Initial submission for MOF 2.0 Query / Views / Transformations RFP

QVT *Partners*

http://qvtp.org/

Version 1.0 (2003/03/03)

Submitted by:

Tata Consultancy Services

Supported by:

Artisan Software
Kinetium
King's College London
University of York

Contents

Preface

This version of the submission

This is an initial submission to the QVT RFP. Although many parts of the document are in a very polished state, other parts remain to be filled in.

This version of the submission was generated at 21:44 GMT on March 3, 2003 by Laurence Tratt.

Submission contact point

Sreedhar Reddy `sreedharr@pune.tcs.co.in`

The QVT Partners host a web site at `http://qvtp.org/` where you can find out further information about the submission. You can also download the latest version of this submission, and join the submission's public announcment `qvt-announce@qvtp.org` and discussion `qvt-discuss@qvtp.org` mailing lists.

Guide to the material in the submission

This document is structured in two main parts:

I. This section contains an overview of our response to the RFP (section 1) and details our resolution of RFP requirements and requests (section 2). This later section also includes a summary of our submissions compliance with existing OMG standards such as CWM (section 2.4).

II. This section contains the technical details of our specification.

Statement of proof of concept

Prototype implementations of the submission are currently being developed in the submitters meta-modeling tools. Examples, including some of those presented in this document, already run successfully through the tools, confirming the power of our proposed solution.

Resolution of RFP requirements and requests

See section 2 on page 21.

Submitters

This document is submitted by the following OMG members:

Tata Consultancy Services	http://www.tcs.com/

Supporters

This document is supported by the following OMG members:

Artisan Software	http://www.artisansw.com/
Kinetium	http://www.kinetium.com/
King's College London	http://www.kcl.ac.uk/
University of York	http://www.york.ac.uk/

Submission team

The following people have been chiefly responsible for the work involved in this version of the submission:

King's College London	Biju Appukkutan	biju@dcs.kcl.ac.uk
	Tony Clark	anclark@dcs.kcl.ac.uk
	Laurence Tratt	laurie@tratt.net
Tata Consultancy Services	Sreedhar Reddy	sreedharr@pune.tcs.co.in
	R. Venkatesh	rvenky@pune.tcs.co.in
University of York	Andy Evans	andye@cs.york.ac.uk
	Girish Maskeri	girishmr@cs.york.ac.uk
	Paul Sammut	pauls@cs.york.ac.uk
	James Willans	jwillans@cs.york.ac.uk

Part I.

Response to the RFP

1. Introduction

This document is an initial submission to the *Queries, Views and Transformations* (QVT) Request For Proposal (RFP) document [OMG02] issued by the Object Management Group (OMG) in April 2002.

In this chapter we aim to give an overview of our interpretation of the RFP, and also an overview of our proposal. As this chapter is, by design, short and to the point it should not be viewed as authoritative; part II of this document contains our definitive specification.

1.1. An overview of the RFP

Queries, views and transformations are subjects which will be vital to the success of the OMG's Model Driven Architecture Initiative (MDA – see section 1.2). The ability to manipulate models involved in the MDA process is crucial to the success of MDA and it is this ability which the QVT RFP aims to find a solution for.

We now present high level definitions of the RFP's subjects:

Queries take as input a model, and select specific elements from that model.

Views are models that are derived from other models.

Transformations are model specifications or model mappings that take as input a model and relate it to a model, or create a new model respectively.

It is important to note that queries, views and transformations can be split into two distinct groups. Queries and transformations take models and perform actions upon them, resulting in a new or changed model. In contrast to that, views themselves are models. Queries and transformations may possibly create views, but views themselves are passive.

1.2. MDA

[OMG02] defines the MDA vision thus:

> MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform, and provides a set of guidelines for structuring specifications expressed as models.
>
> The MDA approach and the standards that support it allow the same model specifying system functionality to be realized on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms, and allows different applications to be integrated by explicitly relating their models, enabling integration and interoperability and supporting system evolution as platform technologies come and go.

In less technical terms, MDA aims to allow developers to create systems entirely with models[1]. Furthermore, MDA envisages systems being comprised of many small, manageable models rather than one gigantic monolithic model. Finally, MDA allows systems to be designed independently of the eventual technologies they will be deployed on; a Platform Independent Model (PIM) can then be transformed into a Platform Specific Model (PSM) in order to run on a specific platform.

In [DSo01] D'Souza presents another perspective of MDA and introduces the concepts of horizontal and vertical dimensions. A vertical dimension represents changing levels of abstraction in a particular part system. A horizontal dimension represents different parts of a system e.g. different departments within a company. A complete model of a system will be comprised of all the relevant horizontal dimensions integrated together, and all at a specific level of abstraction.

No matter what perspective on MDA one has, two common threads run through them all: model federation and platform independence. Figure 1.1 (based partly on a D'Souza example) shows an overview of this idea. It shows a company horizontally split into multiple departments, each of which has a model of its system. These models can be considered to be views on an overall system PIM. The PIM can be converted into a PSM. In order to realize this vision, there has to be some way to specify the changes that models such as that in figure 1.1 undergo. The enabling technology is *transformations*. In figure 1.1 a transformation T_1 integrates the company's horizontal definitions into an overall PIM, and a transformation T_2 converts the overall PIM into a PSM.

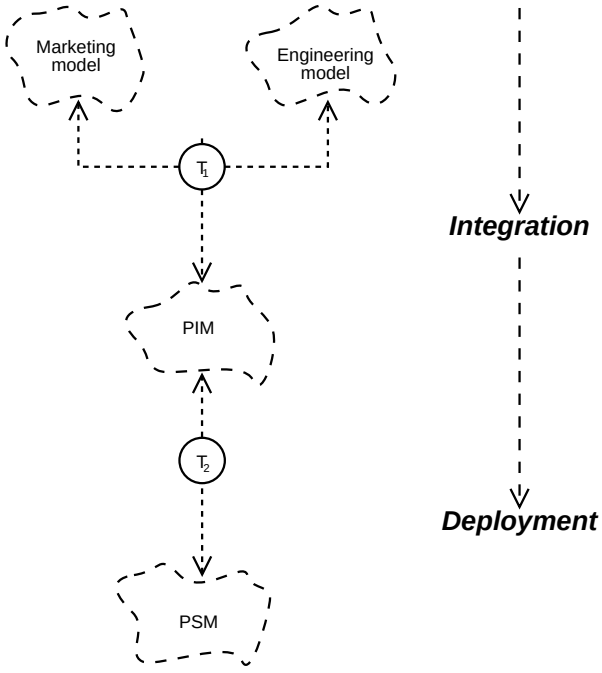

Figure 1.1.: Transformations and MDA

The concepts of abstraction and refinement are also vital to MDA. Not only do these play a part in models created within the MDA framework, but also to the MDA framework itself. Figure 1.2 shows the PIM to PSM

[1]This does not mean that *everything* must be specified fully or even semi-graphically – the definition of model allows one to drill down right to source code level.

transformation from figure 1.1 expanded to reflect the fact that often complex transformations are not done in a single step. Rather, models often go through several intermediate stages before reaching a final transformation.

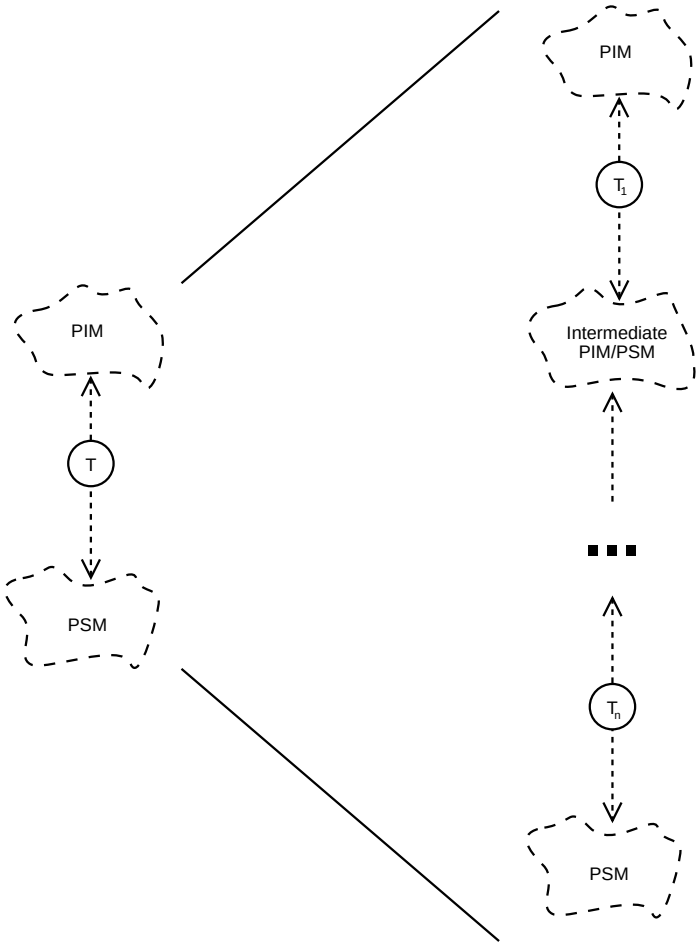

Figure 1.2.: An expanded PIM to PSM transformation

Transformations are undoubtedly the key technology in the realization of the MDA vision. They are present explicitly – as in the transformation of a PIM to a PSM – and implicitly – the integration of different system views – throughout MDA.

