YATL: Yet Another Transformation Language

Octavian Patrascoiu

Computing Laboratory, University of Kent, United Kingdom

O.Patrascoiu@kent.ac.uk

Abstract

With the increased use of modelling techniques has come the desire to use model transformations. Model transformations systems are graph transformations systems that perform translations between languages defined by a corresponding metamodel. The current paper proposes a transformation language called YATL (Yet Another Transformation Language). This transformation language has been defined to perform transformations within the OMG's MDA framework. After having presented YATL, we present several experiments to show how YATL can be used to map from a source model to a target model. YATL is still evolving since it is supposed to match the forthcoming QVT standard.

1. Introduction

The Model-Driven Architecture (MDA) [14][6] is an initiative of the Object Management Group (OMG) to define an approach to software development based on modelling and automated mapping of models to implementations. The basic MDA pattern allows the same platform-independent model (PIM), which specifies business system or application functionally and behaviour, to be mapped automatically to one or more platform-specific models (PSMs).

The MDA approach promises a number of benefits [14][4]:

- Improved portability due to separating the application knowledge from the mapping to a specific implementation technology.
- Increased productivity due to automating the mapping.
- Improved quality due to reuse of well-proven patterns and best practices in the mapping.
- Improved maintainability due to better separation of concerns.
- Enables different applications to be integrated by explicitly relating their models: this facilitates integration and interoperability and supports system evolution as platform technologies change.

While the current OMG standards such as Unified Modeling Language (UML) [18] and Meta Object Facility (MOF) [15] provide a well-established foundation for defining PIMs and PSMs, no such well-established foundation exists for transforming PIMs to PSMs [7]. In 2002, in its effort to define the transformations, OMG initiated a standardization process by issuing a Request for Proposal (RFP) on Query / Views / Transformations (QVT) [9]. This process will lead to an OMG standard for defining model transformations, which will be of interests not only for PIM-to-PSM transformations, but also for defining views on models and synchronization between models. Driven by practical needs and the OMG's request, a large number of approaches to model transformation have been recently proposed [4].

In this paper, we describe YATL, a transformation language developed within the Kent Modelling Framework (KMF) [9]. The compiler and interpreter for YATL are implemented in Java and are designed to maximise the portability to different modelling environments/tools. Both language processors contain a core of elements (classes, methods etc.), which are independent of the modelling environment/tool. The features that are environment-dependant are implemented using delegation. This approach allows a fast implementation under different modelling framework, for example Eclipse Modelling Framework (EMF).

Parts of the above language processors were built using MDA techniques:

- The lexical analyser, the parser and translators were generated automatically using Syntax-Driven Translation Scheme (SDTS), lexical analysers, and parsers generators.

- The Java code associated to the YATL's abstract syntax was generated using KMF Studio, a tool provided by KMF, using the MOF model of the abstract syntax as input.

These parts can be easily regenerated for other environments if appropriate generation tools are provided. For instance, if the target language is C#, the above parts of the language processors can be easily generated as C# parser generators are available and KMF can be configured to generate C# code.

2. Design features

The Yet Another Transformation Language is a hybrid language (a mix of declarative and imperative constructions) designed to express model transformations as required by the MDA [14][6] approach and to answer the Query/Views/Transformations Request For Proposals [10] issued by OMG. It is described by an abstract syntax (a MOF metamodel) and a textual concrete syntax. It does not provide yet a graphical concrete syntax as QVT RFP suggested. A transformation model in YATL is expressed as a set of transformation rules. The recommended style of programming is declarative. Transformations from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) can be written in YATL to implement the MDA.

This paper presents the current version of YATL, which is evolving in order to support all the features provided by [9] and the future QVT standard.

We designed YATL to support beside the ongoing QVT requirements, the following additional requirements:

- Transformation engine must be capable to perform efficient transformation for large-scale systems.
- The process of applying the transformation rules must be deterministic.
- Syntax and semantics of YATL must be well-defined.
- YATL should be a hybrid language containing both declarative and imperative features.
- Queries, views, and transformations are organized in namespaces to provide reusability and avoid name collision.
- YATL must provide all the required computational power, regardless of the host platform or language.

3. Namespaces and Translation Units

A YATL program consists of one or more translation units, each contained in a separate source file. When a YATL program is processed, all of the translation units are processed together. Thus, translation units can depend on each other, possibly in a circular fashion. A translation unit consists of zero or more import directives followed by zero or more declarations of namespace members: queries, views, or transformations.

