Rules and Helpers Dependencies in ATL – Technical Report –

Javier Troya[1], Martin Fleck[2], Marouanne Kessentini[3], Manuel Wimmer[2], and Bader Alkhaze[3]

[1] Department of Computer Languages and Systems, Universidad de Sevilla, Spain
jtroya@us.es
[2] Business Informatics Group, Vienna University of Technology, Austria
{fleck,wimmer}@big.tuwien.ac.at
[3] Computer and Information Science Department, University of Michigan, USA
{marouane,bader}@umich.edu

Abstract. In this technical report we explain the different types of dependencies that can take place in ATL model transformations and exemplify them in a running example. Besides, we explain in detail how we use a static types extraction in order to obtain information of the dependencies.

1 Introduction

The content of this document describes in detail some information that is part of our approach presented in [1]. In such paper, a search-based approach for the modularization of model transformations is proposed. The aim of this proposal is to improve the evolution, comprehensibility, maintainability, reusability and overall quality of model transformation programs. To this end, we propose to automatically organize a model transformation in modules in an optimal way according to specific criteria. The application and execution of this approach is guided by our search framework that combines an in-place transformation engine and a search-based algorithm framework. We demonstrate the feasibility of our approach by using ATL as concrete transformation language and NSGA-III as search algorithm to find a trade-off between different well-known conflicting design metrics for the fitness functions to evaluate the generated modularized solutions.

In order to modularize a model transformation in the best possible way, we need to know the dependencies among its artifacts. These dependencies, together with the fitness function, are the ones that actually drive the modularization. Since our prototype modularizes ATL model transformations, we need to know the dependencies among rules and helpers. In this technical report we explain the different types of dependencies than can take place in ATL model transformations and exemplify them with a running example. Furthermore, we explain in detail a static types extraction in ATL transformations, which is key for identifying one of the different types of dependencies.

2 ATLAS Transformation Language

This model transformation language (named *ATL* for short) has come to prominence in the model-driven engineering community due to is flexibility, support of the main meta-modeling standards, usability that relies on strong tool integration with the Eclipse world, and a supportive development community [3,4].

ATL is a textual rule-based model transformation language that provides both declarative and imperative language concepts. It is thus considered a hybrid model transformation language. Both in-place and out-place model transformations can be defined in ATL. The former are defined using the *refining mode*, while for the latter we use the *default mode* [5].

An ATL transformation is composed of a set of transformation rules and helpers. Each rule describes how certain target model elements should be generated from certain source model elements. Typically, ATL model transformations are composed of declarative rules, namely *matched rules*. These rules are automatically executed by the ATL execution engine for every match in the source model according to the source patterns of the matched rules. There exit also rules that have to be explicitly called from another rule, namely *(unique) lazy rules* and *called rules*, which gives more control over the transformation execution. The former are called from the declarative part of another rule, while the latter are called from the imperative part.

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

Rules are mainly composed of an *input pattern* and an *output pattern*[4]. The input pattern is used to match *input pattern elements* that are relevant for the rule. The output pattern specifies how the *output pattern elements* are created from the input model elements matched by the input pattern. Each output pattern element can have several *bindings* that are used to initialize its attributes and references.

ATL Transformation Example

An example of an ATL transformation that is used for explanation purposes is shown in Listing 1.1. The input and output metamodels are displayed in Figures 1 and 2, respectively.

The transformation takes as input a model that represents a country, where details about cities, schools and their members are modeled. The output of the transformation is a model of the same country that has some *added information* (AI), which is extracted according to the data of the input model.

