Transformation Language Design: A Metamodelling Foundation

Tony Clark, Andy Evans, Paul Sammut, and James Willans

Xactium Limited
andy.evans@xactium.com

1 Introduction

With the advent of the Model-Driven Architecture (MDA) [3] there is significant interest in the development and application of transformation languages. MDA recognises that systems typically consist of multiple models (possibly expressed in different modelling languages and at different levels of abstraction) that are precisely related. The relationship between these different models can be described by transformations (or mappings).

An important emerging standard for transformations is QVT (Queries, Views, Transformations). This standard, being developed by the Object Management Group (OMG), aims to provide a language for expressing transformations between models that are instances of the MOF (Meta Object Facility) [2] metamodel. The MOF is a standard language for expressing meta-data that is being used as the foundation for expressing language metamodels (models of languages). Because of the generic nature of MOF, it is also being used as the means of expressing the QVT language itself.

When the QVT process began (over two years ago), the task initially seemed quite straightforward. After all, many different transformation languages were already described in the literature, and it was felt that it would be straightforward to design such a language for MOF. Unfortunately, this has not been the case. Two key issues have made the task of designing such a language much harder. In the remainder of this paper we will examine these issues and propose a solution that is applicable across a wide variety of language definitions.

2 Design Issues

The first issue that impacts the design of a transformation language relates to transformation languages themselves. In practise, it turns out that there are many different flavours of transformation languages. Some of the choices of language features include:

– *Declarative vs. Imperative*: at what level of abstraction should transformations be expressed? Declarative languages enable transformations to be expressed in a more concise fashion, yet may suffer from being inefficient (or impossible) to implement.

H. Ehrig et al. (Eds.): ICGT 2004, LNCS 3256, pp. 13–21, 2004.
© Springer-Verlag Berlin Heidelberg 2004

- *Compositionality*: whilst a transformation language should ideally be compositional, this is more readily achieved by the use of more declarative primitives, thus invoking the declarative vs imperative issue (above).
- *Patterns vs. Actions*: patterns are widely used as a declarative but executable abstraction for describing transformations (XSLT is a good example of this). Yet, should a transformation language be completely pattern based, or should a mixed language with imperative actions permitted for practicality?
- *Unidirectional vs. Bi-directional*: it is clear that there is a strong distinction between one-stop, unidirectional transformations, and bi-directional mappings that keep two models in sync. Should both be accommodated?

These, and many other choices make the decision process a difficult one for the designers of transformation languages. One approach to tackling the problem is to attempt to mudpack all of the different features into a single language. However, there is clearly a danger of producing an overly complex language. On the other hand, choosing a subset of the features will clearly omit use cases of the language that may be relevant to users of the language.

The second issue relates to MOF itself. During the QVT process, it has become more apparent than ever that current metamodelling practice is too weak. In particular, the standard approach to metamodelling, in which the main focus is on capturing the static properties of a language (i.e. the abstract syntax) does not enable two critical aspects of language design to be expressed: *semantics* (what the language does and means) and *concrete syntax* (how the language is represented). Thus, in order to describe these aspects, the design team must rely on informal textual descriptions or bespoke implementations. In the latter case, this often results in 'analysis paralysis', as there is insufficient information to validate the correctness of the design.

Clearly, in the context of an international standardisation process, this is not satisfactory. In particular, it will be difficult to ensure that implementations of the standard are conformant as there will be gaps in the definition that will be filled in by vendors in different ways.

3 The Way Forward

In order to fully address the needs of QVT and transformation language design in general, it is clear to us that two key changes are required. Firstly, the bar must be raised in the way in which we metamodel languages. Rather than just capturing abstract syntax, the metamodelling language must be rich enough to capture *all* aspects of a language, including concrete syntax, abstract syntax and semantics. This information should be sufficient to rapidly generate tools that implement the language and allow its properties to be fully explored and validated.

Secondly, it must be recognised that there is no single, all encompassing transformation language. Instead we must be prepared to embrace a diversity of languages, each with specific features. Furthermore, the standard must have the flexibility to accommodate this diversity in an interoperable manner, enabling different features to be mixed and matched as required.

At first, these proposals appear to be unconnected. Yet, they are in fact closely related. In practice, we have found that the richer the capability for expressing metamodels, the greater the flexibility and interoperability of the resulting language designs. This occurs because the metamodels capture cohesive language units that can be readily integrated within other languages.

