A Calculus For QVT

Tony Clark
Xactium Ltd.
tony.clark@xactium.com

April 6, 2004

1 Introduction

In 2002 the Object Management Group issued a request for proposals for a Queries View and Transformations standard [1]. The QVT Standard aims to provide languages for querying models, presenting views of models and transforming models. This is widely seen as being enabling technology for the OMG flagship Model Drive Architecture initiative (see [2] for a good overview of MDA) that aims to reify many aspects of the development process to the modelling level and to provide automatic and semi-automatic technologies for transforming models between different levels of abstraction (for example from UML to program code).

The OMG QVT RFP aims to define a standard for expressing queries, views and transformations in MOF [10] (and by implication in UML). A number of proposals have been submitted against this RFP. One of the strongest technical submissions has been delivered by TCS supported by King's College London and The University of York [9]. A common theme has emerged where *transformations* are used as the underlying technology. Model transformations involve three key activities:

- *Selecting* components from source models. The selection will involve type checking, structural tests and calculating properties of the source models.
- *Constructing* and populating new models to form the target of the transformation.
- *Modifying* the source model so that it is used to form part of the target model.

We propose that all model transformations can be defined using these three activities. Our aim is to precisely define a *QVT calculus* which is a universal language for defining model transformations. The calculus strives to exhibit the following features:

- It is *declarative* with respect to the mechanical steps necessary to achieve transformations defined in the calculus. In other words, we strive to distance ourselves from viewing the calculus as a standard imperative programming language.
- It is *compositional* such that transformations can be constructed from component transformations.
- It is *precise* in the sense that it has a formal mathematical semantics although access to the semantics is not necessary in order to use the calculus.

The calculus is intended to be minimal in the sense that there is no redundancy of features. Richer languages are intended to be constructed as *sugarings* of the lower-level calculi. In this way the QVT calculus supports a family of QVT languages.

This report provides an overview of the QVT calculus. The report provides the basis for an *infrastructure* for QVT and is structured as follows: section 4 establishes the relationship between the calculus and UML-style modelling languages; section 5 describes the basic values denoted by the expressions in the calculus; sections 6 and 7 make the distinction between predicative relationships between model elements and mappings that constructively transform input model elements into output model elements. Collectively, we refer to mappings and relations as *transformations*; the formal syntax and semantics of transformations are described in sections 8 and 9. Section 11 describes how the calculus can sugared by introduceing higher-level language constructs that encapsulate recurring patterns. The intention is that the calculus can provide the basis for a development method whereby transformations are specified using relations and then implemented as mappings; section 13 outlines how this might be achieved and forms the basis of future work defining a refinement calculus for QVT. The QVT RFP requires any proposal for QVT to be a practical solution to the transformation of models. Appendix A defines the formal semantics of the QVT calculus in terms of a state transition relation. The state transition machine is intended to form the basis of a reference implementation for QVT.

2 Scenarios for QVT

This section discusses the likely scenarios where QVT technology will be required to support Software Engineering activities. The QVT calculus is not intended to directly provide technology for these activities; it is intended to provide a model within which such technologies can be analysed and developed.

Suppose we have a transformation T that holds between 2 or 3 models. In a real situation, each model will be managed by a tool. The models are managed by a central repository although the tools may have their own local copies, in part or whole, of the model data. In many cases there will be just 2 models (for example a UML model and a C++ program); however, there will be other cases where 3 or more models are related by the transformation. A third model may encode some information about platform data that makes choices between different styles of encoding or represents product specific information in a product line.

QVT should support some or all of the following scenarios:

- The models represent independently constructed and maintained overlapping views of the same logical information. For example, a UML model and its equivalent in a telecomms standard. The transformation T should be able report on the consistency of the two views. Note that it is not sufficient to report that the two views are not consistent; the report must clearly state, in a standard way, the details of why the models are inconsistent

- Model1 is developed and debugged; it is platform independent. Model2 is a platform specific model (for example a model specific to J2EE). T must be able to express the rules of transformation that turn Model1 into Model2. Note that the transformation may be arbitrarily complicated.

- Model1 is developed, T is used to turn Model1 into Model1 (for example code generation). It is possible that the transformation is computationally intensive, e.g. a large model is turned into a vast amount of code. Assume that Model1 is then modified frequently. It is desirable to propagate only the changes to Model1, or deltas, and apply them to the appropriate elements of Model2. A standard model of deltas will also provide scope for efficiently managing versions where many views on the same model change frequently.

- Model1 is developed independently to Model2 and they contain overlapping information. For example, Model1 is a UML model of program data and Model2 is the (legacy) program. We would like to be able to fix either Model1 or Model2 and then request the collection of deltas that must be applied to the other model in order to make them consistent.

- Model1 and Model2 overlap in some regard and Model3 represents the merge of Model1 and Model2. This will occur frequently on projects where different project components are developed by independent teams. The transformation T must be able to state when the three models are consistent. Similarly, a transformation T can be constructed that will perform the merge when this is possible.

- A tool develops Model1 and transforms it to Model2 which is then managed independently by a second tool. Changes applied to either model should be propagated in a standard way to the other tool, thereby ensuring that the tools maintain the models in step. This is an important feature of QVT technologies that require them to be reactive and to support standard ways of publishing deltas.