1.2.1. Uses

The following are some representative MDA related uses where transformations are, or could be, involved:

- Integrating the components of a horizontal direction in a federated model. This is the example used in figure 1.1.
- Converting a model ‘left to right’ or ‘right to left’. This is a very common operation in tools, for example saving a UML model to XML and reading it back in again.
- Converting a PIM into a PSM. The PSM’s meta-model might be one of J2EE or .net. As shown in figure 1.2, there is no reason why the conversion has to necessarily happen in one stage.

- Reverse engineering. For example, a tool which recovers Java source code from class files.
- Technology migration. This is similar to reverse engineering, but whereas reverse engineering is simply trying to recover lost information, technology migration is effectively trying to convert outdated systems into current systems. For example, a tool which migrates legacy COBOL code to Java.

1.3. A general scenario

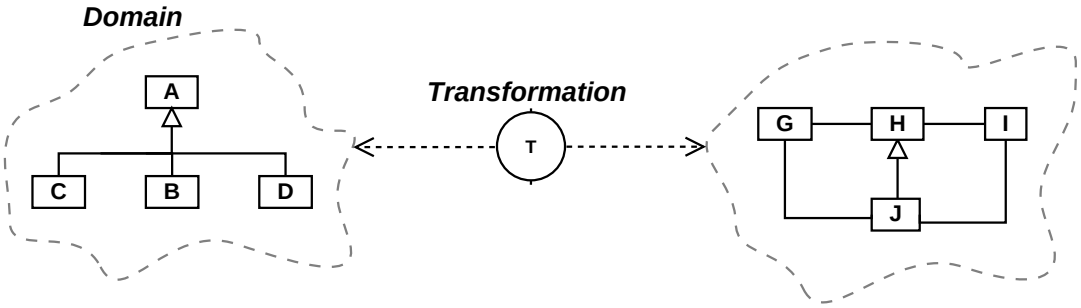

Figure 1.3.: A high level transformation

Figure 1.3 shows a high-level view of what a user might expect a transformation to look like. Please note that this figure does *not* reflect the concrete syntax we use in the rest of this document.

In figure 1.3, a transformation T involves two model domains. We have intentionally drawn the model domains with a vague outline, as different users may have different expectations of what precisely a domain should be – for example, single model elements, sets of model elements, and packages are supported by our proposal. Regardless of the precise definition of domain chosen, the important fact at this high level is that transformations involve domains. At this stage we make no assumptions about the number of domains involved in a transformation, nor about any notion of directionality or executability.

We build upon this high-level scenario throughout the rest of this chapter.

1.4. Our proposal

In the subsequent sections of this chapter, we present an overview of our proposals to the three main parts of the RFP individually. Full technical details are presented in part II of this document.

Our proposal is particularly concerned with providing a comprehensive solution to transformations. This is chiefly because we feel that a practicable, technically sound, definition of transformations will be the main enabling factor in achieving MDA, and that such a definition of transformations presents a considerably greater challenge than similar definitions of queries and views.

1.5. Queries

We propose that a possibly extended version of OCL 2.0 is used as the query language. OCL 2.0 resolves OCL 1.3's deficiencies as a query language [MC99]. Using OCL has several benefits: the user community is

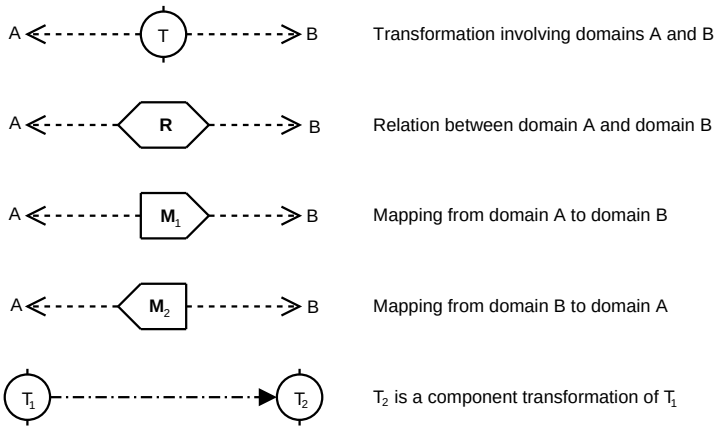

Figure 1.4.: Concrete syntax

intimately familiar with it; no effort need be expended on the definition of 'yet another new language'; and there is already substantial tool support for OCL.

1.6. Views

We propose that a view is a projection on a parent model, created by a transformation. From this simple definition, we can build the necessary machinery to cope with advanced technologies such as RM-ODP style viewpoints [BMR95]. Viewpoints are an interesting and useful abstraction technique. Essentially, they can be viewed as being analogous to a query which not only creates a view but also potentially restricts the meta-model of the view as well. Thus from each viewpoint one does not in general have enough information to rebuild the entire system. One possible mechanism for dealing with viewpoints in our proposal is to use a query to create a view of a model, and then use a transformation to alter the view to reflect the viewpoints restricted meta-model.

For the purposes of this initial submission we note the importance of addressing topics such as model integration and consistency, but do not put forward any concrete proposals for them.

1.7. Transformations

Our submission presents a detailed proposal for transformations. In order to aid quick comprehension, in this overview we present the most relevant points in separate subsections. As this section contains several transformation diagrams, figure 1.4 shows the most important parts of concrete syntax we use in relation to transformations. Note that although figure 1.4 shows transformations, relations and mappings between two domains, our definitions allow each of these to be, in general, between an arbitrary number of domains.

1.7.1. A layered approach to the definition of transformations

Our definition of transformations comes in two distinct layers. Reusing terminology familiar from the UML2 process, we name these layers *infrastructure* and *superstructure*.

We define a simple infrastructure which has a small extension to the MOF meta-model and whose semantics are easily defined in terms of existing OMG standards. The infrastructure contains what we consider to be

a sensible minimum of machinery necessary to support all types of transformations. The infrastructure is necessarily low-level and not of particular importance to end users of transformations. Its use is a simple semantic core; its presence is also useful for tool vendors.

Secondly we present a superstructure which contains a much higher-level set of transformation types suitable for end users. Some parts of the infrastructure are effectively included ‘as is’ in the superstructure. Concepts which exist in the superstructure but not in the infrastructure have a translation into the infrastructure. In this submission we present the definition of a standard superstructure. This superstructure contains plug points to allow it to be easily extended with new features. However, the very nature of our infrastructure/superstructure split also means that it is possible to create completely new superstructures, provided that they have a translation down into the infrastructure.

By separating out the concepts of infrastructure and superstructure we gain a significant advantage: whilst the infrastructure remains unchanged, different types of transformation can be added to the superstructure to support different user domains. Tools which support the infrastructure definition will be able to also support extensions or alterations of the superstructure. Note that we specifically do not preclude the possibility of tools having native support for superstructure, or variants on the superstructure.

Figure 1.5 shows an overview of how a superstructure model is translated into an infrastructure model. The general idea is that rich models in the superstructure are translated into much simpler models in the infrastructure; the information that is lost in the transition from rich to simple models is ‘recovered’ by adding extra information `Q` into the infrastructure translation, as OCL constraints or ASL as appropriate. Information that is encoded in `Q` includes such things as typing and structural information.

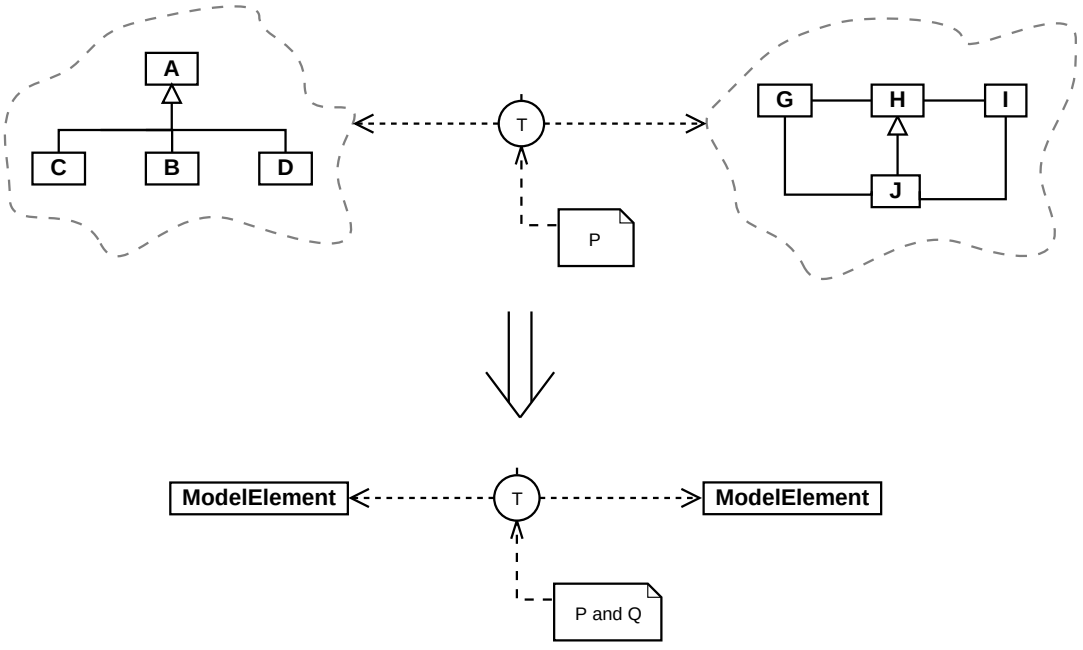

Figure 1.5.: Translating superstructure (top) to infrastructure (bottom)

Most of the rest of this section is relevant to both infrastructure and superstructure; when this is not the case, we explicitly note the fact.

