

An Overview of QVT

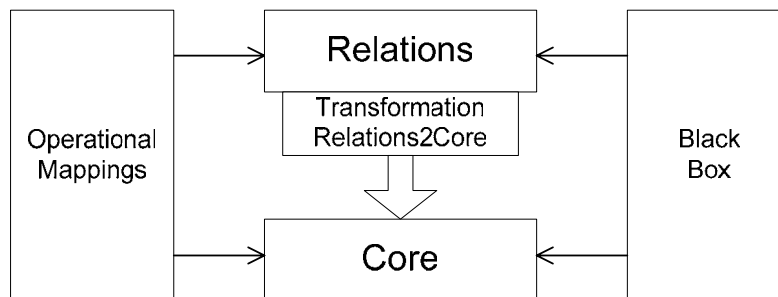
László Lengyel

<http://www.vmts.aut.bme.hu>

Introduction /1

- In the model-driven architecture, QVT (Queries/Views/Transformations) is a standard for model transformation defined by the Object Management Group
- Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification
- The QVT specification has a hybrid declarative/imperative nature, with the declarative part being split into a two-level architecture:
 - The user-friendly *Relations* metamodel and language which supports complex object pattern matching and object template creation
 - A *Core* metamodel and language is defined using minimal extensions to EMOF and OCL
- Imperative part: Operational Mapping.

QVT Architecture



Introduction – Relations Language

- A declarative specification of the relationships between MOF models.
- The Relations language supports
 - Complex object pattern matching, and
 - Implicitly creates trace classes and their instances to record what occurred during a transformation execution.
- Relations can assert that other relations also hold between particular model elements matched by their patterns.
- Has a graphical concrete syntax

Introduction – Core Language

- A small model/language which only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models.
- Treats all of the model elements of source, target and trace models symmetrically.
- Equally powerful to the Relations language, and because of its relative simplicity, its semantics can be defined more simply, although transformation descriptions described using the Core are therefore more verbose.
- In addition, the trace models must be explicitly defined, and are not deduced from the transformation description, as is the case with Relations.

Introduction – Imperative Implementations

- In addition to the declarative Relations and Core Languages, there are two mechanisms for invoking imperative implementations of transformations from Relations or Core:
 - A standard language, Operational Mappings, and
 - A non-standard Black-box MOF Operation implementations.

Introduction – Operational Mappings Language

- The QVT/OperationalMapping language is an imperative language that extends both QVT/Relations and QVT/Core.
- This language is specified as a standard way of providing imperative implementations, which populate the same trace models as the Relations Language
- Provides OCL extensions with side effects that allow a more procedural style, and a concrete syntax that looks familiar to imperative programmers.
- The syntax of the QVT/OperationalMappings language provides constructs commonly found in imperative languages (loops, conditions, etc.).
- Mappings Operations can be used to implement one or more Relations from a Relations specification when it is difficult to provide a purely declarative specification of how a Relation is to be populated.
- Mappings Operations invoking other Mappings Operations always involves a Relation for the purposes of creating a trace between model elements, but this can be implicit, and an entire transformation can be written in this language in the imperative style.
- Operational transformation - A transformation entirely written using Mapping Operations.

Introduction – Black Box Implementations

- Allows complex algorithms to be coded in any programming language
- Allows the use of domain specific libraries to calculate model property values. For example, mathematical, engineering, bio-science and many other domains have large libraries that encode domain-specific algorithms which will difficult, if not impossible to express using OCL.
- However, it is also dangerous. The plugin implementation has access to object references in models, and may do arbitrary things to those objects.

Execution Scenarios

- The semantics of the Core language and the Relations language allow the following execution scenarios:
 - Check-only transformations to verify that models are related in a specified way,
 - Single direction transformations,
 - Bi-directional transformations,
 - The ability to establish relationships between pre-existing models,
 - Incremental updates when a related model is changed after an initial execution, and finally,
 - The ability to create or delete objects and values, while also being able to specify which objects and values must not be modified.

The Relations Language

Transformations and Model Types

- A transformation between candidate models is specified as a set of relations that must hold for the transformation to be successful.
- A candidate model is any model that conforms to a model type.
- Candidate models are named.
- An example is:

```
transformation umlRdbms (uml : SimpleUML,  
  rdbms : SimpleRDBMS) {
```
- A transformation can be invoked either to check two models for consistency or to modify one model to enforce consistency.