- A transformation is published in a repository of transformation for use by other tools. Transformations are accessed in a repository and then used in the definition of new transformations. Existing transformation, written in arbitrary implementation languages and using arbitrary technologies may be used as part of new, standard, transformation definitions.

- Transformation definition is performed by engineers whose domain of expertise is not necessarily modelling. The transformation definition language is tailored to their specific expertise.

The scenarios above indicate two different modes of QVT: *checking* and *performing*. These two activities are different since checking produces a boolean result (hopefully backed up with a reason represented in a standard way) and performing involves modification and construction of new model elements. Checking represents the specification of performing and, for many larger scenarios, will be significantly less effort than producing the actions necessary to perform a transformation. In some scenarios checking is sufficient: a tool that supports QVT simply needs to know when and why two things are no longer in sync. In other scenarios, performing is important since we want to generate one model from another. In one MDA method, the engineer will specify a transformation (in a standard way) before defining its implementation (in a standard way).

3 Requirements on QVT Technologies

The scenarios lead to the following requirements on any technology used to support QVT:

- QVT technology should be language, platform and technology neutral. The language used to define transformations should be defined in the abstract but no concrete syntax should be mandated. Where appropriate guidelines for defining concrete syntaxes should be provided. A non-mandatory but normative concrete syntax for transformation diagrams and text may be provided as this will be very useful for expressing examples.

- The QVT architecture should be open. It should allow legacy transformations to be plugged in, used and distributed in a standard way. Non-standard architectures for QVT should be supported as plugins to allow tool vendors with existing technologies to work alongside new tools that implement the standard.

- The QVT standard should be defined in such a way as to clearly present the semantics of transformations. Where it is appropriate, it is desirable that the semantics of executable parts of the QVT standard are operational.

- The QVT standard will use standard notations for transformations where these are available and appropriate. For example, a transformation should be an extension to the MOF model and OCL can be used to specify transformations. Care should be taken not to use standards where they are not appropriate.

- QVT should be capable of supporting transformation (specification and implementation) of arbitrary complexity. The majority of transformations will not be structure to structure transformations; they will involve complex algorithms that construct or check the target models.
- Transformations should be modelled, specializable and defined in a compositional manner.

4 Models

QVT must support transformations that apply at all levels of the *golden braid*: meta-level; model-level and instance-level. It is undesirable to have more than one representation for such models; in fact it is very desirable to have a single representation since this allows general libraries of transformations to be constructed without committing to a particular level of application within the golden braid.

We require a sufficiently expressive representation for models which spans all 3 levels of the golden braid[1]. We note that other languages which achieve a simple universal representation of this kind (for example XML, Lisp) encode data in terms of a type and a sequence of component data.

The modelling equivalent of this universal representation is the UML *Object Diagram*. Given the meta-model of a model, it is always possible to represent a model as an object diagram. Ambiguity arises in UML object diagrams regarding whether or not the diagram *is* the entity under study or whether it is a *view* of the entity under study. In addition, ambiguity often arises regarding how multiplicities are dealt with: are the links from an object complete?

We represent all model information at the level of object diagrams. An object has a type, an identity and a collection of slots (which are named values). We deal with the problem of whether we are dealing with the real thing or just a view by making a clear distinction between *the model* which is a complete description and *views* of the model as encoded by model patterns. We deal with the multiplicity of links issue by forcing the explicit representation of collections as sets: either something is a single element or it is a set of elements.

[1]Note addressing 3 levels is sufficient because more levels can always be viewed as several occurrences of the golden braid linked together

As a simple example consider the following model:

```
  -----
  | P |
-----------------------------------------------------------------
|   --------------------                  ------------------    |
|   |       Person     |               *  |      Pet        |   |
|   -------------------- --------------> -------------------    |
|   |     name:String  |                  | age:Integer     |   |
|   --------------------                  -------------------   |
-----------------------------------------------------------------
```

We can represent this as an object diagram in textual form:

```
(Package) [
  name = 'P',
  contents = {
    (Class) [
      name = 'Person',
      attributes = {
        (Attribute) [
          name = 'name',
          type = String],
        (Attribute) [
          name = 'pet',
          type = (SetType) [
            elementType = Pet]]}],
   (Class,Pet) [
     name = 'Pet',
     attributes = {
       (Attribute) [
         name = 'age',
         type = Integer]}]}]
```

5 Values

To make the distinction between models that are the entity under study and models that are used to represent a view of the entity under study we define a universe of *values* that represent the entities to be transformed. They are essentially object diagrams. Consider a dog with an age 10:

```
(Dog)[age = 10]
```

Object diagrams are graphs therefore we require a mechanism for representing sharing in values. An object has a unique identity. Here is a dog that refers to itself:

```
(Dog,dino)[age = 10,chasesTailOf = dino]
```

The QVT calculus does not make a distinction between atomic values and objects - they are represented as objects with a given type and whose identity is the value itself. Therefore `10` is equivalent to `(Integer,10)[]`.