4 XMF

We have constructed an approach (and associated tools) for language metamodelling that aims to realise these goals. This approach is based on what we call an eXecutable Metamodelling Framework (XMF). The basic philosophy behind XMF is that many different languages can be fully described via a metamodelling architecture that supports the following:

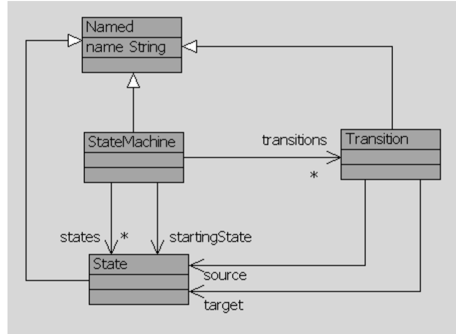
- A platform independent virtual machine for executing metamodels.
- A small, precise, executable metamodelling language that is bootstrapped independently of any implementation technology. This supports a generic parsing and diagramming language, a compiler and interpreter, and a collection of core executable MOF modelling primitives called XOCL (eXecutable OCL).
- A layered language definition architecture, in which increasingly richer languages and development technologies are defined in terms of more primitive languages via operational definitions of their semantics or via compilation to more primitive concepts.
- Support for the rapid deployment of metamodels into working tools. This involves linking executing metamodels with appropriate user-interface technology.

Using this architecture we have implemented many different modelling languages and development technologies for industrial clients. We have used exactly the same approach in the definition of transformation languages. Firstly, some core transformation language abstractions were implemented. These included a pattern matching language and synchronisation language. Two transformation languages were then defined on top of these. The first, XMap, provides a language for generative transformations based on pattern matching. XOCL is integrated in the language, thus enabling mixed declarative and imperative mappings. The second, XSync, supports the dynamic, bi-directional synchronisation of models, this time using XOCL as a means of writing the synchronisation rules.

In the following sections, we firstly give an example of one the languages, XMap, and then describe how the language is defined using a metamodel.

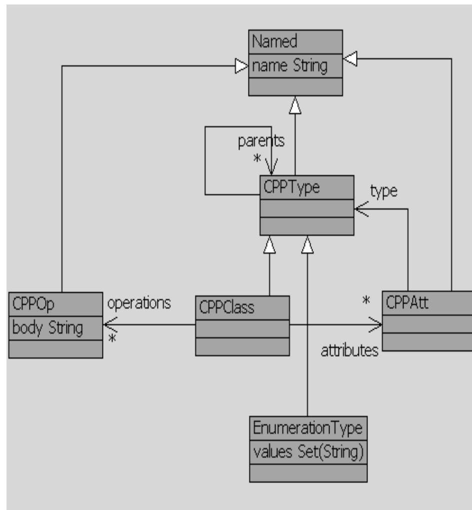
5 XMap Example

The example defines a mapping between two models: a simple model of state machines, and a simple model of C++. The simple state machines model is shown below:



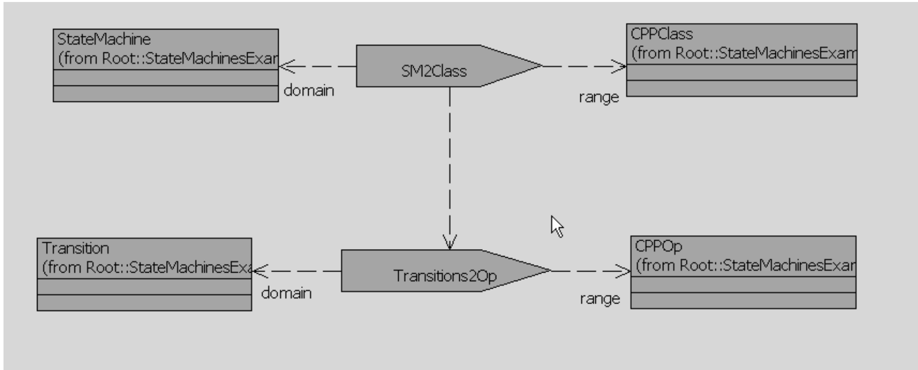
Both states and transitions are labelled with a name. A transition relates a source state to a target state.