Listing 1.1: Excerpt of *Country2CountryAI* ATL Transformation.

```
1 module Country2CountryAI;
```

[4] Please note that the terms source/target and input/output are used synonymously throughout this document

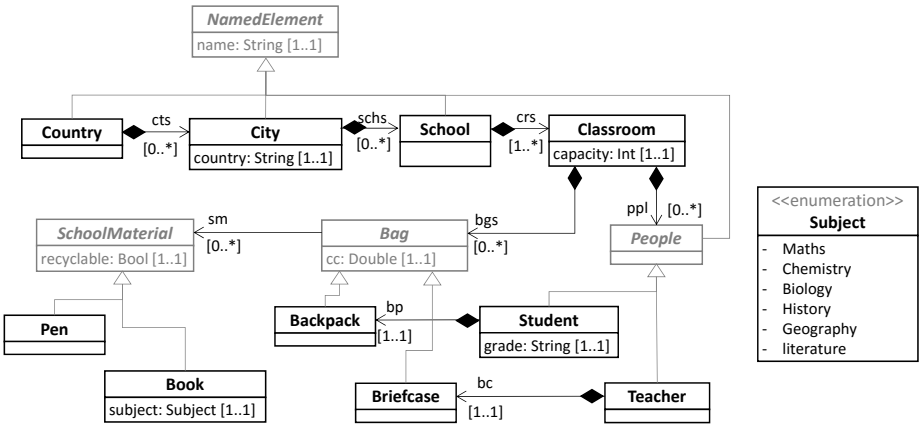

Fig. 1: Input metamodel: *Country Metamodel*

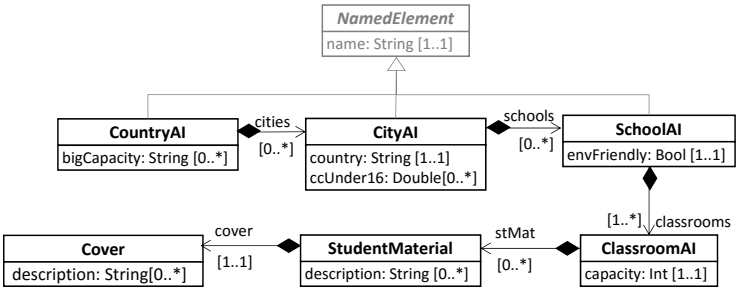

Fig. 2: Output metamodel: *CountryAI Metamodel*

```
2  create OUT : CountryAI from IN : Country;
3
4  helper def : setPenName : String = 'This␣is␣a␣pen␣owned␣by␣a␣student';
5  helper def : setBookName : String = 'This␣is␣a␣book␣owned␣by␣a␣student,
6  ␣and␣therefore␣it␣does␣not␣hold␣that␣' + thisModule.setPenName;
7
8  rule Country2CountryAI{
9    from
10     cIn : Country!Country
11   to
12     cOut : CountryAI!CountryAI(
13       name <- cIn.name,
14       bigCapacity <- cIn.cts -> collect(c|c.schs) -> collect(sch|sch.crs)
             ->select(c|c.capacity>15)->collect(ppl.name),
15       cities <- cIn.cts)
16 }
17
18 rule City2CityAI{
19   from
20     cIn : Country!City
21   to
22     cOut : CountryAI!CityAI(
23       name <- cIn.name,
24       country <- cIn.country,
```

```
25        ccUnder16 <- cIn.schs->collect(s|s.crs)->collect(c|c.ppl)->select(p|
              p.age<16)->collect(p|p.bp)->collect(b|b.cc),
26        schools <- cIn.schs)
27  }
28
29  rule School2SchoolAI{
30    from
31      sIn : Country!School
32    to
33      sOut : CountryAI!SchoolAI(
34        name <- sIn.name,
35        envFriendly <- sIn.crs -> collect(c|c.ppl)->collect(p|p.bc)->collect
              (b|b.sm)->forAll(sm|sm.recyclable),
36        classrooms <- sIn.crs)
37  }
38
39  rule Classroom2ClassroomAI{
40    from
41      cIn : Country!Classroom
42    to
43      cOut : CountryAI!ClassroomAI(
44        capacity <- cIn.capacity,
45        stMat <- cIn.ppl->select(s|s.grade='4A')->collect(s|s.bp)->collect(b
              |thisModule.createStudentMaterial(b.sm)))
46  }
47
48  lazy rule createStudentMaterial{
49    from
50      p : Country!Pen
51    to
52      sm : CountryAI!StudentMaterial(
53        description <- thisModule.setPenName)
54  }
55
56  lazy rule createStudentMaterial{
57    from
58      b : Country!Book
59    to
60      sm : CountryAI!StudentMaterial(
61        description <- thisModule.setBookName)
62    do{
63      sm.cover <- thisModule.CreateCover();
64    }
65  }
66
67  rule CreateCover(){
68    to
69     l : CountryAI!Cover(
70       description <- 'Cover␣of␣a␣book')
71  }
```