We often need to talk of collections of values. A value may also be a set of values. A set does not have an identity, it is a *stateless* container of values. A *stateful* container can be created by wrapping a set with an object. We may wish to have a person with one pet:

```
{(Person) [
   name = 'Fred',
   pets = {
     (Dog,dino) [
       age = 10,
       chasesTailOf = dino]}],
 dino}
```

which is equivalent to:

```
 {(Person) [
   name = 'Fred',
   pets = {dino}],
  (Dog,dino) [
     age = 10,
     chasesTailOf = dino])]}
```

The syntax of values is defined as follows:

$$
\begin{array}{lllll}
X & ::= & & & \text{values} \\
 & & A & & \text{atoms} \\
 & | & \{X_i^{i \in 0 \ldots n}\} & & \text{sets}, P, Q \\
 & | & (X, X)[V_i = X_i{}^{i \in 0 \ldots n}] & & \text{objects}, o \\
 & | & <P^*, \rho, E> & & \text{relations}, r \\
 & | & <P^*, \rho, E, E> & & \text{mappings}, m
\end{array}
$$

6 Relations

QVT should support both transformation *specification* and transformation *implementation* using standard notation and semantics. A *transformation specification* is a definition of a relation. This section describes the QVT calculus representation for relations.

A relation is a set of n-tuples. For transformations we may (but need not) think of a binary relation as having a source model and a target model. In this case the relation specifies the legal outputs which must be produced for any legal input to the transformation.

Often, a translation will consume a source model and produce an output model without any requirement that the source model is changed. For example, we might translate objects to the Java commands that create the objects:

```
(Dog) [ age = a ] <=>
(New) [                          // Object equivalent of:
  class = 'Dog',                 // new Dog(a);
  args = {                       // in Java.
    (Arg) [
      position = 1,
      value = a]}]
```

Suppose that we want to specify a transformation that sorts elements in a dictionary using some suitable definition of $<$ and $=$. The following relation produces a sorted dictionary. Each element in the dictionary has a `data` slot that contains the dictionary value and a `position` slot that contains its position in the dictionary. An invariant on the model requires that all positions are different and in the range defined by the size of the dictionary. Initially the positions are random:

```
(Dict) [ elements = P ] <=> (Dict) [ elements = Q ]
  when
    P->collect(p | p.data) = Q->collect(q | q.data) and
    Q->forAll(q |
      Q->forAll(q' |
        q.data <= q'.data implies q.position <= q'.position)
```

Relations may be named and used in the definition of other relations (even themselves). A relation can be used in two different ways: in a condition or in an expression. If it is used in a condition then it is supplied with all the required arguments and it is `true` with the relation holds for the argument values otherwise it is `false`.

If a relation is used in an expression then it must be supplied with 1 less than the required number of arguments. In this case it is being used as a function to calculate the value of the missing argument. Note however, that the function may be undefined for the given arguments or it may be underspecified in which case there could be more than one value for the supplied arguments. Since we are defining a relation then partial or non-deterministic functions are permitted. It should be noted that relations used as functions are simply sugar for exclusive, but more verbose, use of relations in conditions.

Suppose that we want to specify a transformation on a class model consisting of packages, class and attributes. To make the example simple, suppose we are just calculating all the names of in the model.

```
PNAMES =  // Calculate the names in a package.
  (Package)[ name = n, classes = C ] <=>
   adjoin(n,C->iterate(c S = {} | union(S,CNAMES(c)))
CNAMES = // Calculate the names in a class.
```

```
  (Class) [ name = n, attributes = A ] <=>
    adjoin(n,A->collect(a | ANAME(a)))
ANAME =  // Produce the name of an attribute.
  (Attribute) [ name = n ] <=> {n}
```

Relations may be combined using disjunction. Consider the task of transforming a package by adding a prefix to every name in the package. A package has `contents` which are a mixture of classes and packages:

```
CHANGENAME = CHANGEPNAME + CHANGECNAME
  where
    CHANGEPNAME =
      (Package,p) [ name = n, contents = C ]
      <=>
      prefix
      <=>
      (Package,p) [ name = prefix + n, contents = C' ]
        where C' = C->collect(c | CHANGENAME(c))
    CHANGECNAME =
      (Class,c) [ name = n ]
      <=>
      prefix
      <=>
      (Class,c) [ name = prefix + n ]
```

The definition of `CHANGENAME` above has two component relations: `CHANGEPNAME` and `CHANGECNAME`. The composition of these relations is performed using the relation combinator +. The combinator is to be viewed as defining a resulting relation using the combination of both the component relations.

A typical use of + occurs when defining an inductive relation over a set of elements. In such cases there is usually a base case (such as the empty set) and various inductive steps. For example, consider calculating the number of elements in a set[2]:

```
SIZE = SIZEEMPTY + SIZENONEMPTY
  where
    SIZENONEMPTY = { x | S } <=> size + 1 where SIZE(S,size)
    SIZEEMPTY = {} <=> 0
```

We often wish to define relations as a combination of component relations that address different concerns or *aspects*. For example we may have a transformation that defines how to implement a web front end for a model and a transformation that defines how to implement the database connectivity for the same model. The calculus provides a combinator for constructing relations from multiple aspect relationships.

