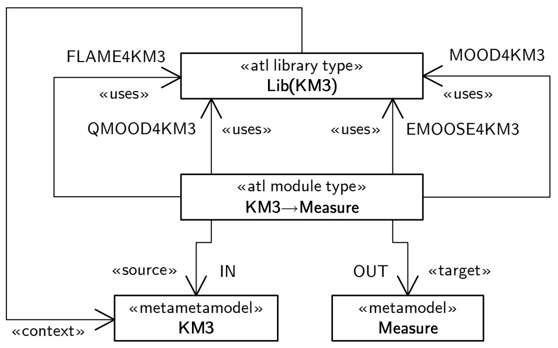

Figure 7: Typing of the complete definition of *Models Measurement* transformation

The complete process is illustrated in a sort of transformation chain in Fig. 8. At the top, Generator is applied to Template and KM32MDesc for producing Measurer4. In this step n is bound to 4 and the resulting type is the one we discussed above. Then, Measurer4 can be applied to module KM32Measure and libraries EMOOSE4KM3, QMOOD4KM3, MOOD4KM3 and FLAME4KM3, since the bindings already discussed allows such an application. Furthermore, we can safely claim that resulting model Measures4KM32M is of type Measure.

6 Implementation

We developed a prototype which implements the transformations described in this work. The *Generator* transformation was fully implemented as a (higher-order) ATL transformation. In turn, the template was partially implemented. Particularly, the template includes the implementation of a subset of the large set of metrics for ATL transformations presented in [6]. An Eclipse workspace containing:

- TransformationDescriptor, ATL, Measure and Metric metamodels,
- Template and Generator ATL source files,
- KM32MDesc terminal model for producing the Measurer4 transformation,
- and KM32Measure, EMOOSE4KM3, QMOOD4KM3, MOOD4KM3 and FLAME4KM3 ATL source files for producing Measures4KM32M terminal model,

is available at `http://mate.dcc.uchile.cl/research/tools/measuring`.

7 Conclusions

In this work we discussed the implementation of metrics for measuring the complete definition of ATL transformations as an ATL measuring transformation. Since the complete definition of an ATL transformation involves a variable number of models, the measuring transformation was replaced by a family of transformation where each member is specialized in measuring ATL transformations composed of a fixed number of models. Each such member may be automatically generated from a template transformation, and a transformation descriptor, through a measurer generator transformation. We successfully prototyped our approach by implementing the required infrastructure and a number of metrics from our base catalogue [6]. This provides a basis for its complete implementation which in turn will enable the applicability of the metrics in a practical context.

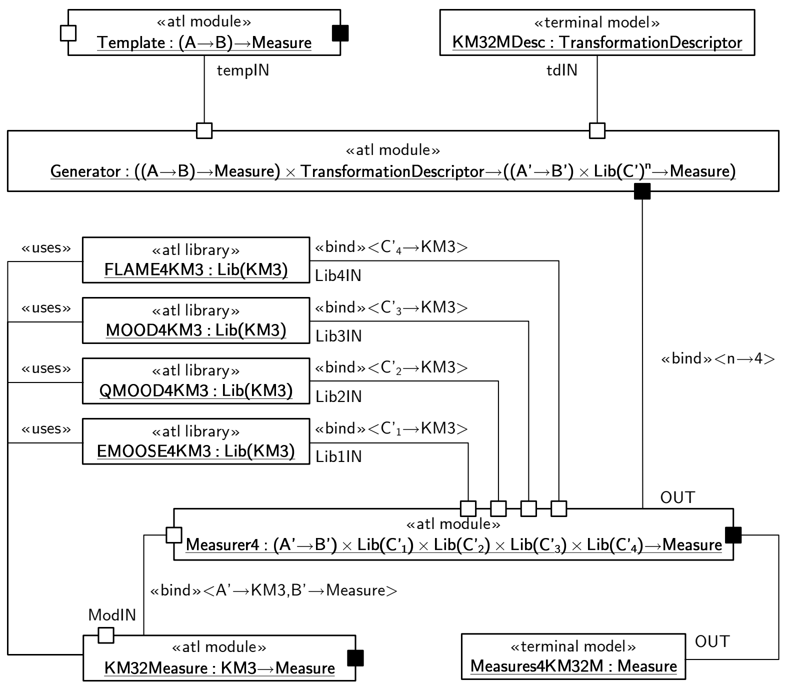

Figure 8: Generation of Measurer4 and its application to the models composing *Models Measurement*

Our approach bases its measure representation on the solution proposed in [5]. However, we separated measures from metrics definition, which avoids redundancy across the produced measure models. Furthermore, unlike the solution proposed in that work for measuring models which makes an intensive use of imperative code, our prototype uses declarative code only.

The ATL use case addressed in this work presented a challenge from a typing point of view. Even providing the type of each measuring transformation separately, the target type of the *Generator* transformation is not actually a single type but rather a family of separate types. Based on the approach presented in [8], we introduced a form of dependent type which enabled us to reasonably represent such a family of types at once.

As future work we plan to extend our prototype with more metrics from [6]. With a complete implementation of such a catalogue detailed experiments can be conducted and the results could be used for enhancing it. This would start an iterative process, where metrics are improved as a result of the experiments enabled by the implementation, and the implementation is updated for reflecting those improvements. Finally, our idea for representing dependent types for handling target transformations with variable signatures could benefit from finding and analyzing additional applicability scenarios.

References

[1] ATL Transformations Zoo. Internet: `http://www.eclipse.org/m2m/atl/atlTransformations/`, 2009.

[2] Atlantic Zoo. Internet: `http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/`, 2009.

[3] B. Mora, F. García, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. A. Carsí, and I. Ramos. Software Measurement by Using QVT Transformations in an MDA Context. In J. Cordeiro and J. Filipe, editors, *ICEIS (1)*, pages 117–124, 2008.

[4] M. F. van Amstel, C. F. J. Lange, and M. G. J. van den Brand. Metrics for Analyzing the Quality of Model Transformations. In G. Falcone, Y.-G. Guéhéneuc, C. F. J. Lange, Z. Porkoláb, and H. A. Sahraoui, editors, *12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2008)*, pages 41–51, Paphos, Cyprus, July 2008.

[5] E. Vépa, J. Bézivin, H. Brunelière, and F. Jouault. Measuring Model Repositories. In *2nd Workshop on Model Size Metrics, co-located with MoDELS 2006 (MSM 2006)*, Genova, Italy, October 2006.

[6] A. Vignaga. Metrics for Measuring ATL Model Transformations. Technical Report TR/DCC-2009-6, Computer Science Department, Universidad de Chile, 2009.

[7] A. Vignaga. Paraphrasing Reference Models and Transformations. Technical Report TR/DCC-2009-3, Computer Science Department, Universidad de Chile, 2009.

[8] A. Vignaga, F. Jouault, M. C. Bastarrica, and H. Brunelière. Typing in Model Management. In R. Paige, editor, *ICMT2009*, to appear.