The following model captures the basic features of C++. Note that the body of an operation is a string. However, if necessary the expressions and statements could also be modelled.



A mapping from the StateMachine model to the C++ model can now be written. It maps a StateMachine to a C++ class, where each state in the state machine is mapped to a value in an enumerated type called STATE. Each transition in the state machine is mapped to a C++ operation with the same name and a body, which changes the state attribute to the target of the transition.

The mapping can be modelled in XMap as shown below. The arrows represent mappings between elements of the two languages. The first mapping, SM2Class, maps a state machine to a C++ class. The second mapping, Transition2Op, maps a transition to an operation.



In order to describe the details of the mapping, XMap uses a textual mapping language based on pattern matching. As an example, the definition of the mapping between a transition and an operation is as follows:

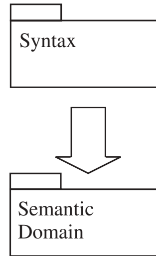
```
context Transition2Op
  @Clause Transition2Op
    Transition
      [name = N,
      target = T]
    do
    CPPOp
      [name = N,
      body = B]
    where
      B = "state = " + T.name
end
```

A mapping consists of a collection of clauses, which are pattern matches between source and target objects. Whenever a source object is successfully matched to the input of the mapping, the resulting object in the do expression is generated. Variables can be used within clauses, and matched against values of slots in objects. Because XMap builds on XOCL, XOCL expressions can be used to capture complex relationships between variables.

In this example, whenever the mapping is given a Transition with a name equal to the variable N and a target equal to T, it will generate an instance of the class Operation, whose name is equal to N and whose body is equal to the variable B. The where clause is used to define values of variables, and it is used here to define the variable B to be concatenation of the text “state = ” with the target state name. For instance, given a transition between the states “On” and “Off”, the resulting operation will have the body “state = Off”.

6 XMap Metamodel

The architecture of the XMap language metamodel is described in the figure below.



The syntax metamodel describes how the metamodel captures concrete syntax representation of the mapping language. There are three ways this can be achieved:

– A textual syntax can be defined for the language by constructing a grammar that states how a textual representation of models written in XMap can be parsed into an instance of the concepts represented in the syntax definition. This is achieved in XMF using XBNF: an extended BNF grammar language that provides information about how to turn textual elements into instance of XMF elements.
– A graphical syntax can be defined by defining a mapping from a model of the graphical syntax of the language to concepts in the syntax domain.
– A mixed approach can be used, in which both graphical and textual elements are defined. This is the approach taken with the XMap language.

6.1 Textual Syntax Metamodel

As an example a textual syntax metamodel, the following fragment of XBNF defines the rules for parsing a clause into an instance of a Clause class:

```
@Class Clause
 @Grammar extends OCL::OCL.grammar
   Clause ::= name = Name
         patterns = ClausePatterns 'do' body = Exp {
                   Clause(name,patterns,body) }.
   ClausePatterns ::= p = Pattern
         ps = (',' Pattern)* { Seq{p | ps } }.
   ClauseBindings :: 'where' Bindings | {Seq{}}.
 end
      ...
end
```

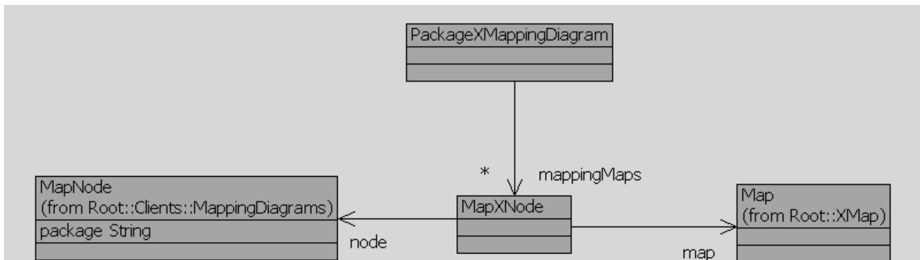
The grammar extends the OCL grammar with the concept of a Clause, where a Clause is defined to be a name, followed by a collection of patterns, followed by a 'do' and a body, which can be an expression, and an optional collection of 'where' bindings. The result of matching any textual input of this form:

```
@Clause <name>
   <patterns>
   do
   <body>
   where
   <bindings>
end
```

is to create an instance of a Clause(), passing it the definitions of name, patterns and bindings. Of course, further XBNF definitions will be needed that define the grammar rules for Pattern, Binding, etc. These are omitted for brevity.