[2]this task can also be defined using `iterate`, however iterate becomes unwieldy for tasks above a certain size

For example, consider defining a relationship between class models and Java classes where we wish to separate the definition of Java class fields from the Java methods that access and update them:

```
JAVA = FIELDS x METHODS
  where
    FIELDS =
      (Class) [
        name = n,
        attributes = A ]
      <=>
      (JavaClass) [
        name = n,
        fields = F ]
      where
        F = A->collect(a | FIELD(a))
        FIELD = ...
    METHODS =
      (Class) [
        name = n,
        attributes = A ]
      <=>
      (JavaClass) [
        name = n,
        methods = { | G,S,M} ]
      where
        G = A->collect(a | ACCESSOR(a))
        S = A->collect(a | UPDATER(a))
        ACCESSOR = ...
        UPDATER = ...
```

Any UML class and Java class related by `JAVA` must satisfy both the component relations `FIELDS` and `METHODS`.

Consider a simple transformation that increments the age of a dog:

```
(Dog,dino) [ age = a ] <=> (Dog,dino) [ age = a + 1 ]
```

The relation specifies that the transformation is legal it holds between two dogs, that the dog is the same before and after and that the age after is greater by 1 than the age before. Note that the object identity `dino` is important because this forces the state of a single dog to change. The following relation need not refer to the same dog in both cases:

```
(Dog) [ age = a ] <=> (Dog) [ age = a + 1 ]
```

Objects may be nested as in the following that changes the age of a pet:

```
(Person,fred) [
```

```
 pet = {
    (Dog,dino) [
       age = a ]}]
<=>
(Person,fred) ]
  pet = {
    (Dog,dino) [
       age = a + 1 ]}]
```

7 Mappings

Mappings are a restricted form of relation such that a choice has been made regarding which are the inputs to the relation and which are the outputs. It is often the case that relations are easier to define than mappings (think of the dictionary sort example given in an earlier section). In this way relations can be viewed as a transformation *specification* and mappings as the corresponding *implementation*.

The QVT calculus allows relations and mappings to be mixed and therefore specifications can be refined to implementations in an incremental manner. The calculus provides a syntax for mappings that is similar to that for relations but which can be executed.

Consider the following relation:

```
(Dog,dino) [ age = a ] <=> (Dog,dino) [ age = a + 1 ]
```

which may be interpreted either as reducing the age of dino or increasing the age of dino. If the relation is refined to a mapping then the interpretation is made clear:

```
(Dog,dino) [ age = a ] => (Dog,dino) [ age = a + 1 ]
```

or alternatively:

```
(Dog,dino) [ age = a ] => (Dog,dino) [ age = a - 1 ]
```

The sorting example can be implemented as follows:

```
SORT = SORTEMPTY + SORTSTEP
  where
    SORTEMPTY = (Dict) [ elements = {} ] => (Dict) [ elements = {} ]
    SORTSTEP =
      (Dict) [ elements = { (Element) [ data = d ] | E }
        when not E->exists(e | e.data < d)
      =>
      (Dict) [
        elements = {
          (Element) [ data = d, position = 0 ] | INCREMENTALL(E') } ]
            where
              (Dict) [ elements = E' ] = SORT((Dict) [ elements = E ])
```

8 Transformations

The syntax of transformations is defined as follows:

$$
\begin{array}{lcll}
T & ::= & & \text{transformations} \\
 & & P(\Leftrightarrow P)* \ \textbf{when}\ C & \text{relations} \\
 & & P* \ \textbf{when}\ C \Rightarrow E & \text{mappings} \\
 & | & T\ \textbf{where}\ B* & \text{local definitions} \\
 & | & T + T & \text{disjunction} \\
 & | & T \times T & \text{conjunction} \\
 & | & T \diamond T & \text{join} \\
 & | & V & \text{named transformations} \\
 & & & \\
P & ::= & & \text{patterns} \\
 & & V & \text{variables} \\
 & | & \{P_i^{i \in 0 \ldots n} \mid P_i^{i \in 0 \ldots m}\} & \text{set patterns} \\
 & | & (C[,V])[V_i = P_i{}^{i \in 0 \ldots n}] & \text{object patterns} \\
 & & & \\
C & ::= & & \text{constraints} \\
 & & V(E_i^{i \in 0 \ldots n}) & \text{relations} \\
 & | & C\ \textbf{and}\ C & \text{conjunction} \\
 & | & C\ \textbf{or}\ C & \text{disjunction} \\
 & | & \textbf{not}\ C & \text{negation} \\
 & & E \rightarrow Q(V \mid C) & \text{quantification} \\
 & & & \\
E & ::= & & \text{expressions} \\
 & & V(E_i^{i \in 0 \ldots n}) & \text{function application} \\
 & | & E \rightarrow M(V \mid E) & \text{mapping and reduction} \\
 & | & E \rightarrow \text{iterate}(V\ V = E \mid E) & \text{iteration} \\
 & | & (C[,V])[V_i = E_i{}^{i \in 0 \ldots n}] & \text{object expressions} \\
 & | & E.V & \text{slot reference} \\
 & | & \{E_i^{i \in 0 \ldots n} \mid E_i^{i \in 0 \ldots m}\} & \text{set expressions} \\
 & | & V & \text{variable values} \\
 & & & \\
B & ::= & P = E \mid V = T & \text{bindings}
\end{array}
$$

9 Semantics

The semantics of the calculus is given in appendix A. The semantics is operational and is defined by a QVT machine. The machine performs relationship checking (given all relationship arguments) and performs mappings. The machine performs full backtracking when non-determinism arises due to selecting elements from sets.

