Solution of adapting QVTo metamodel

# Concepts that need to be unified

Before doing adaptations, some concepts should be unified for the readers:

1. **Concrete syntax.**

Concrete syntax is a concept contrasted with abstract syntax, it is part of the definition of the language. It tells the programmer how to put strings together when programming (in other words, how programmers write the code).

In our case, the EBNF grammar from the QVT specification is the concrete syntax, while the contents in the \*.xtext file describe the concrete syntax.

1. **EBNF grammar.**

In computer science, EBNF is a way (or a notation) for describing formal language grammar. In our case, when we say the EBNF grammar, we refer to the concrete syntax described by EBNF and it is provided by OMG in the QVTo specification document. For example, there is EBNF grammar for QVTo V1.0 in section 8.4.7.1. **Terminal/Non-terminal.**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **terminal and nonterminal symbols** are the lexical elements used in specifying the [production rules](https://en.wikipedia.org/wiki/Production_(computer_science)) constituting [formal grammar](https://en.wikipedia.org/wiki/Formal_grammar). *Terminal symbols* are the elementary symbols of the [language](https://en.wikipedia.org/wiki/Formal_language) defined by formal grammar. *Nonterminal symbols* (or *syntactic variables*) are replaced by groups of terminal symbols (or non-terminals) according to the production rules.

As an example, let’s look at the following line:

<transformation\_refine> ::= 'refines' <moduleref>

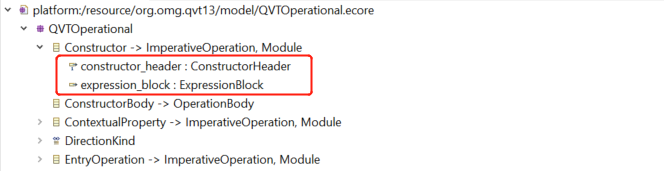
The expressions <transformation\_refine> and <module-ref> are non-terminals, whereas ‘refines’ is a terminal.

1. **Metamodel.**

A metamodel is a model that describes the structure and semantics of another model. In our case, we refer to the Ecore metamodel in Eclipse Modeling Tools.

1. **Attribute.**

There are attributes under a class, cf. the following figure:

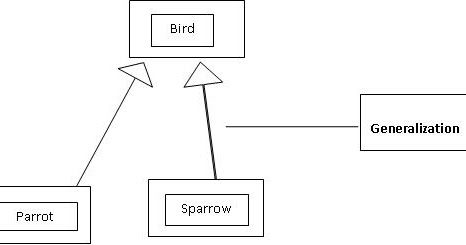


In the above figure, ‘constructor\_header’ and ‘expression\_block’ are the two attributes of class ‘Constructor’. This is a concept in the ecore metamodel.

Note: we have a concept “attribute” in Xtext too which will be introduced at late point.

1. **Generalization.**

Generalization is a basic concept in many programming languages, e.g. java. All subclasses inherit the properties of the parent class.



As an example, we can see in the above figure, the class ‘Bird’ are the superclass of the class ‘Parrot’ and ‘Sparrow’, the relationship between them is called ‘Generalization’.

1. **Supertype.**

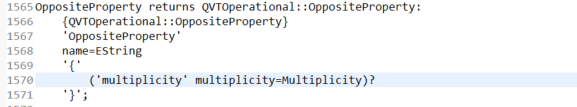
I would call a superclass is ‘supertype’ too, as it is the formal name in the tool Eclipse Modeling Tools. Even though it is not a general name in computer languages. Anyway, let’s call supertype because our work is very based on Eclipse Modeling Tools.

1. **Xtext file.**

Xtext file is a file that ends with the extension ‘.xtext’ which is used to define the language grammar, that is the content of it describes the concrete syntax of a language. And it can be processed by Xtext. What’s more, it can be manually defined or generated from an existing ecore metamodel.