1.7.2. Relations, mappings and implementations

We have devised an overall framework for transformations that allows one to use a variety of different transformation styles; furthermore, our framework also transparently allows transformations to change style throughout the lifetime of a system. Such transparency is enabled by our identification of two distinct sub-types of transformations: *relations* and *mappings*.

Relations are multi-directional transformation specifications i.e. they are declarative. In the general case they are non-executable, but we have identified useful restricted types of bi-directional relations which can be automatically refined into mappings. Relations are written in any valid UML constraint language, OCL being an obvious example.

Typically relations are used in the specification stages of system development.

Mappings are transformation implementations i.e. they are operational. Unlike relations, mappings are potentially uni-directional. Mappings are expressed in the Actions Semantics Language (ASL) and thus encompass all programming language implementations.

Mappings can *refine* any number relations, in which case the mapping must be consistent with the relations it refines.

Figure 1.6 shows a relation `R` relating two domains. There is also a mapping `M` which refines relation `R`; since `M` is directed, it transforms model elements from the right hand domain into the left hand domain.

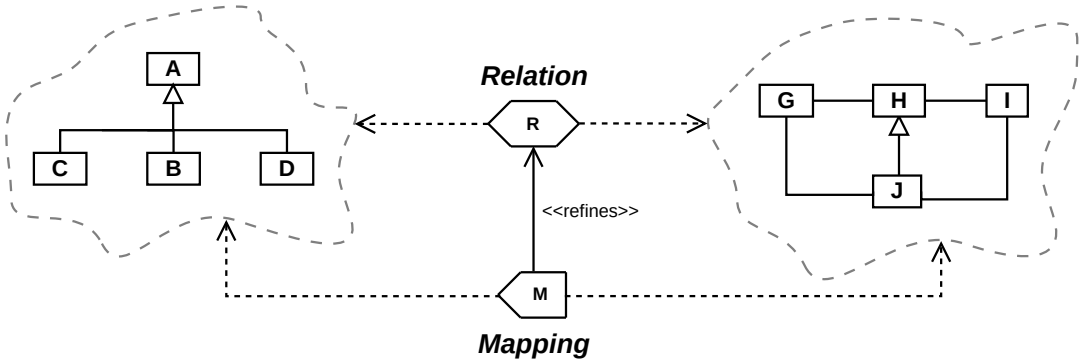

Figure 1.6.: A high level relation being refined by a directed mapping

Figure 1.7 shows how transformations, relations and mappings are placed within the MOF hierarchy. As `Transformation` is a super-type of `Relation` and `Mapping`, when we talk about a transformation we effectively mean ‘either a relation or a mapping, we don’t mind which one’. When we talk about a mapping, we specifically mean ‘a mapping and only a mapping’ – and similarly for relations.

The differentiation between specification and implementation is vital. In many complex applications of transformation technology it is often unfeasible to express a transformation in operational terms. For example, during the initial stages of system development, various choices which will effect an implementation may not have been made, and thus it may be undesirable to write an implementation at that stage. Another more general reason for the presence of specifications is that transformation implementations often carry around large amounts of baggage which, whilst vital to the transformations execution, obscure the important aspects

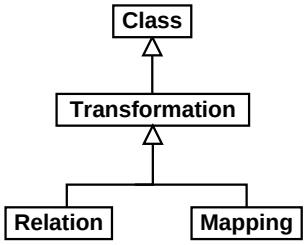

Figure 1.7.: Transformations, relations and mappings in the MOF hierarchy

of a transformation – by using specifications, these important aspects can be easily highlighted. Nevertheless, implementations are vital for the final delivered system. We also propose a standard operational transformation to prevent the need to drop to low level technologies such as the XML transformation system XSLT (XSL Transformations) [W3C99] – in order for transformations to be a successful and integral part of MDA, it is essential that they be modelled. Our proposal allows transformations to seamlessly and transparently evolve from specifications to implementations at any point during the development life cycle.

1.7.3. Transformation state

In many situations, simple transformations which perform a one step transformation are not sufficient. Transformations may need to build up large amounts of information whilst in the process of transforming – particularly if other transformations are involved in the process – and may also need to store information over transformations. A simple example of such a transformation is one which adds to elements a unique identifier based on an incremented counter. Although one could create a new object in the system to track the counter, it is far more natural and less cumbersome for the transformation itself to maintain the counter.

To this end, in our proposal all transformations have state, by virtue of the fact that `Transformation` subclasses `Class` in figure 1.7.

1.7.4. Domains

In the infrastructure, transformations can be specified between an arbitrary number of labelled domains. The primary constituent of a domain is a classifier such as `Class` or `Set{Package}`. Constraints within the domain on the classifier allow the domain to be arbitrarily restricted further. This simple definition of domains in infrastructure allows much richer notions to be built upon it in superstructure.

In the superstructure, domains as found in the infrastructure are not specified directly; see section 1.7.6 for more details.

1.7.5. Reusing transformations

Transformations can be reused either through the specialization mechanism (we recall that transformations are classes), by being referred to via attributes, or by the use of composite transformations. In either case, transformations retain the property of classes that they can be examined independently of their context e.g. if they are the parent of a sub-class.

Transformations can link to other transformations via the standard attribute mechanism. In such a case, the `owner` transformation can then used the `owned` transformations in whichever way it chooses.

In the infrastructure, there are various flavours of composite relations[2] (see figure 4.1 for their integration into the meta-model). Composite relations consist of a parent relation and an arbitrary number of component relations. The semantics of a specific type of composite relation determine how the component relations effect the parent. For example, an `and` composite relation requires that for the parent relation to hold between give elements, all of the component relations must hold as well. Furthermore, composite relations often impose restrictions on the domains of the parent and component relations. For example, in an `and` composite relation, the parent relation's domains must be a merge of all of the component relations domains; if the merge is not well defined, then the parent relation is ill-formed itself.

Figure 1.8 shows some types of relation reuse. The relation R_1 links to relation R_2 through an attribute. R_2 is an `and` composite relationship which has two component relationships R_3 and R_4. In order for an instance of R_2 to hold, both of its component instances R_3 and R_4 must hold as well.

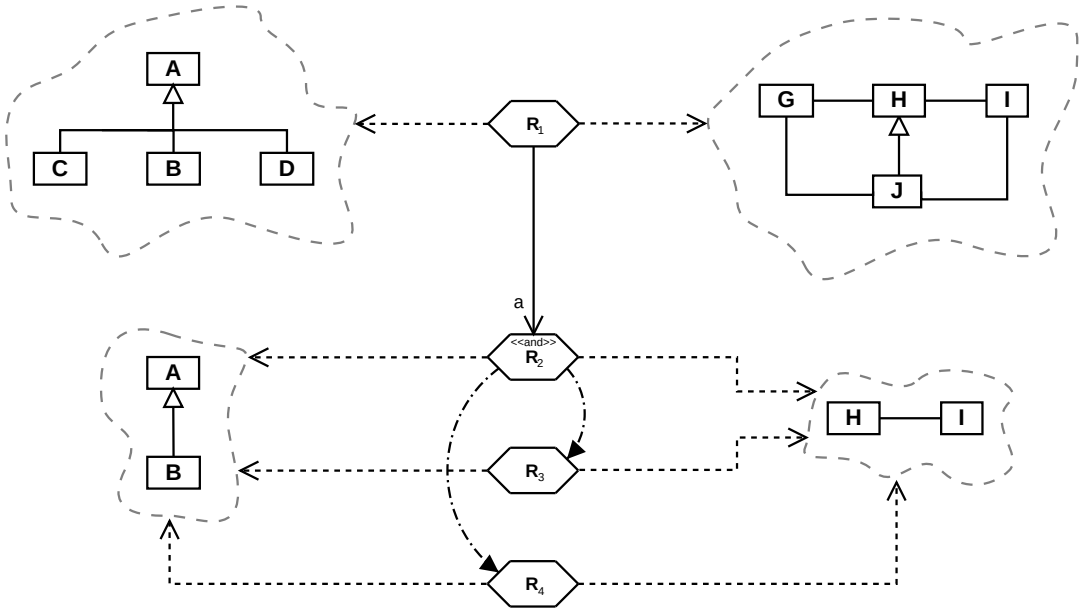

Figure 1.8.: Reusing relations

1.7.6. Pattern matching

Within the superstructure, we define powerful pattern matching languages for transformations. Pattern matching is a proven concept within transformation systems such as XSLT. Our pattern matching language allows model fragments to be matched against against meta-model patterns and used in transformations. In the superstructure, both relations and mappings can have patterns. Patterns expressed in mappings translate directly into the ASL.