Given a QVT expression e with respect to a collection of definitions ρ, the machine performs a non-deterministic evaluation:

$$([], \rho, [e], \epsilon, \delta) \longrightarrow^* ([v], \rho, [], f, \delta)$$

where v is the outcome of evaluating e.

10 Application: Serialization

Consider the problem of specifying a relationship between static UML models and XMI. The structure of the model is similar to the structure of the XMI except that types occur as identifier references in the XMI and occur as links to the appropriate class in the class model.

Consider the task of relating classes to attributes. The tertiary relation `MAPCLASS` relates elements, with an id table and a class. The id table is used to keep track of associations between XML id references and the objects that they refer to:

```
MAPCLASS =
  (Element)[
     tag = 'Class',
     attributes = {
       (Att)[name = 'name',value = n],
       (Att)[name = 'id', value = i] | _},
     children = C]
  <=>
  {(i,c) | T}
  <=>
  (Class,c)[name = n, attributes = A]
  where MAPATTS(C,{(i,c)|T},A)
```

The relation `MAPATTS` is used to relate sets of element children (possibly containing elements with attribute tags) to attributes in the static UML model. It can be defined as follows:

```
MAPPATS1 =
  {(Element)[
    tag = 'Attribute',
    attributes = {
      (Att)[name ='name', value = n],
      (Att)[name = 'type', value = tid]}] | E}
  <=>
  {(tid,t) | T}
  <=>
  {(Attribute)[name = n, type = t] | A}
  where MAPATTS(E,{(tid,t) | T},A)

MAPATTS2 =
  {(Element)[tag = t] | E} <=> T <=> A
     when MAPATTS(E,T,A) and t <> 'Attribute'

MAPATTS = MAPATTS1 + MAPATTS2
```

11 Syntactic Sugar

The QVT calculus is intended to be the base language for a family of QVT languages. We aim to make the QVT calculus simple and expressive enough to support any rich transformation language. This section provides a simple example of how a new language construct can be defined and implemented by a transformation to the basic calculus.

The definition of `MAPATTS` in the preceding section identifies a recurring patttern that maps over a pair of sets. Consider the following sugar:

```
MapSet ::= mapSet Case* end
Case     ::= case Pattern (P*) Pattern when C end
Pattern  ::= P | none
```

The meaning of a mapset is a disjoint collection of mappings each of which selects values from sets based on the patterns. If either the left or right pattern is `none` then no selection takes place. Therefore:

```
mapSet c1 c2 ... cn end
```

is equivalent to:

```
M1 = r1
M2 = r2
...
Mn = rn
M = M1 + M2 + ... + Mn
```

where a case:

```
case p (a) q when c end
```

becomes a relation:

```
{p | P} <=> (a) <=> {q | Q} when c and M(P,(a),Q)
```

If any pattern is `none` then no selection takes place for example:

```
case p (a) none end when c end
```

becomes:

```
{p | P} <=> (a) Q when e and M(P,(a),Q)
```

Therefore:

```
MAPATTS =
  mapSet
     case
       (Element)[
         tag = 'Attribute',
         attributes = {
```

```
          (Att)[name ='name', value = n],
          (Att)[name = 'type', value = tid]}]
      ({(tid,t) | T})
      (Attribute)[name = n, type = t]
    end
    case
      (Element)[tag = t]
      (T)
      none when t <> 'Attribute'
    end
  end
end
```

Another application of a mapset occurs when translating the contents of a package:

```
MAPCONTENTS =
  mapSet
    case
      (Element,e)[tag = 'Package']
      (T)
      p when MAPPACKAGE(e,T,p)
    end
    case
      (Element,e)[tag = 'Class']
      (T)
      c when MAPCLASS(e,T,c)
    end
  end
```

12 Application: Removing Associations

QVT must be able to address the issue desugaring rich modelling languages into less expressive modelling languages. This section provides a simple example of such a desugaring where a package containing classes and binary associations is translated to a package containing just classes. The transformation removes the associations by encoding them as attributes in the respective classes and introducing constraints to ensure the attributes correctly implement the associations that they encode.

The following is a specification of the transformation:

```
REMASSOCSPEC =
  (Package,p) [ contents = C ] <=> (Package,p) [ contents = C' ]
    C' = C->select(c | c.isKindOf(Class)) and
    C->select(c | c.isKindOfAttribute()
```

```
    ->forAll(a |
       C'->exists(c |
         c = a.end1.type and
         c.attributes->exists(x |
           x.name = a.end2.name and
           x.type = a.end2.type) and
       C'->exists(c |
         c = a.end2.type and
         c.attributes->exists(x |
           x.name = a.end1.name and
           x.type = a.end1.type)))
```

The specification does not tell us how to implement this transformation. However we can refine the specification into an implementation by defining a mapping. The specification requires that the contents of a package be updated in place:

```
REMASSOC =
  (Package,p) [ contents = C ] => (Package,p) [ contents = C' ]
    where C' = REMASSOCS(C)
```

The set of elements supplied to `REMASSOCS` contains both classes and associations. The specification requires that the result of `REMASSOCS` contain exactly the classes (suitably modified). Therefore the mapping can proceed on a case-by-case basis and there are two cases to consider: a class and an association:

```
REMASSOCS = REMASSOCSCLASS + REMASSOCSASSOC
```