3 Rules and Helpers Dependencies in ATL

In order to identify the dependencies within ATL model transformations, the first thing we need to know is what kind of dependencies we can find in them. This section is dedicated to the identification and explanation of such dependencies. Each of them is explained in a subsection, and all of them are exemplified with the help of the model transformation of Listing 1.1.

3.1 Helper - Helper

As it has been explained in the previous section, helpers can be seen as auxiliary functions that can be called from any place in an ATL transformation. Therefore, a helper can be referenced from another helper. In the ATL transformation example of Listing 1.1, we can see in line 6 that helper *setBookName* is referencing helper *setPenName*. This has the same effect as adding the content of helper *setPenName* instead of the call *thisModule.setPenName*.

3.2 Rule - Helper

Helpers are also referenced from any kind of rules. For instance, in line 61 of Listing 1.1, the helper *setBookName* is called from the lazy rule *createStudentMaterial*. The purpose is to assign the string declared in the helper to the *description* attribute of the *StudentMaterial* created by the lazy rule.

3.3 Rule - (Unique) Lazy Rule

Unique lazy rules and lazy rules are used to created new elements in the target model. They are called from matched rules or from another (unique) lazy rule. The difference between lazy rules and unique lazy rules is that the former always create new elements when they are called, independently from the parameters they are called with. The behavior of unique lazy rules is different. They do create new elements when they are first called with any parameter. However, if a unique lazy rules is ever called again with a specific parameter used before to call the same unique lazy rule, it does not create again the same elements, but it returns a pointer to the elements previously created.

We can see in line 45 of Listing 1.1 that the matched rule *Classroom2ClassroomAI* is calling the lazy rule *createStudentMaterial*, and it passes as argument an element of type *SchoolMaterial*, that can either be a *Pen* or a *Book* (cf. metamodel in Figure 1). Therefore, the element *ClassroomAI* created by the matched rule will reference the object *StudentMaterial* created by the lazy rule through its *stMat* reference (cf. metamodel in Figure 2).

3.4 Rule - Called Rule

Called rules are similar to lazy rules, as they can also create new elements. However, they are called from the imperative part of rules. For instance, the lazy rule *createStudentMaterial*, beginning in line 56 of Listing 1.1, contains an imperative part (*do* block). In line 63, we can see that this rule is referencing the called rule *CreateCover*, which creates an element of type *Cover*. Thus, the effect is that the *StudentMaterial* created by lazy rule *createStudentMaterial* will reference, through its *cover* reference (cf. metamodel in Figure 2), the *Cover* created by the called rule.

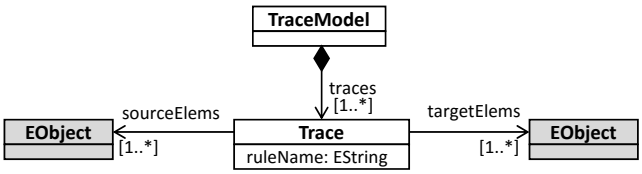

Fig. 3: Trace Metamodel.