[2] In due course we expect to extend this to some types of mapping as well.

1.7.7. A simple example

Figure 1.9 shows a simple example of a transformation `AtoX` which transforms a UML-like attribute into an XML element. The model in figure 1.9 shows a named attribute whose type is specified as a string, and an XML element which has named start and end tags, with the start tag containing `name = value` attributes, and the element consisting of a number of sub elements. Figure 1.10 shows an instance of the model.

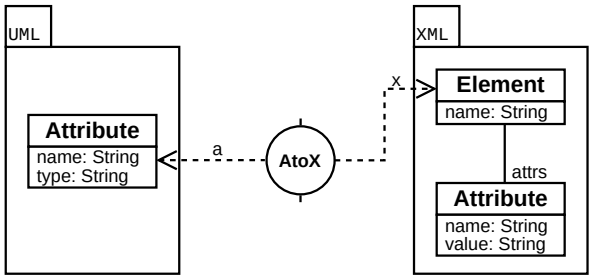

Figure 1.9.: Transforming a UML-like attribute into an XML element

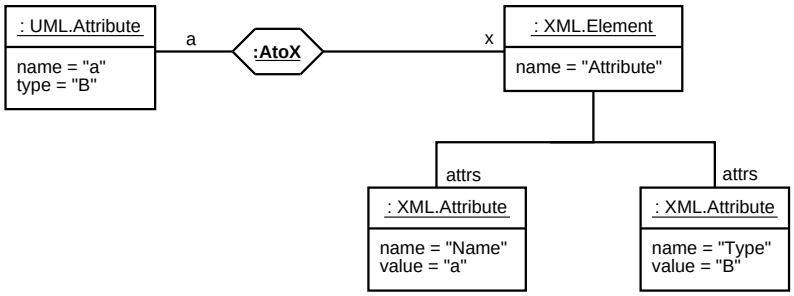

Figure 1.10.: An instance of figure 1.9 as a relation

At this point, note that we have not specified whether the transformation is a relation or a mapping – it could be either of these. Depending on whether the transformation actually represents a relation or a mapping, valid instances of figure 1.9 would either be an object model to be checked against the relation, or an input model to be automatically mapped into an output model. Figure 1.10 shows an example where `AtoX` is a relation. When written out in its more familiar textual concrete syntax, the XML in figure 1.10 would be as follows:

```
<Attribute name="a" type="b" />
```

Here is a sample relation for figure 1.9 written in OCL (note that the following relation holds in figure 1.10):

```
context atox relation:
  a.name = x.name and
  x.attrs->size() = 2 and
  x.attrs->exists(xa |
    xa.name = "type" and
    xa.value = a.type) and
  x.attrs->exists(xa |
    xa.name = "name" and
    xa.value = a.name) and
```

Here is a sample implementation for figure 1.9 written in pseudo-Java:

```
class atox {
  map_uml_attribute(UML.Attribute a) {
    x = new XML.Element();
    x.name = "Attribute";
    a1 = new XML.Attribute();
    a1.name = "Name";
    a1.value = a.name;
    x.attributes.append(a1);
    a2 = new XML.Attribute();
    a2.name = "Type";
    a2.value = a.type;
    x.attributes.append(a2);
  }
}
```

1.7.8. Summary

To summarize, here are the key concepts in our proposal for transformations:

- Our definition is split into a small core infrastructure, and a richer superstructure. Superstructure is translated into infrastructure.
- Transformation is a super type of both relation and mapping.
- Relations are fully declarative specifications. Mappings are operational implementations expressed in the ASL.
- We provide standard pattern matching languages which, when used with mappings, translate into the ASL.
- Transformations are stateful.
- Transformations can be specialized.
- Transformations can be arbitrarily used by other transformations.

2. Resolution of RFP requirements

2.1. Mandatory requirements

1. *Proposals shall define a language for querying models. The query language shall facilitate ad-hoc queries for selection and filtering of model elements, as well as for the selection of model elements that are the source of a transformation.*

Our proposal of a possibly extended version of OCL allows ad-hoc selection and filtering of model elements.

2. *Proposals shall define a language for transformation definitions. Transformation definitions shall describe relationships between a source MOF metamodel S, and a target MOF metamodel T, which can be used to generate a target model instance conforming to T from a source model instance conforming to S. The source and target metamodels may be the same metamodel.*

Mappings in both the infrastructure and the superstructure definitions allow generation of model T from S where the T and S may or may not share a meta-model.

3. *The abstract syntax for transformation, view and query definition languages shall be defined as MOF (version 2.0) metamodels.*

Our infrastructure definition is an extension of the existing MOF definition, and can be easily modified to support MOF 2.0, with minor modifications.

4. *The transformation definition language shall be capable of expressing all information required to generate a target model from a source model automatically.*

Both our infrasturcture and superstructure definitions are capable of expressing all the neccessary information for transformations.

5. *The transformation definition language shall enable the creation of a view of a metamodel.*

In our submission, views are created by transformations.

6. *The transformation definition language shall be declarative in order to support transformation execution with the following characteristic:*

    - *Incremental changes in a source model may be transformed into changes in a target model immediately.*

Our submission has the concept of relations which are fully declarative.

7. *All mechanisms specified in Proposals shall operate on model instances of metamodels defined using MOF version 2.0.*

2.2. Optional requirements

1. *Proposals may support transformation definitions that can be executed in two directions. There are two possible approaches:*

   - *Transformations are defined symmetrically, in contrast to transformations that are defined from source to target.*
   - *Two transformation definitions are defined where one is the inverse of the other.*

Relations allow transformations to be defined between any number of domains. Although they are not, in general, executeable, as relations are fully declarative, they place no restrition on which directon transformations go betweens.

Mappings are directed transformations, and can therefore one mapping can be an inverse of another.

2. *Proposals may support traceability of transformation executions made between source and target model elements.*

In the superstructure, transformation tasks can be used to trace transformation executions.

3. *Proposals may support mechanisms for reusing and extending generic transformation definitions. For example: Proposals may support generic definitions of transformations between general metaclasses that are automatically valid for all specialized metaclasses. This may include the overriding of the transformations defined on base metaclasses. Another solution could be support for transformation templates or patterns.*

Reuse is an integral part of our definition. Transformations may specialise one another. Composite transformations reuse transformations as their constituent components. Transformations may also have links to other transformations and control them in completely arbitrary ways.

4. *Proposals may support transactional transformation definitions in which parts of a transformation definition are identified as suitable for commit or rollback during execution.*

The superstructure definition contains support for defining transactions. A transformation task can be marked as transactional – its constituent transformations actions are either committed or rolled back together.

5. *Proposals may support the use of additional data, not contained in the source model, as input to the transformation definition, in order to generate a target model. In addition proposals may allow for the definition of default values for this data.*

Our submission places no constraints on whether or not implementations allow either relations or mappings to perform incremental updates.

6. *Proposals may support the execution of transformation definitions where the target model is the same as the source model; i.e. allow transformation definitions to define updates to existing models. For example a transformation definition may describe how to calculate values for derived model elements.*

Our definitions do not mandate whether transformations are update in place or functional copy.

2.3. Issues to be discussed

1. *The OMG CWM specification already has a defined transformation model that is being used in data warehousing. Submitters shall discuss how their transformation specifications compare to or reuse the support of mappings in CWM.*

Both the infrastructure and the superstructure definitions reuse parts of CWM. For example, both the infrastructure and the superstructure reuse familiar graphical concrete syntax from CWM. The strictly uni-directional nature of CWM transformations limits the amount of reuse possible from CWM, as our definitions of transformation are more flexible. Nevertheless we intend aligning with CWM as far as possible.

To give an idea of CWM's influence on our definition, we now present a few concrete examples of overlap. In the superstructure definition, transformation grouping elements such as `TransformationStep` and `TransformationTask` are identical to their CWM namesakes. `Relation` and `Domain` in our definition find their counterparts in `Transformation` and `DataObjectSet` in CWM.

2. *The OMG Action Semantics specification already has a mechanism for manipulating instances of UML model elements. Submitters shall discuss how their transformation specifications compare to or reuse the capabilities of the UML Action Semantics.*

It is a fundamental part of our infrastructure and superstructure definitions that mappings are expressed in terms of the ASL, allowing them to encompass all programming language definitions.

3. *How is the execution of a transformation definition to behave when the source model is not well-formed (according to the applicable constraints?). Also should transformation definitions be able to define their own preconditions. In that case: What's the effect of them not being met? What if a transformation definition applied to a well-formed model does not produce a well-formed output model (that meets the constraints applicable to the target metamodel)?*

In our definition, transformations may specify preconditions.