A class in the input is passed to the output:

```
REMASSOCSCLASS =
  { (Class,c)[] | C } => { (Class,c)[] | REMASSOCS(C) }
```

If an association is encountered in the input then we must modify the appropriate classes by adding attributes and constraints:

```
REMASSOCSASSOC =
  {(Association) [
     end1 = e1,
     end2 = e2 ],
   (Class,c1) [ attributes = A1 ],
   (Class,c2) [ attributes = A2 ]
   | C }
  when
    c1 = e1.type and
    c2 = e2.type
  => REMASSOCS({
      (Class,c1) [ attributes = { e2 | A1 } ],
      (Class,c2) [ attributes = { e1 | A2 } ]
      | C})
```

13 Refinement

The calculus supports a development process that allows relations to be incrementally refined to produce mappings. Since relations and mappings can be mixed, this allows individual components of a transformation specification to be refined to an implementation whilst leaving other elements as high-level specifications.

We would intend that the semantics of the QVT calculus would allow a library of pre-proven refinement transformation to be constructed. Since the calculus can be given an object representation, then the refinement rules can be expressed as calculus relations and mappings.

The following is a simple example of a refinement. Consider the selection of the oldest person in a group of people:

```
{(Person,p)[age=a] | P } <=> p when P->forAll(p' | p'.age <= p.age)
```

A generic refinement rule states that a relation with n patterns can be refined to a mapping with n - 1 patterns when the free variables in the relation constraint is a sub-set of the free variables in n - 1 pattern of the relation. This is the case here so:

```
{(Person,p)[age = a] | P } when P->forAll(p' | p'.age <= p.age) => p
```

Of course the resulting mapping is still non-deterministic. If we want to get an implementation we need to remove the non-determinism. This will involve iterating through the set to find the oldest person. A refinement rule allows us to introduce auxiliary mappings as follows:

```
{p | P} => M(p,P)
  where
    M = (Person,p)[age = 1], P when P->forAll(p' | p'.age < p.age) => p
```

A standard refinement rule allows universal quantification to be replaced by a recursive mapping:

```
{p | P} => M(p,P)
  where
    M = M1 + M2
       where
         M1 = p,{|} => p
         M2 = p, {p'|P} when p'.age < p.age and M(p,P)
```

Note that the non-deterministic selection from the initial set allows the is forced to select the correct element by the recursive case M2. A further standard pattern allows this non-local non-determinism to be removed. Where we rely on the correct element to be selected earlier on, we can select a truly arbitrary element and search for it later on changing our choice if necessary. Therefore we get:

```
{p | P} => M(p,P)
 where
   M = M1 + M2
      where
        M1 = p,{|} => p
        M2 = M3 + M4
          where
            M3 = p, {p' | P} when p'age > p.age => M(p',P)
            M4 = p, {p' | P} when p'age <= p'age => M(p,P)
```

Now the mapping does not depend on the person that is initial selected. It can be transformed into code:

```
{
  Iterator i = people.elements();
  Person p = (Person)i.next();
  while(i.hasNext()) {
    Person p' = (Person)i.next();
    if(p'.age > p.age)
      then p = p'
  }
  return p
}
```

14 Conclusion

This report has introduced the QVT calculus which is proposed as the basis for the Queries Views and Transformations RFP currently proposed by the OMG. We have proposed that technology for QVT is fundamentally based on selecting, constructing and updating model elements. The QVT calculus provides a precise basis for defining technologies for QVT that is declarative and compositional. We have shown that the calculus can represent model elements both at the MOF level (in terms of a meta-circular model) and at the UML level. A number of examples have been provided that show the calculus can represent typical transformations including model serialisation that is representative of transforming UML models to XMI. The practicality of the approach has been established by defining a QVT calculus machine that can execute transformations.

The calculus could be viewed as a higher order version of XSLT [11] although there are significant differences, particularly with respect to composition. There are many other proposals for transforming models, many of these are based on logic or graph theory: [3], [4], [5], [6], [7], [8], [12], [13]. We believe that the QVT calculus can address all issues raised by these methods in a single simple uniform language.

Acknowledgments: This work has been supported by Tata Consultancy Services Ltd. and King's College London.

References

[1] Object Management Group, Request for Proposal: MOF 2.0 Query / Views / Transformations RFP, ad/2002-04-10 (2002).

[2] D. DSouza, Model-driven architecture and integration - opportunities and challenges, http://www.kinetium.com/catalysis-org/publications/papers/2001-mda-reqs-desmond-6.pdf (2001).

[3] J. Bezivin, From object composition to model transformation with the MDA, in: TOOLS 2001, 2001.

[4] M. A. de Miguel, D. Exertier, S. Salicki, Speci

cation of model transformations based on meta templates, in: J. Bezivin, R. France (Eds.), Workshop in Software Model Engineering, 2002.

[5] K. Lano, J. Bicarregui, Semantics and transformations for UML models, in: J. Bezivin, P.-A. Muller (Eds.), The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, 1998, pp. 97 – 106.

[6] K. Lano, J. Bicarregui, UML refinement and abstraction transformations, in: Second Workshop on Rigorous Object Orientated Methods: ROOM 2, Bradford, May, 1998., 1998.

[7] W. M. Ho, J.-M. Jezequel, A. L. Guennec, F. Pennaneac'h, UMLAUT: An extendible UML transformation framework (1999).