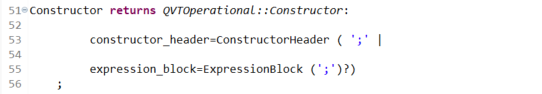
1. **Grammar Rule.**

A little has been mentioned above, i.e., Xtext would generate a grammar rule for each of the classes in the Ecore metamodel when we generate the Xtext projects from the existing Ecore metamodel. The grammar rules in the \*.xtext file describe the concrete syntax of the language, while the Ecore metamodel defines the abstract syntax of the language. Cf. a figure shows an example of a grammar rule:



1. **Attribute in Grammar rule.**

We mentioned “attribute” above which is a concept in ecore metamodel, while there are also attributes in a grammar rule too (we have introduced the concept of grammar rule in the last point). Please see the following figure which shows a grammar rule in a xtext file:



This figure is a screenshot of a fragment of text in the Xtext file (i.e., the language grammar file). This fragment is a grammar rule that Xtext generates for the class ‘Constructor’. The two expressions (the line starts with ‘constructor\_header’ and the line starts with‘expression\_block’) are the attributes. When we say the left expression ‘constructor\_header’ that also means we are referring to the whole attribute.

1. **OMG metamodel, original metamodel, and official metamodel.**

OMG organization provides all versions of QVTo metamodels on its website, for example in <https://www.omg.org/spec/QVT/1.3/About-QVT/> , the metamodel of QVTo could be downloaded here. It is usually necessary for a language developer to adapt the QVTo metamodel before creating the QVTo language (the reason will be explained later), so we call the one we downloaded from the OMG website ‘OMG metamodel’.

1. **Literal.**

The alternative values in an enumeration are called “Literals”. “Literal” is the formal name in Eclipse Modeling Tools.

1. **Counterpart, Missing.**

Every non-terminal in the concrete syntax (i.e., the EBNF grammar) is corresponding to each class or enumeration in the abstract syntax (i.e., the Ecore metamodel). We call a non-terminal and its corresponding class/enumeration a pair of counterparts.

As an example, there is a non-terminal called <helper> in the EBNF grammar, and there is a class called ‘Helper’ which is the counterpart for the non-terminal <helper>. Both counterparts don’t need to have the exact same name.

For the right expressions of a production rule in the EBNF grammar, there should be an attribute or an association (i.e., containment or reference) as a counterpart for each element with <>. Containment and reference are also attributes from an Ecore metamodel perspective. Let’s see the following production rule:

<import> ::= 'from' <unit> 'import' (<identifier\_list> | '\*') ';'

| 'import' <unit> ';'

There is a class called ModuleImport in the metamodel as a counterpart for the non-terminal <import>. Then on the right side of <import>, there is a non-terminal <unit>, so there should be an attribute as a counterpart for it, but there isn’t. This is called ‘Missing’.

Another example of the opposite is there is a class ‘Library’ in the QVTo metamodel, and there is an expression called <library> in the EBNF grammar, they are the counterparts for each other.

# QVTo metamodel adaptation

## Classification of adaptations

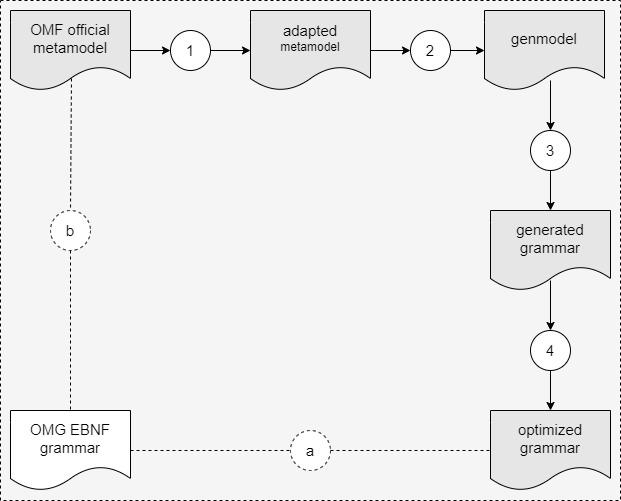
The below figure is the workflow of generating an transformed Xtext grammar from an OMG official metamodel. The steps are

Step 1: Create an empty EMF project and import the official metamodel which is downloaded from the OMG website. And adapt the metamodel (we will explain why we adapt the metamodel below).