4. *Proposals shall discuss the implications of transformations in the presence of incremental changes to the source and/or target models.*

2.4. Relationship to Existing OMG Specifications

Object Constraint Language (OCL) OCL is used extensively throughout the submission and forms the basis of our query language.

Meta Object Facility (MOF) The infrastructure meta-model is a simple extension to MOF; the superstructure is a slightly more involved extension of MOF. Both the infrastructure and the superstructure definitions can therefore be considered as a new member of the MOF based family of languages that currently includes UML and CWM amongst others.

Common Warehouse Metamodel (CWM) See section 2.3.

Action Semantics Language (ASL) See section 2.3.

Part II.

TECHNICAL DETAILS

3. Overview

Our definition of transformations is split into two parts:

Infrastructure The infrastructure is the small semantic core of our definition. It is not intended for end users, although it is also useful for tool vendors who wish to provide a ‘lowest common denominator’ option. The infrastructure is a purposefully small extension to existing OMG standards, and has a simply defined semantics.

Superstructure The superstructure is the semantically and syntactically rich part of the definition. It is intended for end users. The semantics of the superstructure are given by its translation into the infrastructure.

Since the superstructure is effectively defined in terms of the infrastructure, the definition of the infrasturcture is critical for the overall proposal. Furthermore, the specification of the superstructure to infrastructure translation is given separately from either definition.

4. Infrastructure

Our definition of infrastructure is divided into separate definitions of abstract syntax, semantic domain, and semantic relation[1] in a similar manner to other specifications such as [ADP03]. For further details about this approach to language definition see [CEK02].

4.1. Abstract syntax

Figure 4.1 shows the infrastructure abstract syntax package. This package can be merged with the standard MOF definition to produce an extended version of MOF. Original MOF elements are shown in grey; our new elements are in black.

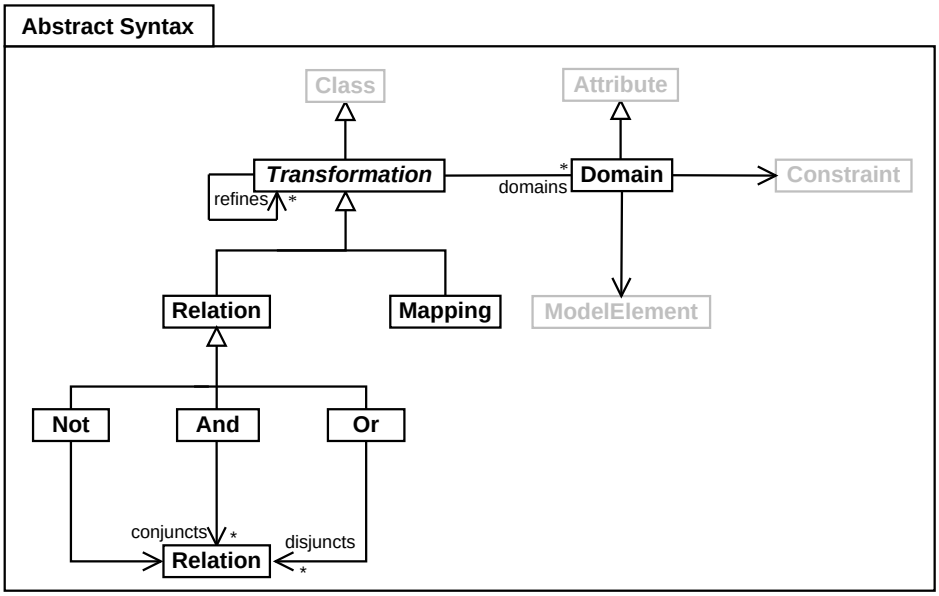

Figure 4.1.: Transformations abstract syntax extension to the MOF meta-model

Transformations contain a number of domains. `Domain` is a subtype of `Attribute` in order that each domain is named. Transformations can refine an arbitrary number of other transformations (though note that well-formedness rules restrict this association); that is, the refining transformation can be said to be in some way conformant to the refined transformation. `Transformation` is abstract and can not be directly instantiated: every transformation is actually a relation or a mapping. There are currently three types of composite relations: `Not`, `And` and `Or`. Each composite relation type links to component relations.

[1] We note that in other documents that use a similar approach to semantic definition, this was called ‘semantic mapping’; in our terminology it is a relation rather a mapping.

4.1.1. Operations

In the following operation definitions, we use the convention that `super.`*x* calls the method *x* in the superclass of the current class with `self` bound to the current object.

One part of our definition deviates significantly from the norm. For model instances, we can not allow the standard practice of constraints effectively being evaluated by an unknown external system at unknown intervals, especially as the failure of a constraint immediately triggers an exception. This behaviour would cause significant problems in our model since, for example, the semantics of the `Not` composite relation are that a composite relation is satisfied when the constraints on it's component relation fail. We therefore assume that all model elements have an operation `satisfiedBy`, which returns `true` or `false` to denote well-formedness with respect to a particular model instance: this allows the constraints on a component relation of a `Not` to fail, but the parent relation to still hold. The assumption then is that an observer then calls the `satisfiedBy` method to determine if a model element is well-formed.

All constraints on a relation must be satisfied in order for it to hold:

```
context Relation::satisfiedBy(i : RelationInstance) : Boolean
  self.constraints->forAll(c |
    c.satisfiedBy(self)))
```

All component relations must be satisfied in order for an `And` relation to hold:

```
context And::satisfiedBy(i : AndInstance) : Boolean
  super.satisfiedBy(i) and
  i.conjuncts->forAll(r |
    r.of.satisfiedBy(r))
```

At least one component relation must be satisfied in order for an `Or` relation to hold:

```
context Or::satisfiedBy(i : OrInstance) : Boolean
  super.satisfiedBy(i) and
  i.conjuncts->exists(r |
    r.of.satisfiedBy(r))
```

The component relation must not be satisfied in order for a `Not` relation to hold:

```
context Not::satisfiedBy(i : NotInstance) : Boolean
  super.satisfiedBy(i) and
  not self.transformation.satisfiedBy(i.relationInstance)
```

4.1.2. Well-formedness rules

Attributes and domains must all be uniquely named within a transformation:

```
context Transformation inv:
  self.domains->forAll(d |
    not self.attributes->exists(a |
      d.name = a.name))
```

A relation can not refine a mapping, because mappings are operational:

```
context Relation inv:
  not self.refines->exists(t |
    t.isKindOf(Mapping))
```

The domains of an `And` are a superset of the merge of all its conjuncts' domains. `merge` finds the most specific element of all domains with the same name:

```
context And inv:
  merge(self.conjuncts.domains)->forAll(d |
    self.domain->exists(d' |
      d.name = d'.name and
      d.classifier.isSuperTypeOf(d'.classifier)))
```

The domains of an `Or` are a superset of the merge of all its disjuncts' domains.

```
context Or inv:
  merge(self.conjuncts.domains)->forAll(d |
    self.domain->exists(d' |
      d.name = d'.name and
      d.classifier.isSuperTypeOf(d'.classifier)))
```

The domains of a `Not` are a superset of its component.

```
context Not inv:
  self.relation.domains->forAll(d |
    self.domain->exists(d' |
      d.name = d'.name and
      d.classifier.isSuperTypeOf(d'.classifier)))
```

4.2. Semantic domain

Figure 4.2 shows the semantic domain for transformations.

4.3. Semantic relation

Figure 4.3 shows the semantic relation for transformations.