[8] T. Levendovszky, G. Karsai, M. Maroti, A. Ledeczi, H. Charaf, Model reuse with metamodel-based transformations, in: C. Gacek (Ed.), ICSR, Vol. 2319 of Lecture Notes in Computer Science, Springer, 2002.

[9] QVT-Partners initial submission to qvt-rfp, http://www.qvtp.org/downloads/1.0/qvtpartners1.0.pdf (2003).

[10] Object Management Group, Meta Object Facility (MOF) Specification, formal/00-04-03 (2000).

[11] W3C, XSL Transformations (XSLT), http://www.w3.org/TR/xslt (1999).

[12] M. Gogolla, Graph transformations on the UML metamodel, in: J. D. P. Rolim, A. Z.

[13] Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, J. B. Wells (Eds.), ICALP Workshop on Graph Transformations and Visual Modeling Techniques, Carleton Scientific, Waterloo, Ontario, Canada, 2000, pp. 359.

A Semantics

For the purposes of defining a semantics we reduce the calculus to a simpler form:

$$
\begin{array}{lcl}
E & ::= & V \mid P(\Leftrightarrow P)* \textbf{ when } E \mid P* \textbf{ when } E \Rightarrow E \mid \\
 & & \{E* \mid E*\} \mid (E,E)[S*] \mid EE* \mid K \\
P & ::= & V \mid \{P* \mid P*\} \mid (P,P)[F*] \mid K \\
K & ::= & \text{constants} \\
S & ::= & V = E \\
F & ::= & V = P
\end{array}
$$

The semantics of the calculus is defined by a machine. Notational conventions:

- Sequences are: empty $[]$; cons head and tail $h : t$; append $s_1 + s_2$; or listed by element $[v_1, v_2, \ldots, x_n]$. The length of a sequence is $|s|$.
- Sets are the usual notation except $x \rightarrow X$ is the set X with element x added.
- An environment ρ is a sequence of bindings $n \mapsto v$ where n is a name and v is a value.
- Various instructions are used as in $\text{mkset}(i)$, the format is the name of the instruction followed by its operands.
- Tuples are used for various anonymous data structures, for example (ps, ρ, e) is a relation consisting of a sequence of patterns ps, a lexically closed environment ρ and the condition e.
- ! is used to denote the subsequent occurrence of a variable in a pattern. the first occurrence will just bind the variable, subsequent occurrences will check that the corresponding value has the same value each time.

The machine states are (s, ρ, c, f, d) where:

- s is the stack used to contain values. It is a sequence.
- ρ is the current lexical environment.
- c is the control consisting of expressions and machine instructions.
- f is the fail continuation - a machine state. If the machine ever needs to backtrack then it makes a transition to f.

- d is the dump - a machine state without f. This is used to save the current context when a mapping or relation is applied. When the application terminates, the machine returns to d.

The function span produces sequences of spanning sub-sets of a set given the length of the sequences, for example:

$$\begin{array}{l}\mathrm{span}(\{1,2,3\},2) = \\ \quad [[\emptyset,\{1,2,3\}], \\ \quad\ [\{1\},\{2,3\}], \\ \quad\ [\{2,3\},\{1\}],\ldots]\end{array}$$

Each state transition rule is defined below:

A constant evaluates to itself:

$$(s,\rho,k:c,f,d) \longrightarrow (k:s,\rho,c,f,d) \tag{1}$$

A variable in an expression evaluates to produce its value in the lexical environment:

$$(s,\rho,v:c,f,d) \longrightarrow ((\rho.v):s,\rho,c,f,d) \tag{2}$$

A relation definition evaluates to produce a relation:

$$(s,\rho,(ps,e):c,f,d) \longrightarrow ((ps,\rho,e):s,\rho,c,f,d) \tag{3}$$

A mapping definition evaluates to produce a mapping:

$$(s,\rho,(ps,e_1,e_2):c,f,d) \longrightarrow ((ps,\rho,e_1,e_2):s,\rho,c,f,d) \tag{4}$$

A set expression evaluates by evaluating each of the element and sub-set expressions and then constructing the set from the elements and sets on the stack. Note that the number of elements and the number of sub-sets are recorded as operands in the machine instruction:

$$(s,\rho,\{es \mid\ ds\}:c,f,d) \longrightarrow (s,\rho,es+ds+[\mathrm{mkset}(|es|,|ds|)]+c,f,d) \tag{5}$$

$$(v+w+s,\rho,\mathrm{mkset}(i,j):c,f,d) \longrightarrow ((\bigcup_{k\in 0\ldots i}\{v_k\} \cup \bigcup_{k\in 0\ldots j} w_k):s,\rho,c,f,d) \tag{6}$$

An object expression evaluates to produce an object. An object consists of a type expression, an identity expression and some field expressions. All are evaluated, leaving the values on the stack. the number of fields is recorded as an operand to the instruction:

$$(s,\rho,(e_1,e_2)[fs]:s,f,d) \longrightarrow (s,\rho,[e_1,e_2]+fs+[\mathrm{mkobj}(|fs|)]+c,f,d) \tag{7}$$

$$(fs+[v_2,v_1]+s,\rho,\mathrm{mkobj}(i):c,f,d) \longrightarrow ((v_1,v_2)[fs]:s,\rho,c,f,d) \tag{8}$$

$$(s,\rho,(v=e):c,f,d) \longrightarrow (s,\rho,e:\mathrm{mkfield}(v):c,f,d) \tag{9}$$

$$(w:s,\rho,\mathrm{mkfield}(v):c,f,d) \longrightarrow ((v=w):s,\rho,c,f,d) \tag{10}$$