Step 2: Create a genmodel file from the ecore metamodel.

Step 3: Generate the Xtext file from the genmodel file. The xtext file contains the contents that describe the concrete syntax of the language.

Step 4: Transform the grammar by modifying the texts in the xtext file. We have many texts describing the motivation for transforming the grammar in the paper.



We adapt the QVTo metamodel for two reasons:

1. To make sure that the genmodel could be created and run for the metamodel, cf. step ‘2’.
2. To make the Xtext grammars consistent with the EBNF grammar, cf. consistency ‘a’. In other words, this criteria is for adding missing non-terminals from the concrete syntax (i.e., the EBNF grammar) to the metamodel or correcting the type inconsistency issues. The addition includes two types, i.e., 1) associations (containment/referent), and 2) generalization (called ‘supertype’ in Eclipse Modeling Tool).

## Generalized rules for adapting QVTo metamodels

### Counterparts for EBNF productions in the OMG metamodel

Because we are adapting the OMG QVTo metamodel, we need to first determine the counterparts of the production expressions in EBNF in the metamodel. We found the following counterparts in the OMG metamodel for the production rules in the EBNF grammar: <https://docs.google.com/spreadsheets/d/1_1ZC7RTwu7twYiJo2gNGPZtbREpbZ3fY/edit#gid=2068672572>

Once we can not find a counterpart in the metamodel for production in the EBNF grammar, then we add one. This is the basis for creating the adaptation criteria MAR-1,2,3.

### Criteria and Action of Applying the Metamodel Adaptation Rules.

The criteria and the corresponding actions of applying the adaptations in the QVTo metamodel have been listed in the below table. Among them, the ‘extra step’ below is a separate step for excluding OCL expressions which is the same as what we do in ATL.