The concept of namespace was introduced to allow YATL programs to solve the problem of names collision that is a vital issue for large-scale transformation systems. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system—a way of presenting program elements that are exposed to other programs. A YATL program can reuse a transformation by importing the corresponding namespaces and invoking the appropriate rules.

A YATL query is an OCL expression, which is evaluated into a given context (package, classifier, property, operation etc.). The returned value can be a primitive type, model elements, collections or tuples. Queries are used to navigate across model elements and to interrogate the population stored in a given repository. YATL uses the OCL implementation that was initially developed under KMF and then under Eclipse as an open source project [1][2].

A YATL transformation is a construct that maps a source model instance to a target model instance by matching a pattern in a source model instance and creating a collection of objects with given properties in the target model instance. The matching part is performed using the declarative features of OCL, while the creation of target instances is done using the imperative features provided by YATL. YATL provides also the possibility of interacting with the underlying machine using *native* statements. Although we do not encourage the use of such features, they were provided to support the modeller when some operations

are not available at the metamodel level (e.g. the standard library of OCL 2.0 does not provide a function to convert lowercase letters to uppercase letters).

4. Views

According to [10] a transformation model language shall enable the creation of a view of a metamodel. Views are not supported by YATL yet. This is an area of ongoing research.

5. Transformations

To support the mandatory requirements from [10], a YATL transformation describes a mapping between a source MOF metamodel S, and a target MOF metamodel T. The transformation engine uses the mapping to generate a target model instance conforming to T from a source model instance conforming to S. The source and the target metamodels may be the same metamodel. Navigation over models is specified using OCL.

Each transformation contains one or more transformation rules. A transformation rule consists of two parts: a left-hand side (LHS) and a right-hand side (RHS). The LHS of a YATL transformation is specified using a filtering expression written either in OCL or native code (Java, C#, scripts etc). This approach allows filter expressions to include both modelling information (e.g. navigational expressions, properties values, collections, etc.) and platform dependent properties (e.g. special conversion functions), which makes them extremely powerful. A compound statement specifies the effect of the RHS. The LHS and RHS for the YATL transformation are described in the same syntactical construction, called transformation rule. A rule is invoked explicitly using its name and with parameters.

The abstract syntax of YATL namespaces, translation units, queries, views, transformations, and transformations rules is described in Figure 1.

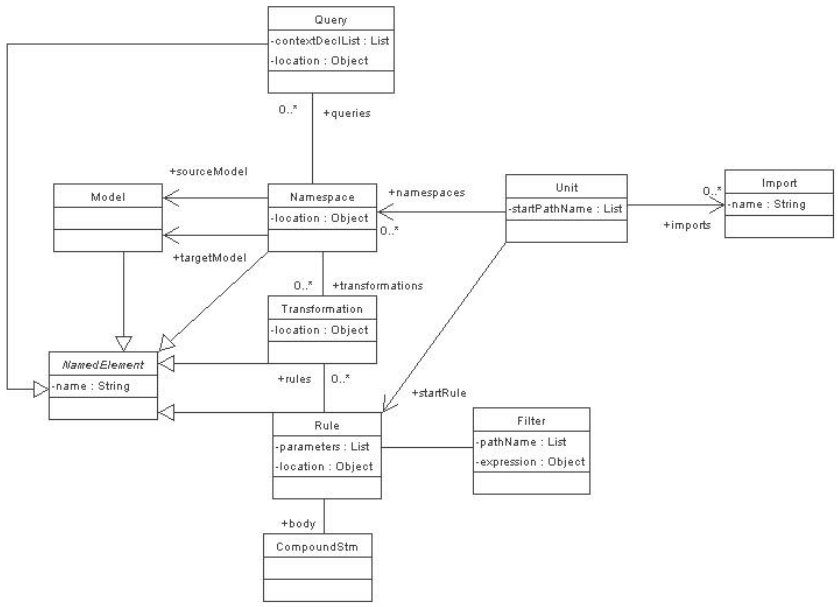

Figure 1 Abstract Syntax

6. Declarative features

YATL is a hybrid language containing both declarative and imperative constructions. The declarative features come mainly from OCL expressions and the description of the LHS of transformation rules. YATL acts similar to a database system that uses SQL to interrogate the database and the imperative host language to process the results of the query. We choose OCL to describe the matching part of YATL rules because it is a well-defined language for querying the UML models, it provides a standard library with an acceptable computational expressiveness, it is a declarative language, and it is a part of the OMG's standards.