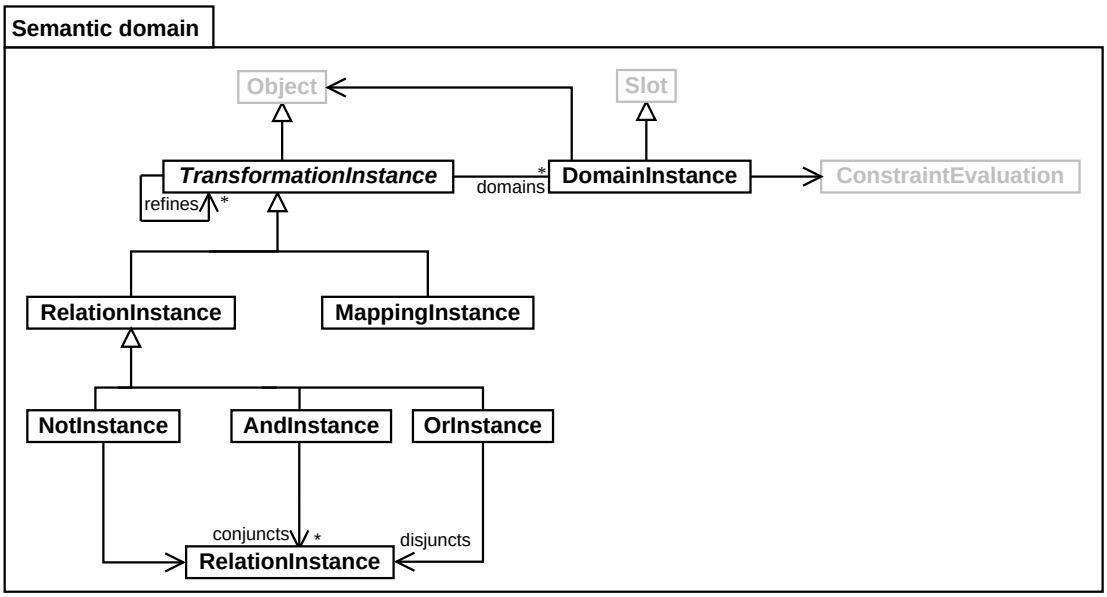

Figure 4.2.: Transformations semantic domain

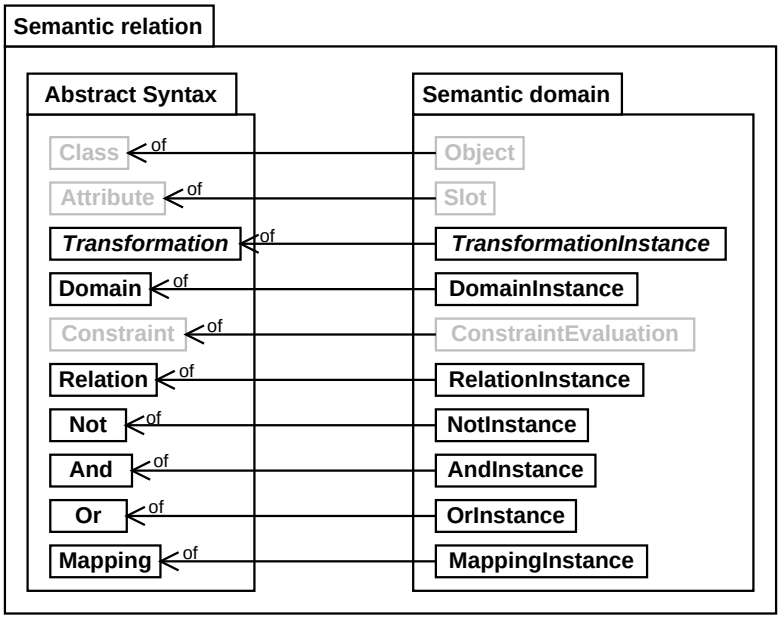

Figure 4.3.: Transformations semantic relation

5. Superstructure

5.1. Relations

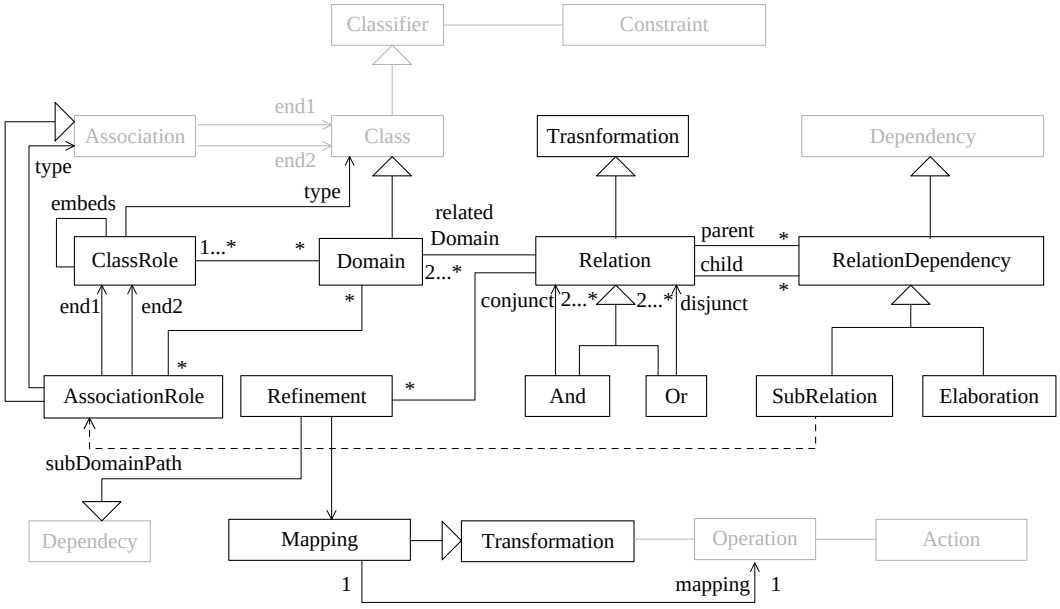

Figure 5.1.: Relations meta-model

Figure 5.1 shows a transformations meta-model that extends the transformations meta-model given in the infrastructure. The elements `Transformation`, `Relation`, `Domain`, `And`, `Or` and `Mapping` inherit from and extend the corresponding elements in the infrastructure. This model extends the MOF core model. Elements from MOF core are shown in gray.

The heart of the model is the element `Relation`. It specifies a relationship that holds between instance models of two or more `Domains`. Each `Domain` is a view of the meta-model, and is constituted of `Class` and `association roles`. A `Role` has a corresponding type that the elements bound to it must satisfy. A `Domain` may also have an associated query to further constrain the model specified by it. The query may be specified as an OCL expression. A `Relation` also may have an associated OCL specification. This may be used to specify the relationship that holds between the different attribute values of the participating domains. A binary directed-relation is a special case with a `source Domain` and a `target Domain`. An example transformation specification is shown in the figure 5.2. It consists of two relations - `ClassToTable` and `AttributeToColumn`. `ClassToTable` specifies a relationship between two domains, one being constituted of `Class` and the other constituted of `Table`. The OCL specification associated with the `Relation` specifies the relationship between class name and the corresponding table name.

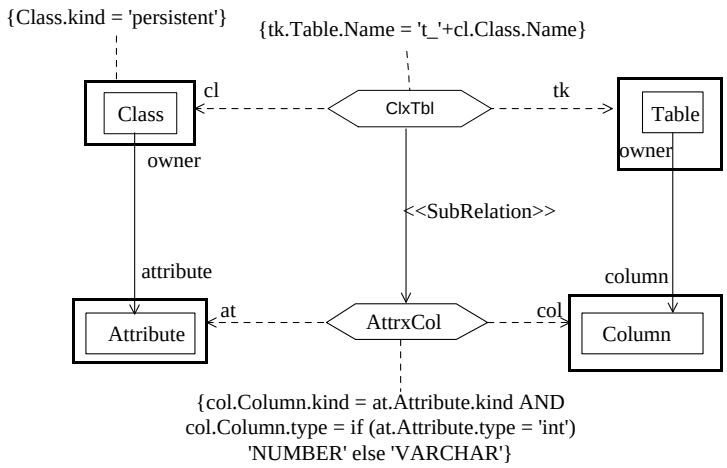

Figure 5.2.: An example relation specification

5.1.1. Structuring Relations

Complex relations can be built either by composing simpler relations using `And` and `Or` or by structuring them using `SubRelation` and `Elaboration`. In the figure 5.2, the `Attribute` to `Column` relation is a sub-relation of `Class` to `Table` relation. This specifies two things - the relation `ClxTbl` is incomplete without the relation `AttrxCol`, and the relation `AttrxCol` should hold for all `Attributes` of the corresponding `Class`. These are explained in detail below and also through examples in appendix A.

5.1.2. Refinement

A set of operational `Mappings` may be used to refine a `Relation`. There may be a `Mapping` between each pair of domains (source and target) participating in the relation. Each `Mapping` specifies how the target model is obtained from the source model.

5.1.3. Meta-entity overview

We now present a brief overview of the new meta-entities in figure 5.1.

Relation

Attributes

`isAbstract = {true, false}`

An abstract relation must be elaborated by a set of relations.

Associations

`constraint → Constraint`

is used to specify the relationship between attribute values of the participant domains.

`relatedDomain → Domain`

is used to specify the meta-model views participating in the `Relation`.

`client` → `RelationDependency`
is used to structure relations by either Elaborations or SubRelations.

A relation is a `SubRelation` of another relation if the relationship specification of the latter is incomplete without the relationship specification of the former. `SubRelation` also has a set of associations between the parent domain and the child domain. Intuitively, a `SubRelation` dependency specifies that a relationship holds between a set of domain elements only if all the sub relationships hold between all the reachable domain elements of the sub relations.

A `Elaboration` dependency specifies how an 'abstract' relation is elaborated in terms of a set of detailed relationship specifications at the next level. This allows a transformation specification to be organized in a hierarchy.

And

An `And` relation has a set of associated `conjunct Relations`. An `And` relation is satisfied if and only if all of its conjuncts are satisfied.

Or

An `And` relation has a set of associated `disjunct Relations`. An `Or` relation is satisfied if and only if at least one of its disjuncts is satisfied.

Domain

A `Domain` identifies one component of a set of related domains participating in a relationship. A `Domain` is constituted of one or more classes and associations. The same class or association may occur more than once in a domain. The different occurrences are distinguished by means of roles.

Associations

`composedFrom` → `ClassRole`

`composedFrom` → `AssociationRole`
Specifies the class and association roles of which a domain is composed.