|  |  |  |
| --- | --- | --- |
| Metamodel adaptation rules | Application Criteria | Action  (with examples) |
| **MAR-1** | **The counterpart (i.e. class or enumeration) in the metamodel for a non-terminal in the EBNF grammar is missing.**  **Note: if the non-terminal is just a way to express the multiplicity, then the missing counterpart for it in the metamodel could be skipped.**  For example, there is an expression <top\_level> in the EBNF grammar, its counterpart is missing in the metamodel.  Another example, <expression\_list> is just a way to express <expression>\*, so we don’t need to create a counterpart for it in the ecore metamodel. | **Add the missing counterpart in the metamodel by adding a metaclass or an enumeration before creating the genmodel file. When we add the class (or the enumeration ), the name should be the same name as the name of the non-terminal in the EBNF grammar.**  **Two special cases here:**   1. **if the name of the non-terminal is exactly the same as a java keyword, then name the added attribute as "moduleXXX" (here XXX is the name of the non-terminal;** 2. **if the name of the non-terminal is exactly same as a class name exists in other ecore, then name the added class as "YYYGO" (here YYY is the name of the non-terminal).**   In the example mentioned in the left column, we add a class ‘TopLevelGO> as a counterpart for the expression <top\_level> in the EBNF grammar. |
| **MAR-2** | **The counterpart (i.e. an attribute or literal) in the metamodel is missing for a terminal/non-terminal on the right-hand side of a production rule in the EBNF grammar.**  For example:  *<param\_list> ::= <param> (',' <param>)\**  We have created a class called ‘ParamList’ in the metamodel as a counterpart for the non-terminal <param\_list>, then there should be an attribute as the counterpart for the non-terminal <param>, while it is missing in the metamodel. | **Add the missing counterpart (attribute or literal) to the metamodel for the terminal/non-terminal in the EBNF grammar. When we add an attribute (or a literal), the name should correspond to the name of the terminal/non-terminal in the EBNF grammar. And the type of the added attribute should be the type corresponding to the non-terminal with the same name. The lower/upper bounds of the added attribute should exactly be the same as what the corresponding terminal/non-terminal indicates.**  Here, in the example mentioned in the left column, we add an attribute ‘param’ as the counterpart of the non-terminal <param>. The type of the attribute ‘param’ should be the corresponding class for the non-terminal ‘param’ which is class ‘VarParameter’. Here, <param> is from 0 to infinite. So the lower bound of the added attribute ‘param’ should be 0 in the metamodel; while the upper bound of the added attribute ‘param’ should be -1 in the metamodel.  Note: -1 in Eclipse Modeling Tools means infinite. |
| **MAR-3** | **In the EBNF grammar, if there are multiple non-terminal symbols in the right expression of the production, and they have only one relationship, that is, the relationship of "or" with each other. Then the class in the meta-model corresponding to the non-terminal in the left expression is the superclass of the classes in the meta-model corresponding to each non-terminal in the right expression. If there is no generalization relationship between the class corresponding to a nonterminal in the right expression and the class corresponding to the nonterminal in the left expression, it needs to be added.**  For example, there is a non-terminal <unit\_element> in the EBNF grammar, and its counterpart in the metamodel is the class Module. And there is a non-terminal <constructor>, and its counterpart in the metamodel is the class Constructor. In the production rule for the non-terminal <unit\_element>, we could see that <constructor> is one of the terms in the right expression of <unit\_element>, i.e., <unit\_element> could be <constructor>. However, there is no relationship between the class Module and the class constructor. There should be a generalization association between them, i.e., class Module should be the supertype of class Constructor. | **Add the missing generalization (i.e., ‘SuperType’ in Eclipse Modeling Tools) for the two classes in the metamodel, that is, set the class corresponding to the non-terminal in left-expression as the superclass of the classes corresponding to the non-terminals in the right-expression.**  In the case we mentioned in the left cell, we add a supertype relationship between class ‘Module’ and class ‘Constructor’, that is the class ‘Module’ is the supertype of class ‘Constructor’ |
| **MAR-4** | **Attributes with the same name in superclass and subclass cause errors in the metamodel. Eclipse Modeling Tools calls this error “feature conflict” which can be seen in the error messages.**  For example, after adding the missing generalization to the metamodel (this is explained above), the class ContextualProperty got a new supertype ImperativeOperation so that the class ContextualProperty will derive the attributes from the class ImperativeOperation. However, class ContextualProperty and class ImperativeOperation both contain attributes ‘overridden’ and ‘context’, that is, the same attributes are repeated in the superclass and subclass. This causes a feature conflict error in the class ContextualProperty. | **Remove the repeated attribute(s) from the subclass.**  For the example we mentioned in the left cell, we removed the attributes ‘overridden’ and ‘context’ from the class ‘ContextualProperty’. And it will still have the two attributes by deriving from its supertype ‘ImperativeOperation’.  (ContextualProperty is not a supertype for any other class. If it is, then removing repeated attributes from it in such a case will still not affect anything, because its subclass will also inherit the two attributes). |
| **MAR-5** | **There is the attribute(s) with the same name in two (or more) superclasses of a class in the metamodel.**  For example, after adding the missing generalization to the metamodel, class ModelType got a new super class Module. ModelType then has two superclasses simultaneously, i.e., type Class and type Module, both of which are derived from the class ‘NamedElement’ indirectly. That means both super types would have an attribute ‘name’ (derived from NamedElement), and this causes a feature conflict error in ModelType. | **Remove the unnecessary supertype from the metamodel with the help of referring to the EBNF grammar. The removal would cause losing attributes derived from the removed supertype, and there are no counterparts in the EBNF grammar for those attributes.**  For the example we mentioned on the left, we analyze it by referring to the EBNF grammar. Class ‘ModelType’ is the counterpart of non-terminal <modeltype> in the EBNF grammar, and class ‘Module’ is the counterpart of non-terminal <unit\_element>. The non-terminal <unit\_element> can be replaced by <modeltype>, that is class Module should be the supertype of ModelType (this is why we add the generalization). The EBNF grammar doesn’t show there is another non-terminal that could be interpreted by <modeltype>. So supertype ‘Class’ is not necessary for the class ‘ModelType’, so we could remove it to address the ‘feature conflict’ error. |
| **MAR-6** | **Failed to create the genmodel due to an error. When the error occurs, there will be an error message in the message window “The containment reference of a type with a container feature org.eclipse... that requires instances to be contained elsewhere cannot be populated.”** | **Changed the lower bound of the attribute mentioned in the error message from 1 to 0.**  The lower bound of attribute ‘package’ in the class ‘Type’ in the original EMOF.ecore in version 1.0 is 1, this leads to a failure for creating a genmodel from the ecore metamodel. So we changed it from 1 to 0. |
| **MAR-7** | **Failed to generate Xtext artifacts for the language from the Xtext grammar file due to errors. When the errors occur, there are error messages in the message window: Couldn’t resolve reference to EClassifier ‘TopLevelGO’. There are lines of error messages for all the classes in the metamodel. There should be no standardized file extension (such as ‘.xml’) in the value of ‘Ns URI’.** | **Reconfigure the value of ‘Ns URI’ to the form of “**[**http://schema.omg.org/spec/QVT/X.X/qvtoperational**](http://schema.omg.org/spec/QVT/X.X/qvtoperational)**” (X.X here means the version of the QVTo).**  In our case, we remove the '.xml' substring from the 'Ns URI' string. Because the wrong value causes the error “Couldn't resolve reference to EPackage”. |
| **Extra step** | **Exclude the OCL part, including calls to OCL types, attributes in OCL types, and generalization relationship with OCL types.** | **To exclude the Expression part from the QVTOperational.ecore, the following operations should be done:**   1. **Create a dummy type ExpressionGO with a string (the string is for the dummy content) in the QVTOperational.ecore once.** 2. **Replace the type OclExpression with ExpressionGO in all the attributes in the metamodel qvtoperational.ecore which are defined as OclExpression.** 3. **Change the OCL super types to the dummy class (i.e., Expression GO).** |