7. Imperative features

YATL supports several kinds of imperative features that are presented thereafter. This features were selected such that YATL provides lifecycle operations like creation and deletion, operation to change the value of properties, declaration, decision, and iteration statements, native statements to interact to the host machine, for reasons that were previously explained, and build statements to ease the construction of target model instance. Compound statements contain a sequence of instructions, which are to be executed in the given order. These syntactic constructions make use of OCL expressions to specify basic operations such as adding two integer values. YATL uses the same type system as OCL 2.0 [16]. The abstract syntax of the YATL statements is described in Figure 2.

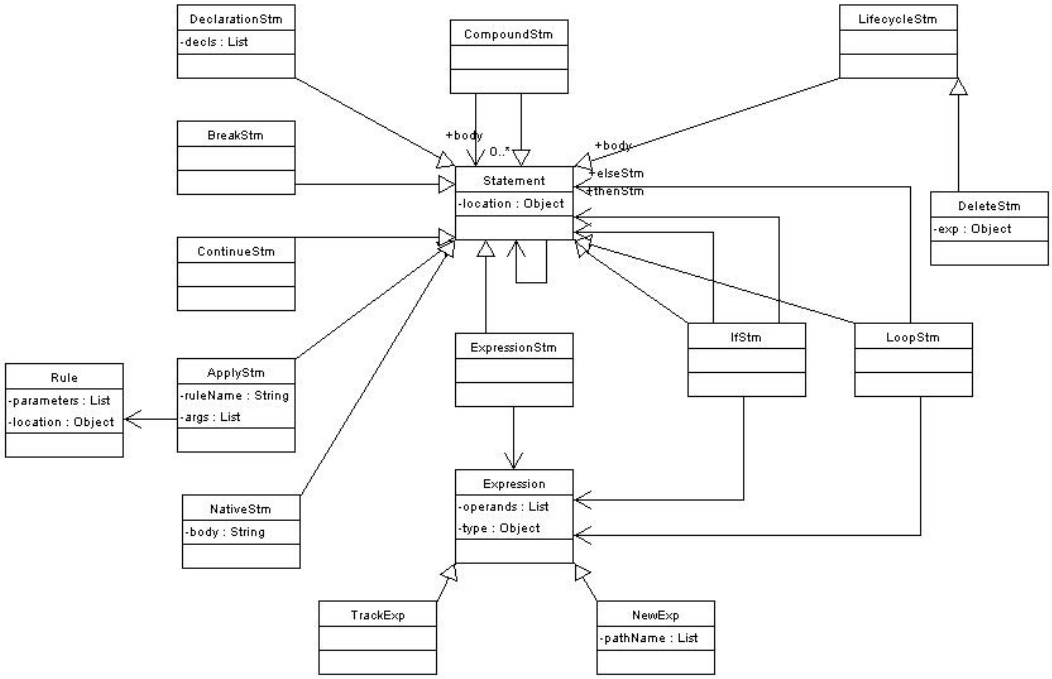

Figure 2 Statements

7.1. Variable declarations

A YATL variable declaration follows the OCL syntax:

let varName : oclType = oclExpression;

YATL variables are typed and must be defined before first use, so YATL is a strongly typed language.

7.2. Assignment statements

All the imperative languages alter the internal state of the associated virtual machine by changing the values attached to variables. The RHS of a transformation rules changes the state of the transformation engine by building parts of the target model instance. The building process requires side-effects, which are not supported by OCL. Hence, a transformation should provide explicit or implicit side-effects. YATL supports assignment statements, which are used to create new instances and bound properties to a given value:

oclExpression1 := oclExpression2

It assigns the value of the right operand to the variable or property given by the left operand.

7.3. Lifecycle instructions

YATL supports explicit creation or destruction of objects in an imperative manner. Instance creation uses the classic new operator:

varName : = *new* pathname;

Deletion is supported by a statement such as:

delete oclExpression;

7.4. Building pattern

In most of the cases, the RHS of a transformation acts like a building pattern. To support such a feature, YATL has a construction called *build* statement. This construction creates a complex object from independent parts that make up the object. For example, if the modeller wants to create an instance of a class Book that has two properties, *name* and *pages*, the following construction

build Book { name = 'test', pages = 5 }

is more convenient then creating the instance, using a new statement, and then setting the values of the properties with assignment statements. If the values of some properties are not specified, the values are set to the undefined value from OCL.

7.5. Conditional statements

OCL provides a conditional construction similar to the conditional operator from Java, C#, C/C++. However, a conditional construction at the level of statements is required. The if statement in YATL has the following concrete syntax:

if condition *then* statement *else* statement *endif*

The semantics of if statement in YATL is similar with the semantics of if statements of imperative languages like Java, C#, and C/C++.