3.5 Matched Rule - Matched Rule

In the previous four types of dependencies, the name of the helper, (unique) lazy rule or called rule is explicitly called in the ATL model transformation. This means that it is trivial to automatically identify these dependencies with, for instance, a parser. In contrast, it is not trivial to identify this last type of dependency. Matched rules are declarative rules that are not called explicitly anywhere in the transformation, but they are matched by the ATL engine. However, the elements created in some matched rules may reference some elements created by other matched rules.

Let us have a look at line 15 in Listing 1.1. It is part of the matched rule *Country2CountryAI*. This rule takes an element of type *Country* and creates an element of type *CountryAI*. In line 15, the *cities* reference of the *CountryAI* created is initialized with the elements of type *City* referenced by the matched *Country* (Country.cts). However, an output element cannot reference an element from the input model. Therefore, what the *cities* reference will actually point is the elements that are created from *Country.cts* by another matched rule. The ATL execution engine needs to resolve which are such elements.

In order to resolve such elements, ATL uses an internal tracing mechanism. Thereby, every time a rule is executed, it creates a new trace and stores it in the internal trace model. A trace model can be automatically obtained from a transformation execution, e.g., by using Jouault's *TraceAdder* [2], and is composed of a set of traces, one for each rule execution. The metamodel to which a trace model conforms is displayed in Figure 3. A trace captures the name of the applied rule and the elements conforming to those classes from the source metamodel (*sourceElems* reference) that are used to create new elements conforming to classes in the target metamodel (*targetElems* reference). The classes in the source and target metamodels are represented with *EObject*, since they can be any class of said metamodels. When a trace model is created, its traces reference actual elements of the source and target models. This means that we have three models (the source model, the target model and the trace model) linked by several inter-model references. Therefore, by navigating the traces, it is possible to obtain information of which target element(s) have been created from which source element(s) and by which rule.

An example of a trace model is shown in Figure 4. In this example, the transformation has been executed taking as input a source model composed of one *Country* that has one *City*. The model produced as output by the transformation

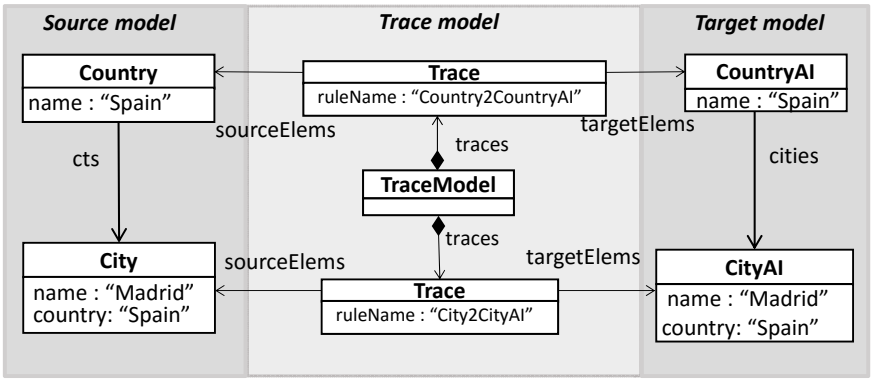

Fig. 4: Traces in a specific transformation scenario of *Country2CountryAI*.

contains a *CountryAI* and a *CityAI*. As we see in the figure, two *Trace* instances have been created, one for each of the two rules that have been executed. Coming back to the example before, line 15 of the model transformation deals with the creation of the *cities* reference in the target model. Therefore, in order to know which *CityAI* must be pointed, ATL analyzes the trace model shown in Figure 4 and determines it.

3.6 Summary

As we have explained, the first four types of dependencies can be straightforwardly identified automatically by any parser. However, the way ATL has to resolve the *matched rule - matched rule* dependency is by using the trace mechanism at execution time. In the next section, we propose our strategy to be able to determine the dependencies of this type without needing to execute the ATL model transformation. It is based on the types extraction of the OCL expressions. In fact, the idea is to statically obtain a dependencies graph that gathers all the dependencies present in an ATL model transformation, of any type. For the running example of this document, the dependencies graph is the one shown in Figure 5.