Relations and Domains

- Relations in a transformation declare constraints that must be satisfied by the elements of the candidate models. A relation, defined by two or more domains and a pair of when and where predicates.
- A domain is a distinguished typed variable that can be matched in a model of a given model type.
- A domain has a pattern, which can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain's type.
- A pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must satisfy to qualify as a valid binding of the pattern.

```
relation PackageToSchema /* map each package to a schema */  
{  
  domain uml p:Package {name=pn}  
  domain rdbms s:Schema {name=pn}  
}
```

When and Where Clauses

- A relation can be constrained by two sets of predicates, a *when* clause and a *where* clause.
- The when clause specifies the conditions under which the relationship needs to hold.
- The where clause specifies the condition that must be satisfied by all model elements participating in the relation, and it may constrain any of the variables in the relation and its domains.
- The when and where clauses may contain any arbitrary OCL expressions in addition to the relation invocation expressions.

```
relation ClassToTable
{
  domain uml c:Class {
    namespace = p:Package {},
    kind='Persistent',
    name=cn
  }
  domain rdbms t:Table {
    schema = s:Schema {},
    name=cn,
    column = cl:Column {
      name=cn+'_tid',
      type='NUMBER'},
    primaryKey = k:PrimaryKey {
      name=cn+'_pk',
      column=cl}
  }
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}
```

Top-level Relations

- A transformation contains two kinds of relations top-level and non-top-level.
- The execution of a transformation requires that all its top-level relations hold.
- Non-top-level relations are required to hold only when they are invoked directly or transitively from the where clause of another relation.

```
transformation umlRdbms (uml : SimpleUML, rdbms :
SimpleRDBMS) {
  top relation PackageToSchema {...}
  top relation ClassToTable {...}
  relation AttributeToColumn {...}
}
```

Check and Enforce

- When a transformation executes in the direction of a checkonly domain, it is simply checked to see if there exists a valid match in the relevant model that satisfies the relationship.
- When a transformation executes in the direction of the model of an enforced domain, if checking fails, the target model is modified so as to satisfy the relationship, i.e. a check-before-enforce semantics.

```
relation PackageToSchema
{
    checkonly domain uml p:Package {name=pn}
    enforce domain rdbms s:Schema {name=pn}
}
```

Pattern Matching

- A template expression match results in a binding of model elements from the candidate model to variables declared by the domain.
- A template expression match may be performed in a context where some of the domain variables may already have bindings to model elements (e.g. resulting from an evaluation of the when clause or other template expressions). In this case template expression match finds bindings only for the free variables of the domain.

```
c:Class { namespace = p:Package {},
    kind='Persistent',
    name=cn
}
```

Keys and Object Creation using Patterns

- An object template expression also serves as a template for creating an object in a target model: When for a given valid match of the source domain pattern, there does not exist a valid match of the target domain pattern, then the object template expressions of the target domain are used as templates to create objects in the target model.

```
t:Table {  
  schema = s:Schema {},  
  name = cn,  
  column = cl:Column {name=cn+'_tid', type='NUMBER'},  
  primaryKey = k:PrimaryKey {name=cn+'_pk', column=cl}  
}
```

- Creating objects we want to ensure that duplicate objects are not created when the required objects already exist.

```
key Table {schema, name};
```

Change Propagation

- In relations, the effect of propagating a change from a source model to a target model is semantically equivalent to executing the entire transformation afresh in the direction of the target model.
- The semantics of object creation and deletion guarantee that only the required parts of the target model are affected by the change:
 - The semantics of check-before-enforce ensures that target model elements that satisfy the relations are not touched.
 - Key-based object selection ensures that existing objects are updated where applicable. Deletion semantics ensures that an object is deleted only when no other rule requires it to exist.

In-place Transformations

- A transformation may be considered in-place when its source and target candidate models are both bound to the same model at runtime. The following additional comments apply to the enforcement semantics of an in-place transformation:
 - A relation is re-evaluated after each enforcement-induced modification to a target pattern instance of the model.
 - A relation's evaluation stops when all the pattern instances satisfy the relationship.

Integrating Black-box Operations with Relations

- A relation may optionally have an associated black-box operational implementation to enforce a domain.
- The black-box operation is invoked when the relation is executed in the direction of the enforced domain and the relation evaluates to false as per the checking semantics.
- The invoked operation is responsible for making the necessary changes to the model in order to satisfy the specified relationship. It is a runtime exception if the relation evaluates to false after the operation returns.
- The Relations which may be implemented by Mapping Operations and Black box Operations are restricted in the following ways:
 - Their domain should be primitive or contain a simple object template (with no sub-elements)
 - The when and where clause should not define variables.
- These restrictions allow for a simple call-out semantics, which does not need any constraint evaluation before, and constraint checking after the operation invocation.