7.6. Loop statements

Loop statements with pre and post condition are supported in YATL:

while condition *do* statement

and

do statement *until* condition

A loop statement iterating over the elements of a collection is also supported:

foreach variableDeclaration *in* oclExpression *do* statement

where the type of *oclExpression* must be an OCL collection type.

7.7. Native statements

The computational support provided by YATL comes mainly from the OCL standard library. However, in the case of large-scale systems additional computational power is required (eg. converting letters from uppercase to lowercase or vice versa). This problem can be solved either by improving the OCL standard library or by providing a mechanism to communicate with the host platform/language. Such additional support can be used from YATL using native statements:

native {

code for the host platform (C#, Java etc)

}

The implementation of this statement is platform dependent. To communicate with the host language/framework, the names of imported/exported variables from/to native block must be prefixed with a distinct symbol (e.g. hash sign). The syntax and semantics errors will be reported by the interpreter/compiler of the host language/framework.

7.8. Expressions

Expressions in YATL are written in OCL and can be used as statements as in Java and C#. The abstract syntax of the expression supported by YATL is described in [2]. Beside the OCL expressions, YATL contains two new kinds of expressions: *new* and *track* expressions; *new* expressions are used to create an

instance of a given type while *track* expressions are used to store and retrieve the mappings at run-time. Track expressions are useful in large scale systems to optimise the runtime and to solve some problems which involve circular processing.

8. Properties of YATL

A YATL transformation is unidirectional. We believe that a model transformation language should be unidirectional, otherwise it cannot be used for large scale models. This happens mainly because a bidirectional transformation language acts like a reasoning machine, using the unification operation to fill in the gaps. However, when a transformation model contains only declarative rules, it should be possible to derive a part of the reverse transformation.

On a real model-to-model transformation, traceability is absolutely necessary to make the approach workable. To trace the mapping between source and target model instances, YATL comprises an operator called *track*. Track expressions are, from the concrete syntax point of view, similar to DSTC's track constructions [11]. The main difference is that YATL's tracks are defined using concepts like relation name, domain, and imagine, and not Prolog-like concepts (e.g. unification). This approach makes the traceability system of YATL suitable for large-scale systems.

During the transformation process the source, target, and transformation model should be navigable. This feature is not supported by the current version of YATL. This is an area of ongoing research.

Current version of YATL does not allow incremental changes in a source model to be transformed into changes in a target model immediately. This is an area of ongoing research.

9. Transforming UML classes to Java classes

We experimented YATL on substantial and representative examples for clarification and validation purposes (UML class diagrams to Java classes, spider diagrams [8] to OCL, and EDOC to BPEL, WSDL, and XSD). Due to lack of space we present here only the classic mapping from UML class diagrams to Java classes. The goal of this experiment is to prove the feasibility of model transformations with YATL. The experiment was performed both under KMF and EMF to illustrate the portability of the YATL implementation. Figure 3 contains a brief description of the transformation.

```
start kmf::uml2java::main;
namespace kmf(uml, javaModel) {
 transformation uml2java {
 -- 1-1 Mappings
 -- Map a UML class to a Java class
 rule umlClass2JavaClass match uml::Foundation::Core::Class () {
  -- Create Java class
  let jClass: javaModel::JavaClass;
  jClass := new javaModel::JavaClass;
  -- Set name
  jClass.name := self.name.body_;
  -- Store mapping
  track(self, class2class, jClass);
 }
 -- Map a UML attribute to a Java field
 rule umlAttribute2JavaField match uml::Foundation::Core::Attribute () {
  . . .
 }
 -- Map a UML association end to a Java field
 rule umlAssociation2JavaField match uml::Foundation::Core::AssociationEnd(){
```

```
  . . .
 }
 -- Map a UML method to a Java operation
 rule umlOperation2JavaMethod match uml::Foundation::Core::Operation () {
  . . .
 }
 -- Link all the fields to the corresponding class
 rule linkAttribute2Class match uml::Foundation::Core::Attribute () {
  -- Get the Java Class that owns the corresponding field
  let umlOwner: uml::Foundation::Core::Classifier, jClass :
javaModel::JavaClass;
  umlOwner := self.owner;
  jClass := track(umlOwner, class2class, null);
  -- Get the Java Field
  let jField: javaModel::JavaField;
  jField := track(self, attribute2field, null);
  -- Link field and class
  jClass.fields := jClass.fields->including(jField);
  jField.class_ := jClass;
 }
 rule linkAssociationEnd2Class match uml::Foundation::Core::AssociationEnd() {
  . . .
 }
 -- Link all the operations to the corresponding class
 rule linkOperation2Class match uml::Foundation::Core::Operation () {
  . . .
 }

 -- main rule
 rule main () {
  -- Map individual elements
  apply umlClass2JavaClass();
  apply umlAttribute2JavaField();
  apply umlAssociationEnd2JavaField();
  apply umlOperation2JavaMethod();
  -- Add fields to Java classes
  apply linkAttribute2Class();
  apply linkAssociationEnd2Class();
  -- Add operations to Java classes
  apply linkOperation2Class();
 }
}
}
```

