Ingredients

Reading: We continue to rely on [1], already used in the previous lecture.

The QVT is specification is available at http://www.omg.org/spec/QVT/. If you want to learn more about ATL, the language guide is available at http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language

1 A Glimpse at QVT Relations

Remember, that our interest is in transforming models. In the past lecture, we have discussed how models can be translated to models in other languages using an imperative programming language Xtend. Today we look at rule based alternatives.

The cited survey lists multiple approaches to this problem. In particular it seems appealing to specify such transformation as rewrite rules, that match patterns in graphs and transform them into subgraphs adhering to another metamodel. This gives quite a high level mode of operation. Let us get a flavour of such an approach by looking at the following example of a *QVT relations* transformation.

The following example presents a simple meta-model of a class diagramming language, and a simple metamodel of relational database schema:

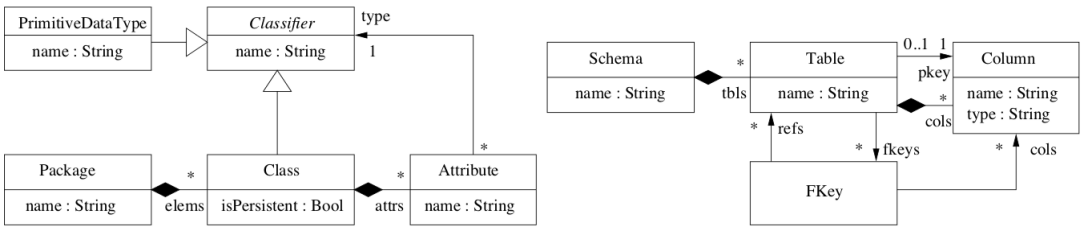

Figure 1: Figure from [1]

Our objective is to be able to translate between relational schema and class diagrams, in either direction. Our transformation would be able to create a proper relational schema for persistence of instances of our class diagram, and vice-verse: a proper class diagram describing structure of data that can be recovered from a give relational database. This can be specified by the following QVT Relations transformation [1]:

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {

  key Table (name, schema)
  key Column (name, table)

  top relation PackageToSchema {
    domain uml p:Package {name = pn}
    domain rdbms s:Schema {name = pn}
  }

  top relation ClassToTable {
    domain uml c:Class {
      package = p:Package {},
      isPersistent = true,
      name = cn
    }
    domain rdbms t:Table {
      schema = s:Schema {},
      name = cn,
      cols = cl:Column {
        name = cn,
        cols = cl:column {
          name = cn + '_tid',
          type = 'NUMBER'},
        pkey = cl
    }
    when {
      PackageToSchema (p,s);
    }
    where {
      AttributeToColumn(c,t);
    }
  }
  ...
```

Mappings are represented as relations, each having one domain declaration per domain involved in the transformation. Free variables in patterns are bound to values occurring in instances of the domain metamodels. So, for instance, in the first rule above 'pn' is a free identifier, and the first rule for Packages and Schemas, means that they will have the same name.

QVT Relations specifications are executable in either direction — the direction of execution is specified at runtime. Clearly, not all transformations can be implemented with QVT Relations, as not all transformations are reversible.

The QVT standard contains a few other languages (including an imperative language). The implementation of QVT within Eclipse is underway (http://wiki.eclipse.org/M2M/QVT_Declarative_(QVTd)).