Checking Semantics

- For each valid binding of variables of the when clause and variables of domains other than the target domain k , that satisfy the when condition and source domain patterns and conditions, there must exist a valid binding of the remaining unbound variables of domain k that satisfies domain k 's pattern and where condition.

Enforcement Semantics

- For each valid binding of variables of the when clause and variables of domains other than the target domain k , that satisfy the when condition and source domain patterns and conditions, if there does not exist a valid binding of the remaining unbound variables of domain k that satisfies domain k 's pattern and where condition, then create objects (or select and modify if they already exist) and assign properties as specified in domain k pattern.
- Also, for each valid binding of variables of domain k pattern that satisfies domain k condition, if there does not exist a valid binding of variables of the when clause and source domains that satisfies the when condition, source domain patterns and where condition, and at least one of the source domains is marked checkonly (or enforce, which entails check), then delete the objects bound to the variables of domain k when the following condition is satisfied: delete an object only if it is not required to exist by any other valid binding of the source domains as per the enforcement semantics.

The Relations Metamodel

- The Relations metamodel is structured into three packages:
 - QVTBase - This package contains a set of basic concepts, many reused from the EMOF and OCL specifications that structure transformations, their rules, and their input and output models.
 - QVTTemplate, and
 - QVTRelation.

Operational Mappings

QVT Operational Mapping

- The QVT Operational Mapping language allows either to define transformations using a complete imperative approach (operational transformations) or allows complementing relational transformations with imperative operations implementing the relations (hybrid approach).

Operational Transformations

- An operational transformation represents the definition of a unidirectional transformation that is expressed imperatively. It defines a signature indicating the models involved in the transformation and it defines an entry operation for its execution (named *main*). Like a class, an operational transformation is an instantiable entity with properties and operations.

```
transformation Uml2Rdbms(in uml:UML,out rdbms:RDBMS) {  
  main() {  
    uml.objectsOfType(Package)->map packageToSchema();  
  }  
  ...  
}
```

Operational Mapping Overview

- Model types - A model type is defined by a metamodel (a set of MOF packages), a conformance kind (strict or effective) and an optional set of conditions. Model types introduce accurate flexibility for writing transformation definitions that are applicable to similar metamodels.
- Libraries - A library contains definitions that can be reused by transformations.
- Mapping operations - A mapping operation is an operation that implements a mapping between one or more source model elements into one or more target model elements.
- A mapping operation is syntactically described by a signature, a guard (a when clause), a mapping body and a postcondition (a where clause).

Object creation and population in mapping operations

- The QVT operational mappings language defines a specific "high-level" facility to create and/or update model elements.

```
object s:Schema {
    name := self.name;
    table := self.ownedElement->map class2table();
}

mapping Package::packageToSchema() : result:Schema
when { self.name.startingWith() <> "_" }
{
    population {
        object result:Schema {
            name := self.name;
            table := self.ownedElement->map class2table();
        }
    }
}
```

Further Features /1

- Constructor operations
- Helper is an operation (helper, query): A helper is an operation that performs a computation on one or more source objects and provides a result. A helper may have side-effects on the parameters.
- Intermediate data
- Updating objects and resolving object references: A common technique in model transformation is the usage of various passes to solve cross referencing between model elements. The language provides resolution constructs to access target objects created previously from source objects. These facilities implicitly use the trace records created by the execution of a mapping operation.

Further Features /2

- Composing transformations
- Reuse facilities for mapping operations (inherit, merge)
- Disjunction of mapping operations: A mapping operation may be defined as a disjunction of an ordered list of mappings. This means that an invocation of the operation results on the selection of the first mapping whose guard (type and when clause) succeeds.
- Type extensions: The language extends the OCL and MOF type system with some general purpose data types. These are mutable lists, dictionary types and anonymous tuples.

Further Features /3

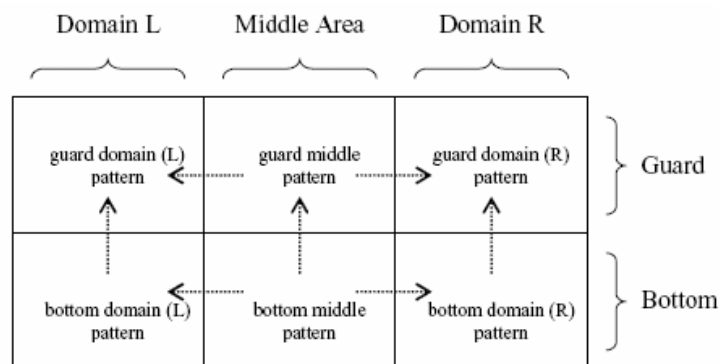
- Imperative expressions: The QVT operational mapping language is an imperative language to define transformations. It extends OCL but includes all the necessary machinery that is needed to write in a comfortable way complex transformations. The imperative expressions in QVT realize a compromise between some nice functional features found in OCL and the more traditional constructs that we found in general purpose languages like Java. (e.g. *compute*, *forEach*, *while*, *continue*, *break*)
- Pre-defined variables: *this*, *self* and *result*
- *Null*
- Advanced features: dynamic definition and parallelism