Figure 3 Transformation Rules

YATL was also used to map spider diagrams [8] to OCL expressions. These experiments proved that YATL is powerful enough to support complex transformation, but the YATL developing environment should be improved to support debugging at runtime.

10. Conclusions and related work

We have learned a lot during this work. The experiments forced us to add new features to YATL and improve the implementation, especially the mapping from spider diagrams to OCL because is not a conventional mapping from a visual language to a textual language.

YATL is still evolving because one of our main goals is to make it complaint to the QVT standard. But we also hope to add many original features to the YATL development environment and to integrate it with KMF and EMF.

Since OMG launched its QVT RFP [10] in 2002, several submissions were made. DSTC's submission [11] contains a declarative definition of QVT and uses high-level concepts that are similar with those from Prolog. Unfortunately it cannot cope with large-scale transformations because its concepts make the implementation very slow. QVT Partners submission [12] considers that transformations are special cases of relations and describes them using a graphical syntax. This approach is in way similar with the one presented in [3]. This submission provides a mechanism for relations' refinement. In the near future YATL will provide a similar support, although it will be described in a textual way. The French submission [13] is very similar with the approach that we took. But, there are a lot of differences such as the concrete syntax, the semantics of the rules, the tracking mechanism, the support for interaction with the host machine and creation of target model instance etc.

Acknowledgement. This work has been funded by the UK EPSRC (Engineering and Physical Sciences Research Council) under grants GR/R63509/01 and GR/R 63516/01.

11. References

[1] Akehurst D., P. Linington, and O. Patrascoiu. OCL 2.0 – Implementing the Standard for Multiple Metamodels. In *OCL2.0-"Industry standard or scientific playground?" - Proceedings of the UML'03 workshop*, page 19. Electronic Notes in Theoretical Computer Science, November 2003.

[2] Akehurst D., P. Linington, and O. Patrascoiu. Technical report, Computer Laboratory, University of Kent, November 2003.

[3] Akehurst D., S. Kent, O. Patrascoiu. A relational approach to defining and implementing transformations between metamodels, SoSym, volume 2, number 4, December 2003, 215-239.

[4] Czarnecki K., S. Helsen. Classification of Model Transformation Approaches, OOPSLA 2003 Workshop: Generative techniques in the context of MDA.

[5] Eclipse Modeling Framework http://www.eclipse.org/emf.

[6] Frankel D. S. Model Driven Architrecture: Applying MDA to Enterprise Computing. John Wiley & Sons, 2003.

[7] Gerber A., M. Lawley, K. Raymond, J. Steel, A. Wood. Transformation: The Missing Link of MDA, in A. Corradini, H. Ehring, H. J. Kreowsky, G. Rozenberg (Eds): Graph Transformation: First International Conference (ICGT 2002)

[8] Gil J., J. Howse, and S. Kent. Formalising Spider Diagrams, Proc. IEEE Symp on Visual Languages (VL99), IEEE Press, 130-137. 1999.

[9] Kent Modelling Framework http://www.cs.kent.ac.uk/projects/kmf

[10] QVT Query/Views/Transformations RFP, OMG Document ad/02-04-10, revised on April 24, 202. http://www.omg.org/cgi-bin/doc?ad/2002-4-10

[11] MOF Query/Views/Transformation, Initial submission, DSTC and IBM.

[12] MOF Query/Views/Transformation, Initial submission, QVT Partners.

[13] MOF Query/Views/Transformation, Initial submission, Alcatel, SoftTeam, Thales, TNI-Valiosys.

[14] MDA Model Driven Architecture http://www.omg.org/mda.

[15] MOF Meta Object Facility http://www.omg.org/mof

[16] OCL Object Constraint Language Specification Revised Submission, Version 1.6, January 6, 2003, OMG document ad/2003-01-07.

[17] OCL http://www.cs.kent.ac.uk/projects/ocl.

[18] UML Unified Modeling Language http://www.omg.org/uml.