`constraint` → `Constraint`
Specifies further constraints on the model elements bound to the roles of the domain.

Constraints

The class roles and association roles should commute with their corresponding classes and associations.

ClassRole

Associations

`type` → `Class`
Specifies the type of the role.

`embeds` → `ClassRole`
In auto-transformations, where modifications are performed logically in-place (i.e. in the same extent),

the question arises as to what should be done with classes and associations not covered in the transformation specification. This is not an issue in cross-model (source and target extents are distinct) transformations as one is only interested in classes and associations covered in the transformation specification. One alternative is to explicitly include all associations of a covered class in the specification, with some kind of identity transformation. But this is cumbersome. 'Embeds' association provides another alternative. It specifies that all unspecified associations of an element bound to a class role are to be transferred on to the element bound to the corresponding class role in the destination model. In effect it specifies how mapped target model elements are to be embedded in the rest of the model.

AssociationRole

Associations

`type` $\rightarrow$ `Association`

Specifies the type of the role.

Action

This is a *plug point* for the standard Action Semantics. Our pattern matching language will translate into the action semantics.

Mapping

A mapping is a class with a distinguished operation mapping. Instances of a mapping are sent a mapping message along with instances of the domains of the mapping. The result is an instance of the range of the mapping. Note that since the range type of a mapping can be a collection, it is possible for mappings to produce multiple results.

Refinement

Relationship specifications can be refined into implementations by translating them into mappings.

Constraint

This is a standard OCL constraint but with a specific method *check*.

5.1.4. Well-formedness rules

```
context AssociationRole inv:
  self.end1.type = self.type.end1
  and
  self.end2.type = self.type.end2

context SubRelation inv:
  self.subDomainPath->exists(ar:AssociationRole |
    self.parent.relatedDomain.class->exists(c:ClassRole |
          ar.end1 = c))
  and
```

```
self.subDomainPath->exists(ar:AssociationRole |
  self.child.relatedDomain.class->exists(c:ClassRole |
        ar.end2 = c))
```

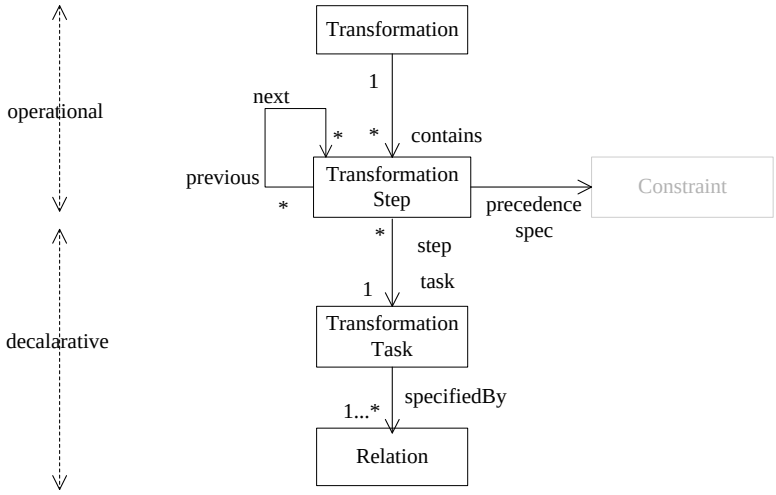

Figure 5.3.: Transformation organization meta-model

5.1.5. Transformation organization meta-model

Figure 5.3. shows a meta-model for organizing a large transformation activity into a set of transformation steps and their flow composition. This model adds an additional operational super-structure over the declarative relationship specifications described above.

Transformation Task is specified by a set of relations. It is a transactional unit. It can specify if the `extents` of the source and target models are the same (auto) or different (cross-model).

Attributes

`isTransactional = {true, false}`
specifies whether a transformation task is transactional - the transformations performed in such a transformation task are either committed or rolled back together.

`isAuto = {true, false}`
specifies whether the transformation task performs auto transformations. In an auto-transformation the target model is the same as the source model.

5.1.6. Examples

Figure 5.4 shows the concrete syntax used in the transformation diagrams. Meta-model diagrams use UML syntax.

Please note that the transformation diagrams also show the required OCL constraints. As explained in section 1.7.1 these constraints are further augmented when the super-structure models are translated down to infrastructure models; these additional constraints are derived from the semantics of various relation composition constructs.

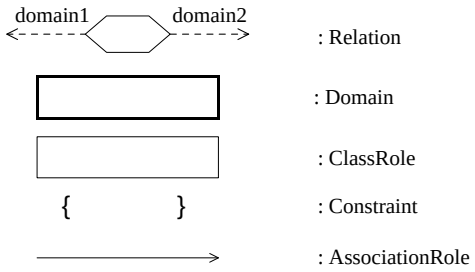

Figure 5.4.: Concrete syntax

Object-relational mapping

This example describes a transformation between a UML model and a RDBMS model. The sections below describe the relevant portions of the meta-models of the object and RDBMS models, their mapping requirements, and a transformation specification that satisfies the requirements.

A sample UML meta-model

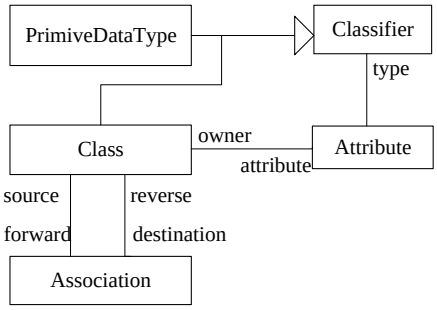

Figure 5.5.: A sample UML meta-model

A sample UML meta-model is shown in figure 5.5. A class has attributes. An attribute's type can be either a primitive data type or another class (complex types). Classes are related to each other through Association objects. Only classes that are marked as `persistent` for the property `kind` are considered for Some attributes have the property `kind` set to `Primary` to indicate that they are the key attributes.

A sample RDBMS meta-model

A sample RDBMS meta-model is shown in figure 5.6. A table has columns. Every table has a mandatory primary key (Key). A table may optionally have foreign keys. A foreign key refers to a primary key of another

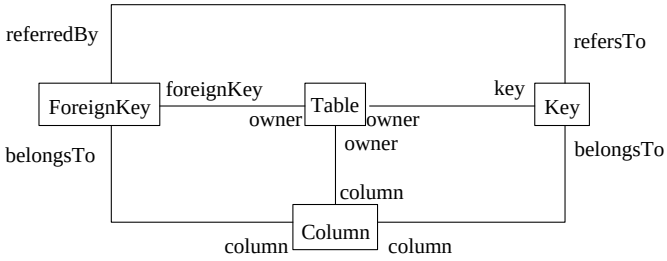

Figure 5.6.: A sample RDBMS meta-model

associated table.

Mapping requirements

A class maps on to a single table. A class attribute of primitive type maps on to a column of the table. Attributes of a complex type are drilled down to the leaf-level primitive type attributes; each such primitive type attribute maps onto a column of the table. An association maps on to a foreign key of the table corresponding to the source of the association. The foreign key refers to the primary key of the table corresponding to the destination of the association.

Transformation specification

Figure 5.7 shows the UML to RDBMS transformation structured as an And composition of two transformations 'Class to table transformation' and 'Association to foreign-key transformation'. Figure 5.8 shows the detailed specification of 'Class to table transformation' and figure 5.9 shows the detailed specifications of 'Association to foreign-key transformation'.