4 Types Extraction

In order to explain how the types extraction in ATL works, we exemplify it with the ATL transformation of Listing 1.1. The first step in the types extraction is to inject the textual ATL transformation into a model-based representation. It is done automatically by means of a text-to-model transformation. The obtained model conforms to the ATL metamodel, which is in turn made up of three packages: ATL, OCL and PrimitiveTypes. Then, an ATL transformation (in fact, a so-called higher-order transformation) takes the obtained model, as well as

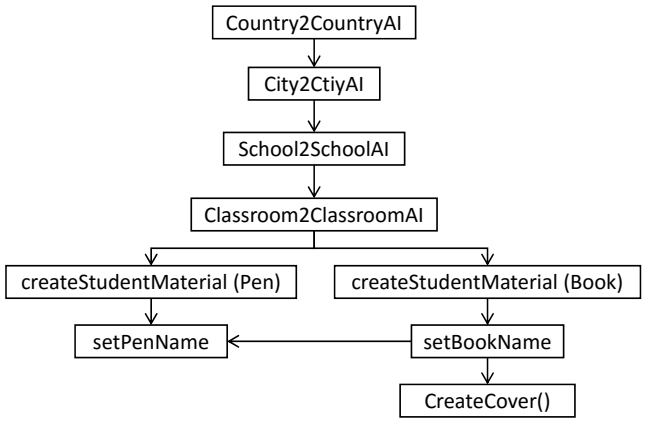

Fig. 5: Dependencies Graph.

the input and target metamodels of the original transformation, and generates a model with information of the types used in each and every rule. Focusing on a rule (let us say a matched rule for instance), it is quite straightforward to obtain the types of the elements in the input pattern of the rules as well as those created in the output pattern. To do so, we need to navigate those objects of type InPattern, OutPattern and Binding of the ATL package[5]. The most challenging part is to extract the types from the OCL expressions, which can be present in the filter part, local variables, bindings and the imperative part. These textual expressions are built conforming to the OCL package[6] of the ATL metamodel. The extraction of the types in the OCL expressions is a three-step process. In the first step, we only need information of the ATL transformation (expressed as a model, as explained before), while in the second and third steps we need information of the source and target metamodels of the transformation in order to be able to navigate them.

An OCL expression can be made up of iterators (in a model level, they are objects of type IteratorExp), such us collect and select. The first step of the types extraction consists of taking every OCL expression and removing the iterators. When doing so, from each OCL expression (that may contain iterators), one or more navigation paths are obtained. Let us explain this with an example. Let us consider the OCL expression of line 14: City -> collect(c | c.schs) -> collect(sch | sch.crs) -> select(c | c.capacity>15) -> collect (ppl.name). First of all, let us recall the purpose of an OCL expression in a model transformation: it is to retrieve an object, or a collection of objects, that need to be accessed by means of a navigation through other elements in the model. In OCL, this is expressed by

[5] A snapshot of the ATL package is available from `http://www.lsi.us.es/~jtroya/documents/ATL.png` (the references to the OCL package are not displayed)

[6] A snapshot of the OCL package is available from `http://www.lsi.us.es/~jtroya/documents/OCL.png` (the references to the ATL package are not displayed)

navigating through the references in a metamodel level. The collect operation is used to specify a collection that is derived from some other collection, and which contains different objects from the original collection. When we have more than one collect operation in an expression, we want to navigate until the last feature of the last collect. If a select or reject operation is found between two collect operations, we firstly ignore it because it does not add information about the overall navigation path. Consequently, for the given OCL expression, one of the navigation paths extracted is City.schs.crs.ppl.name. Although we have ignored the select operation in this navigation path, it has an influence in the object (or collection or objects) extracted by the expression because, at some point in the navigation, it selects a subset of a collection. This is why, for each select operation, we also extract a path that goes from the beginning of the OCL expression until it. Consequently, in the given example, we also extract City.schs.crs.capacity.