```
1  module DoCapitalizeNames;
2  create OUT: tripLoose from IN: tripLoose;
3
4  abstract rule CapitalizeName {
5      from s :tripLoose!NamedElement
6      to t :tripLoose!NamedElement (name <- s.name.toUpper() )
7  }
8
9  rule CapitalizePerson extends CapitalizeName {
10     from s: tripLoose!Person to t: tripLoose!Person
11 }
12
13
14 rule CapitalizeCar extends CapitalizeName {
15     from s: tripLoose!Car to t: tripLoose!Car
16 }
17
18 rule CapitalizeTrip extends CapitalizeName {
19     from s: tripLoose!Trip
20     to t: tripLoose!Trip (
21             driver <- s.driver,
22             passengers <- s.passengers,
23             vehicle <- s.vehicle
24         )
25 }
26
27 rule CaptializeTripModel extends CapitalizeName {
28     from s: tripLoose!TripModel
29     to t: tripLoose!TripModel ( elements <- s.elements )
30 }
```

Figure 2: Capitalize all names in ATL.

2 Rule-based M2M Transformation with ATL

ATL is a M2M transformation language originally developed as a response to the OMG Request for Proposals for the QVT standard. It is now one of the popular transformation languages, with a free implementation compatible with Eclipse's modeling tools. If you followed our standard procedure for installing Eclipse, then you already have ATL tooling installed.

ATL supports both declarative and imperative style of transformations. We have largely covered the imperative style, when talking about xtend. This part of the lecture aims at demonstrating the rule-based style.

Figure 2 shows an example of an ATL transformation capitalizing all names in an instance of the tripLoose metamodel (Sec. 3). If you recall, we have implemented this transformation in Java in one of the earlier lectures. The Java transformation was an endo-transformation, so it modified the input model by raising the case of all names. ATL also allows making endo-transformations, which it calls refinements. We decided to show a copying transformation, which is a bit longer, but this way we can demonstrate the ATL rules more clearly.

The header of the transformation names the module, and establishes the source and target meta-models (the precise URI's to meta-model ecore files are established in the run configuration of the transformation, or in the workflow, if you incorporate it into a workflow). In

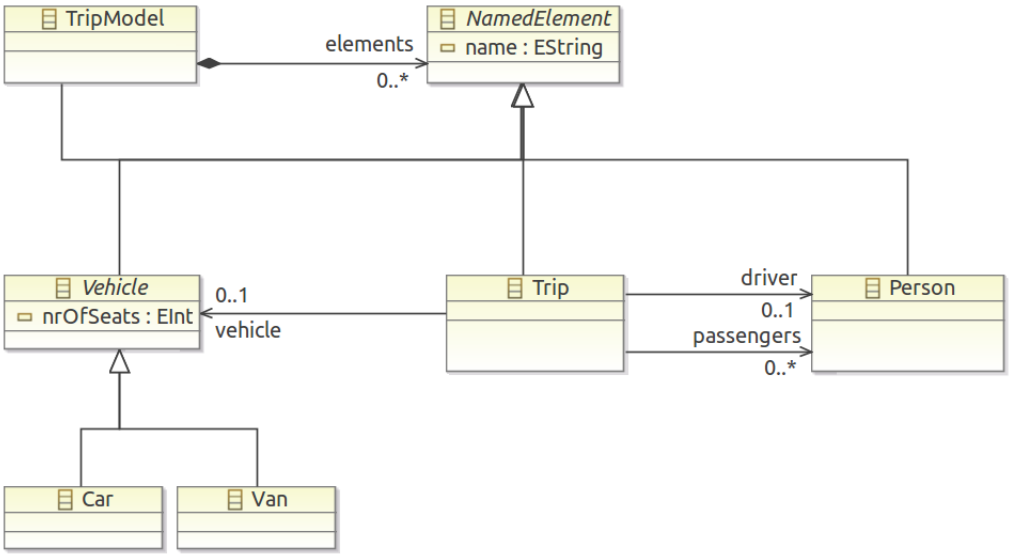

Figure 3: The tripLoose meta-model from previous lectures. It is identical to Trip, but does not contain the backwards reference from Persons to Trips.

this case the input and output metamodels are the same: we are going to copy a tripLoose instance, and while we do the copying, we will change all names to uppercase.

Then we have five transformation rules. The first rule is abstract: it applies to abstract classes, and it is extended by the other four rules. This way the actual capitalization is only written once. All the other rules simply do the copying of the input model elements.

Each rule has a **from** binding and a **to** section. In parentheses of the latter, attributes of the target model elements are created. The leftarrow symbol denotes assignments, and the right hand side expressions are written in OCL (so OCL is the expression language of ATL, which reflects the fact that ATL was developed as a proposal of an OMG standard).

Note that in comparison with Java and Xtend, one does not need to write any scheduling code, or any model traversal code — the rules are applied to all instances of types that satisfy the application conditions.

The other example (see Fig. 4) shows a transformation that converts a tripLoose model into a trip model. If you recall, this requires restoring the backwards reference from Persons to Trips, which lists the trips in which a Person participates. We have implemented such a transformation in Xtend in the previous lecture.

The transformation uses a helper function that given a person computes a sequence (a set) of all trips, which contain this person in the passenger list. The body of the function is just an OCL expression, that uses the select collection iterator. This function is called in the Person2Person rule.

The remaining rules just copy Cars and Trips (one more rules for TripModel is omitted for brevity).

```
module tripLoose2trip;
create OUT: trip from IN: tripLoose;

helper def : tripsReferringTo(p :tripLoose!Person) : Sequence(tripLoose!Trip) =
    tripLoose!Trip.allInstances()->select(t | t.passengers->includes(p) );

rule Person2Person {
    from
        s: tripLoose!Person
    to
        t: trip!Person (
            name <- s.name,
            trips <- thisModule.tripsReferringTo (s)
        )
}

rule Car2Car {
    from
        s: tripLoose!Car
    to
        t: trip!Car (
            name <- s.name,
            nrOfSeats <- s.nrOfSeats
        )
}

rule Trip2Trip {
    from
        s: tripLoose!Trip
    to
        t: trip!Trip (
            name <- s.name,
            driver <- s.driver,
            passengers <- s.passengers,
            vehicle <- s.vehicle
        )
}
```

Figure 4: Convert tripLoose instances to trip instances using ATL.

Observe the apparent type mismatch in object creation rules. For example in line 35

```
vehicle <- s.vehicle
```

assigns an object of type `tripLoose!Vehicle` to an attribute of type `trip!Vehicle`. This is not an error. The ATL execution language calls the transformation on the Vehicle (here the Car2Car rule) to convert the object first, before the reference is assigned.

ATL also offers so called unique lazy rules, that avoid multiple creation of objects, even if called multiple times (they correspond to xtend's create methods).

We mentioned before that large collections of meta-models exists (so called meta-model Zoo) freely accessible online. ATL boasts a nice free collection of transformations at: http://www.eclipse.org/m2m/atl/atlTransformations/. It contains many interesting transformations. For example a converter from MOF models to proper UML class diagrams, extractors of information from Excel files, mapping from MySQL schema to meta-models, computations of metrics, make 2 ant, and many others ...

3 Example Exam Questions

⚠ What are the main characteristics of the ATL language? What are the advantages of writing transformations in a language like ATL over Java or Xtend? Explain a simple transformation provided to you on paper.

Of course, if you used ATL in your project, you will be asked more intricate questions about it.

References

[1] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches, 2006.