The application results of MAR-4,5,6,7 in the generation of the genmodel in QVTo V1.0 are:

<https://docs.google.com/spreadsheets/d/1_1ZC7RTwu7twYiJo2gNGPZtbREpbZ3fY/edit#gid=2063885379>

The application results of MAR-4,5,6,7 in the generation of the genmodel in QVTo V1.1 are:

<https://docs.google.com/spreadsheets/d/11h7S8rdozb1Ns_7QHcP0SCETu5lNNGGYVgIOEA3Tf0k/edit#gid=0>

The application results of MAR-4,5,6,7 in the generation of the genmodel in QVTo V1.2 are:

<https://docs.google.com/spreadsheets/d/1ex9ZWfoCik8wxTmGhKbnCsfAA9bNQDD10t9RdvM8SOs/edit#gid=1642646231>

The application results of MAR-4,5,6,7 in the generation of the genmodel in QVTo V1.3 are:

<https://docs.google.com/spreadsheets/d/1yB1nNspoWd-R3ijMq0YXqt7N7OqXHZ0INSEQbZJ3sL4/edit#gid=1129535824>

The application results of MAR-1,2,3 and the extra step in the QVTo V1.0 are:

<https://docs.google.com/spreadsheets/d/1_1ZC7RTwu7twYiJo2gNGPZtbREpbZ3fY/edit#gid=1800008321>

The application results of MAR-1,2,3 and the extra step in the QVTo V1.1 are:

<https://docs.google.com/spreadsheets/d/11h7S8rdozb1Ns_7QHcP0SCETu5lNNGGYVgIOEA3Tf0k/edit#gid=1953468721>

The application results of MAR-1,2,3 and the extra step in the QVTo V1.2 are:

<https://docs.google.com/spreadsheets/d/1ex9ZWfoCik8wxTmGhKbnCsfAA9bNQDD10t9RdvM8SOs/edit#gid=349524193>

The application results of MAR-1,2,3 and the extra step in the QVTo V1.3 are:

<https://docs.google.com/spreadsheets/d/1yB1nNspoWd-R3ijMq0YXqt7N7OqXHZ0INSEQbZJ3sL4/edit#gid=0>