With the first step, we manage to transform the OCL expressions that contain iterators into navigation paths from a class in a metamodel until a given feature. Said paths navigate through references in the metamodel. In the second step, we substitute the references in the navigation path for the type (the target class) of such references. The last element in the path, which is an object of type EStructuralFeature according to the Ecore metamodel[7], remains as it is. For the first navigation path extracted before, the path obtained now is City.School.Classroom.People.name. In order to create it, we need to access the metamodel (Figure 1) for obtaining the target class (eType) of each reference. An OCL expression may also navigate through the reference that starts from a subclass of a certain class. For example, the OCL expression in line 25: School -> collect(s | s.crs) -> collect(c|c.ppl) -> select(p | p.age < 16) -> collect(p | p.bp) -> first().cc is accessing the reference bp from objects of abstract type People. This means that the objects are, in fact, of type Student. It is important to take this into consideration in this step. The navigation path extracted, in the first step, from this expression is: School.crs.ppl.bp.cc; and the path extracted in the second step is: School.Classroom.People.Backpack.cc. Similarly, an OCL expression may navigate through the reference starting from a superclass of a certain class. Consider for example the OCL expression in line 35: Classroom -> collect(c | c.ppl) -> (p|p.bc) -> collect(b|b.sm) -> forAll(sm | sm.recyclabe). In the first step, we get the navigation path Classroom.ppl.bc.sm. For the second step, we have to know that sm reference departs from a superclass of Briefcase. We obtain the path Classroom.People.Briefcase.SchoolMaterial.recyclable. The last OCL expression shown contains a forAll operation. It often appears in the filter part of ATL rules and at the last part of the expression, so we navigate until it.

The third step consists of chopping the path obtained in the second step, so that we finally have the most significant types from the OCL expression. We only leave the last two features. Thus, for path School.Classroom.People.Backpack.Bag.cc, we only take Bag.cc.

[7] A snapshot of the Ecore metamodel is available from `http://www.lsi.us.es/~jtroya/documents/Ecore.png`

Finally, we return as output the type(s) of the last expression. For the case of Bag.cc, we return Bag.cc and Bag, since the type of cc is a primitive type. If the last feature is a reference instead of an attribute of a primitive type, we return its type. For instance, from the OCL expression in line 45: Student-> select(s | s.grade = '4A') -> collect(s | s.bp) -> collect(b|thisModule.createStudentMaterial(b.sm), one of the navigation paths we get is (first step) Student.bp.sm, from which we obtain (second step) Student.BackPack.sm, then we chop it and have (third step) Backpack.sm. Finally, we return Backpack.sm, Backpack and SchoolMaterial. When the last feature is of an enumeration type, we retrieve such type. For example, if we obtain in the third step Book.subject, then we return Book.subject, Book and Subject.

After the types of all rules have been extracted, we can automatically determine the *matched rule - matched rule* dependencies. Thus, we say that a rule, R1, *depends on* another rule, R2, if the intersection of the types of the bindings of R1 with the ones of the *InPatternElements* of R2 is not empty.

References

1. Fleck, M., Troya, J., Kessentini, M., Wimmer, M., Alkhazi, B.: Model Transformations Modularization as a Many-Objective Optimization Problem. IEEE Transactions on Software Engineering pp. 1–22 (2016), submitted
2. Jouault, F.: Loosely Coupled Traceability for ATL. In: Workshop Proc. of ECMDA (2005)
3. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
4. Project, E.M.: Atlas Transformation Language – ATL. `http://eclipse.org/atl` (2015)
5. Troya, J., Vallecillo, A.: A Rewriting Logic Semantics for ATL. Journal of Object Technology 10, 5:1–29 (2011)