The Core Language

Comparison with the Relational Language

- Supports pattern matching
- The Core Language is as powerful as the Relations language, though simpler.
- The semantics of the Core Language can be defined more simply, though transformations described using the Core are more verbose.
- Relations Language implicitly creates trace classes and objects to record what occurred during a transformation execution. In the Core Language, these traces are explicit.
- The following aspects, implied in the relational language, have to be defined explicitly when using the core language:
 - The patterns that match against, and create the instances of the classes that are the trace objects of the transformation
 - Atomic sets of model elements that are created or deleted as a whole, specified in smaller patterns than commonly specified in the Relational Language.

Core Domains and Pattern Dependencies



Mappings

- A transformation contains mappings. A mapping has zero or more domains. A domain has an associated model type of the transformation. A domain does not have a name; it is uniquely identified by the mapping and the associated model type. Thus, in a transformation execution, each domain specifies a set of model elements of exactly one of the candidate models that is of interest to a mapping.
- A mapping defines a one-to-one relation between all bottom patterns of that mapping. This means that for any successful match of one of the bottom patterns there should exist exactly one successful match for each other bottom pattern. The one-to-one constraint between the bottom patterns is only checked or enforced if a successful match for each guard pattern of that mapping exists.

Patterns

- A pattern is specified as a set of variables, predicates and assignments. Patterns can be matched and enforced.
- Patterns can depend on each other. A pattern that depends on another pattern may use the variables of that other pattern in its own predicates and assignments, and is matched using value bindings of the variables produced by a match of that other pattern.
- In a mapping, bottom patterns depend on guard patterns (in the same column) and middle patterns depend on domain patterns (in the same row).

Guards and Bottom Patterns

- Guards of a mapping narrow the selection of model elements to be considered for the mapping. The matching of bottom patterns takes place in the context of a valid combination of valid bindings of all the guard patterns.
- The bottom patterns of a mapping are the patterns that are checked and possibly enforced. A mapping declares essentially that all bottom patterns should relate one-to-one.
- Bottom patterns can have (in addition to variables and predicates) realized variables, assignments and black-box operations. These extra features can have side effects when executed in enforcement mode, and when a one-to-one constraint is violated.

Relations to Core Transformation

Trace classes

- Trace classes and their instances are important for supporting efficient implementation of incremental execution scenarios.
- In relations, trace classes are not explicitly specified and used: a relation directly specifies the relationship that should hold between source and target domains.
- In core, trace classes and patterns over them are an essential part of the specification of mappings.

When and Where Clauses

- Relation's when and where clauses map to the middle area of a mapping.
- The when clause maps to the middle-guard, and the where clause maps to the middle-bottom.
- A domain pattern maps to a domain area in the core domain variables that occur in the when clause map to the domain-guard and the remaining domain pattern maps to the domain-bottom.

Relation Invocation Dependencies

- Relations can have arbitrary invocation dependencies. A relation can invoke multiple relations and a relation can be invoked by multiple relations. A relation can invoke another relation in its pre-condition (when clause) or post-condition (where clause). This style is intuitive to users and allows for complex composition hierarchies to be built.
- When mapping from relations to core we need to decompose relation invocation dependencies into simpler mapping dependencies.
- Since relations are expressed in terms of patterns, relation dependencies can be translated to corresponding pattern dependencies.
- A relation domain can have a complex pattern consisting of multiple object nodes, properties and links. While translating to core an enforced relation domain's complex pattern needs to be split into simpler patterns of multiple nested composed mappings.

Criticisms

- No complete implementation yet
- Fuzzy requirements - The requirements for QVT are mainly the translation of PIM to PSM. Since the specification of what is really a PIM and a PSM was never achieved in the OMG MDA guide, the QVT requirements remains quite fuzzy.
- Limited usability - QVT is for XMI to XMI transformations only. Since the quantity of real data natively encoded in XMI is very limited (and rarely found), the potential applicability of QVT in industry is also and consequently rather limited. In order to be practically used, QVT tools need to be complemented by Model to Text or Text to Model tools.
- Huge Specification
- Premature standardization
- Committee-defined



Questions

- Will QVT be a wide spread transformation language?
- Will graph transformation results reused on QVT level?
(QVT To Graph Transformation mappings)



Thank you for your attention!