6.2 Diagram Syntax Metamodel

As an example of a metamodel of diagrammatical syntax, the following diagram describes a part of the syntax of XMap mapping diagrams. A mapping diagram extends a class diagram with MapNodes (uni-directional arrows).



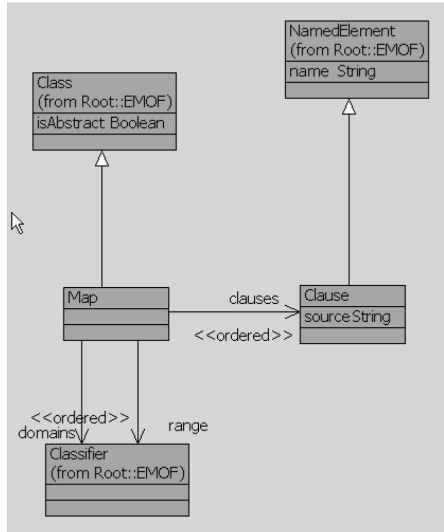
The relationship between a MapNode and a mapping (denoted here by the class Map) is kept constantly in step via the MapXNode mapping. This body of this mapping is written in XSync, thus synchronising the relevant aspect of the two elements. For instance, the name of the Map and the MapNode must always be kept in step.

6.3 Semantic Domain

The syntax of the language can be viewed as a syntactical sugar for concepts in the semantic domain. A semantic domain expresses the meaning of the concepts in terms of more primitive, but well-defined concepts. A semantics is thus defined for XMap via a translation from the syntactical representation of a mapping into a semantic domain model.

The semantic domain model for the XMap language is described by the following model:

Here, a Map denotes a mapping. It is a subclass of the class Class and therefore inherits all the properties of the class Class. It can therefore be instantiated



and define operations that can be executed. In addition, a Map has a domain and range, and a sequence of clauses.

The translation step that is performed is to translate each clause of a mapping into a case statement belonging to a distinguished operation of the class Map. Each case statement can contain an XOCL expression. Note that in this case, XOCL has also been extended with patterns, thus enabling values to be matched against other values.

As an example, consider a mapping with the following clauses:

```
@Clause Clause0
  Transition
    [name = ""]
  do
  CPPOp
    [name = "Empty"]
end

@Clause Clause1
  Transition
    [name = N]
  do
  CPPOp
    [name = N]
end
```

This would be translated into the following operation of the class Map:

```
@Operation invoke():Element
  @Case of
```

```
      Transition[name = N] do
        CPPOp[name = N]
      end
      Transition[name = [ "" ]] do
        CPPOp[name = "Empty"]
      end
      else self.error("Mapping failed.")
   end
end
```

Running this operation will thus execute the appropriate case statements and perform the mapping.

The way in which the translation from syntax to semantic domain occurs is a matter of choice. At the diagram level, the desugaring is maintained by the synchronised mapping between the diagram and the semantic domain model. At the syntax level, a desugar() operation can be added to the grammar to tell it how to construct the appropriate case statement.

7 Conclusion

Our approach to defining transformation languages is to use rich metamodels to capture all aspects of their definition, including syntax and semantics. A key property is that definitions of existing languages and technologies (such as XOCL) can be merged in with the new language, creating richer, more expressive capabilities.

The result is a precise definition that is: platform independent (no reliance on external technology), transparent (the entire definition, including its semantics can be traced back through the metamodel architecture); extensible and interoperable (new features can be added by adding new language components), and executable (enabling the language to be tested and validated).

There has been much recent interest in the design of domain specific languages [1], and the approach described in this paper offers a scalable solution to the problem of how to generate new languages and tools that support those languages in a generic fashion.

In summary, our position is that a crucial step in the design of transformations languages must be the adoption of more complete and semantically rich approaches to metamodelling.

References

1. S. Cook. Domain-specific modeling and model driven architecture. *MDA Journal*, January 2004. `http://www.bptrends.com/publicationfiles/01-04%20COL%20Dom%20Spec%20Modeling%20Frankel-Cook.pdf`.
2. Object Management Group. Meta-object facility. `http://www.omg.org/mof`.
3. Object Management Group. Model driven architecture. `http://www.omg.org/mda`.