An application expression could be a relation application or a mapping application. We just evaluate all the components and leave an apply instruction with an operand telling us how many arguments there are expected on the stack:

$$(s, \rho, e(es) : c, f, d) \longrightarrow (s, \rho, es + [e, \text{apply}(|es|)] + c, f, d) \tag{11}$$

If we apply a relation thenwe need to match against the patterns and then check the constraint. Pattern matching simply adds the patterns to the control and the corresponding values to the stack. Checking a constraint involves evaluating it and then continuing if it produces `true` or backtracking if it produces `false`:

$$((ps, \rho_1, e) : (w + s), \rho_2, \text{apply}(i) : c, f, d) \longrightarrow (w, \rho_1, ps + [\text{check}(e)], f, (s, e, c, d)) \tag{12}$$

If we apply a mapping then we need to match the patterns, check the constraint and then evaluate the body.

$$((ps, \rho_1, e_1, e_2) : (w{+}s), \rho_2, \text{apply}(i) : c, f, d) \longrightarrow (w, \rho_1, ps{+}[\text{check}(e_1), e_2], f, (s, e, c, d)) \tag{13}$$

Check just evaluates the expression:

$$(s, \rho, \text{check}(e) : c, f, d) \longrightarrow (s, \rho, e : \text{check} : c, f, d) \tag{14}$$

Check will backtrack if the constraint failed:

$$(v : s, \rho, \text{check} : c, f, d) \longrightarrow \begin{cases} (s, \rho, c, f, d) & \text{when } v = \text{true} \\ f & \text{otherwise} \end{cases} \tag{15}$$

Initial occurrences of variables just bind to values:

$$(x : s, \rho, v : c, f, d) \longrightarrow (s, \rho[v \mapsto x], c, f, d) \tag{16}$$

Subsequent occurrences of variables check that the value is consistent with the value of the variable in the current environment:

$$(x : s, \rho, !v : c, f, d) \longrightarrow \begin{cases} (s, e, c, f, d) & \text{when } \rho.v = x \\ f & \text{otherwise} \end{cases} \tag{17}$$

Selection from sets during pattern matching involves trying all alternatives and keeping track of which elements have been tried when we backtrack. If we run out of choices then we must backtrack:

$$(Q : s, \rho, \{ps \mid Qs\} : c, f, d) \longrightarrow ((Q, \emptyset) : s, \rho, \text{select}(ps) : \{|Qs\} : c, f, d) \tag{18}$$

If we have run out of element patterns then we succeed and leave the rest of the set on the stack:

$$((Q, R) : s, \rho, \text{select}(\emptyset) : c, f, d) \longrightarrow (Q \cup R : s, \rho, c, f, d) \tag{19}$$

If we have run out of possibilities then fail:

$$((\emptyset, \_) : s, \rho, \text{select}(p \to P) : c, f, d) \longrightarrow f \tag{20}$$

Select an element but remember which elements have been tried in case we backtrack:

$$((v \to V, Q) : s, \rho, \text{select}(p \to P) : c, f, d) \longrightarrow$$

$$(v : (\emptyset, V \cup Q) : s, \rho, p : \text{select}(P) : c, ((V, v \to Q) : s, e, \text{select}(p \to P) : c, f, d))) \tag{21}$$

Sub-set patterns involved constructing all the possible spanning subsets and trying them all:

$$(Q : s, \rho, \{|Qs\} : c, f, d) \longrightarrow (\text{span}(Q, |Qs|) : s, \rho, \text{split}(Qs) : c, f, d) \tag{22}$$

$$([] : s, \rho, \text{split}(Qs) : c, f, d) \longrightarrow f \tag{23}$$

$$((Ps : Pss) : s, \rho, \text{split}(Qs) : s, f, d) \longrightarrow (Ps{+}s, \rho, Qs{+}c, (Pss : s, \rho, \text{split}(Qs) : s, f, d), d) \tag{24}$$

An object pattern must match the type pattern, the identity pattern and all the field pattterns:

$$((v_1, v_2)[S] : s, \rho, (p_1, p_2)[F] : c, f, d) \longrightarrow (v_1 : v_2 : S : s, \rho, p_1 : p_2 : F : c, f, d) \tag{25}$$

If we run out of field patterns then we have matched the object:

$$(S : s, \rho, \emptyset : c, f, d) \longrightarrow (s, \rho, c, f, d) \tag{26}$$

Select the matching field:

$$((S_1 \cup \{v = x\} \cup S_2) : s, \rho, (F_1 \cup \{v = p\} \cup F_2) : c, f, d) \longrightarrow (x : (S_1 \cup S_2) : s, \rho, p : (F_1 \cup F_2) : c, f, d) \tag{27}$$

If we successfully complete a relation or mapping then return to the caller:

$$(v : \_, \_, [], f, (s, \rho, c, d)) \longmapsto (v : s, \rho, c, f, d)) \tag{28}$$