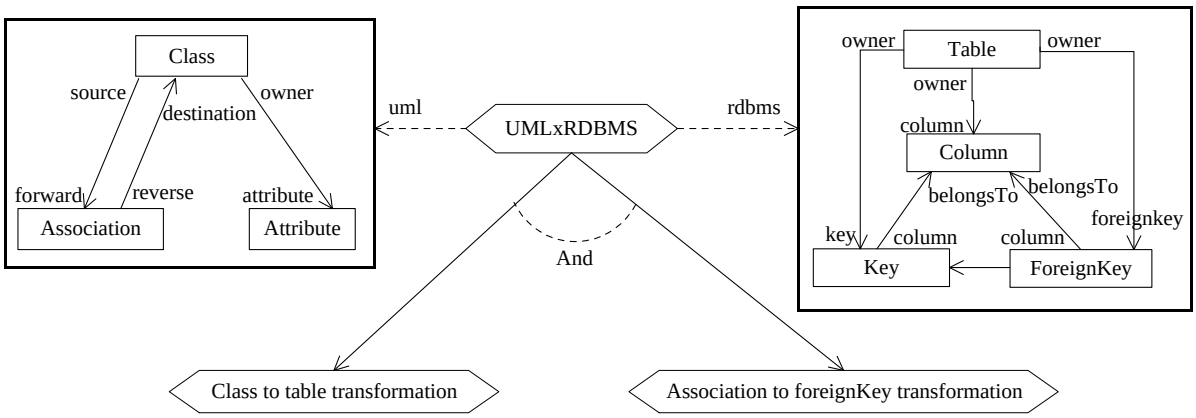

Figure 5.7.: UML to RDBMS mapping

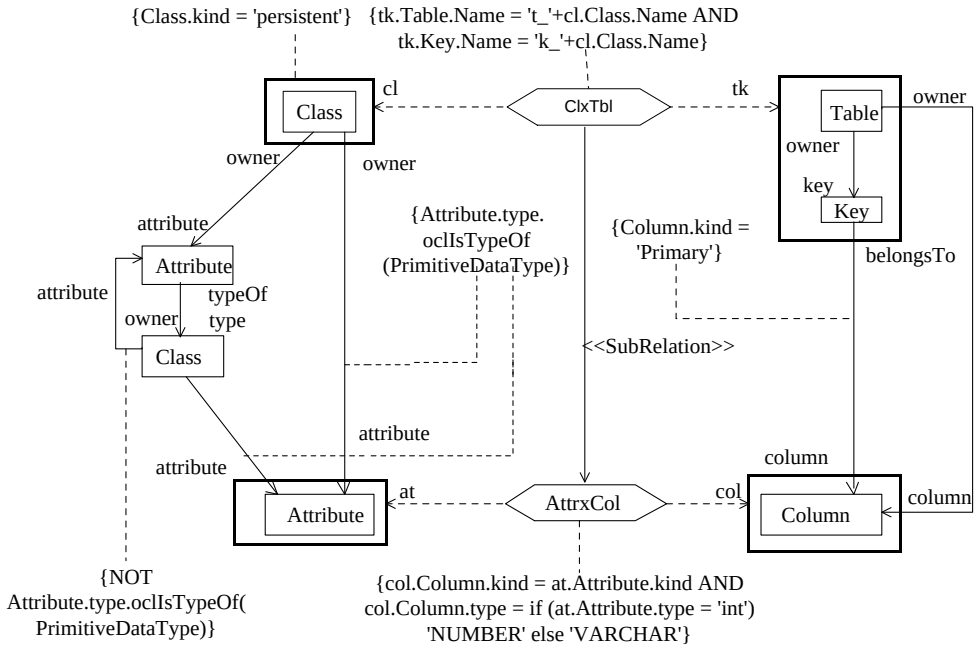

Figure 5.8.: Class to table transformation

5.1.7. Remove multiple inheritance

In this example we transform an object model that supports multiple inheritance into an object model that supports interfaces but does not support multiple inheritance. This is given as an example of auto transformation where a model is modified in-situ. Figure 5.10 shows two transformation tasks `Create Interfaces` and `Remove Inheritance`. Please note that the concrete syntax notation <<Task>> represents the combination of abstract syntax elements 'Transformation Step' and 'Transformation Task'. In figures 5.11 and 5.12 a number in brackets represents an 'embeds' association between the corresponding elements of the mapped domains.

Task 1: Create Interfaces

For each class create an interface containing the same set of operations. Figure 5.11 shows this transformation.

Task 2: Remove Inheritance

Copy the operations of a super-class down to its sub-class and remove the inheritance. A sub-class implements the interfaces of its super class. Figure 5.12 shows this transformation.

5.2. Mappings

To be completed.

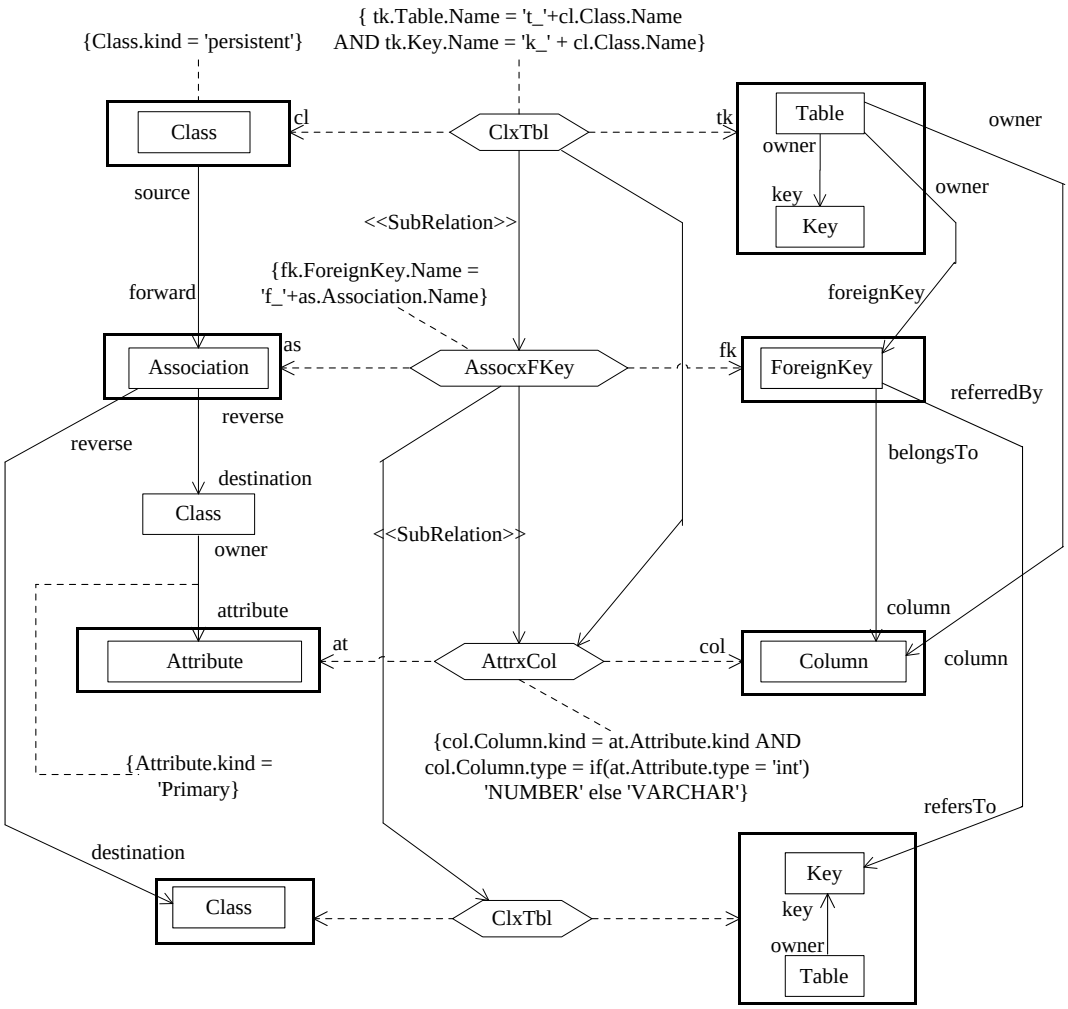

Figure 5.9.: Association to foreignKey transformation

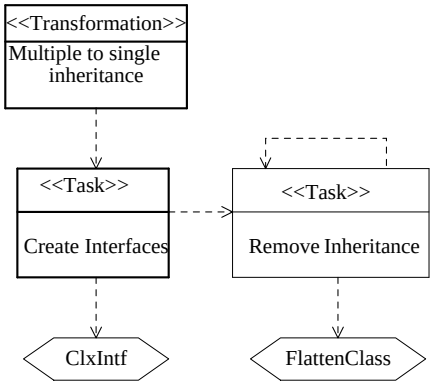

Figure 5.10.: Remove multiple inheritance (transformation organization)

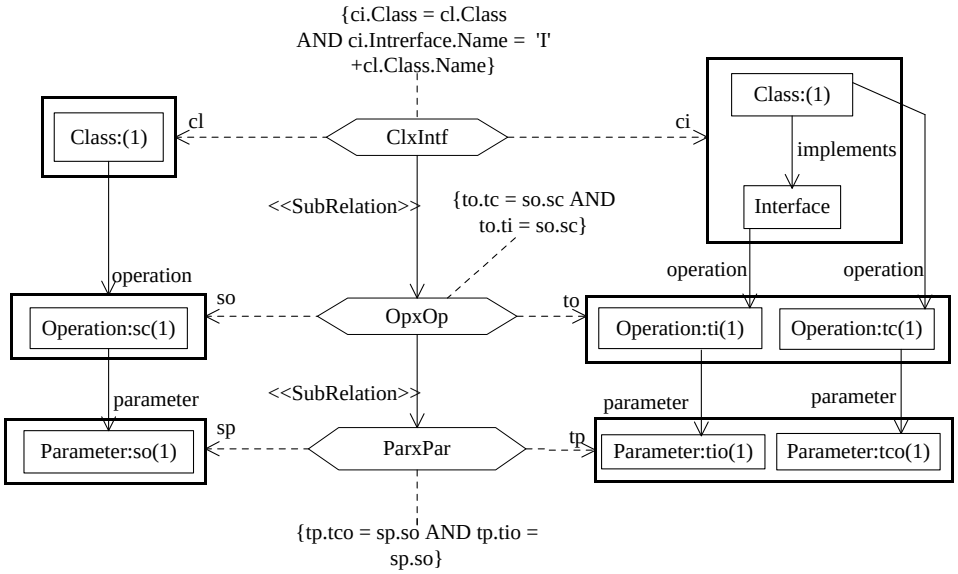

Figure 5.11.: Create interfaces

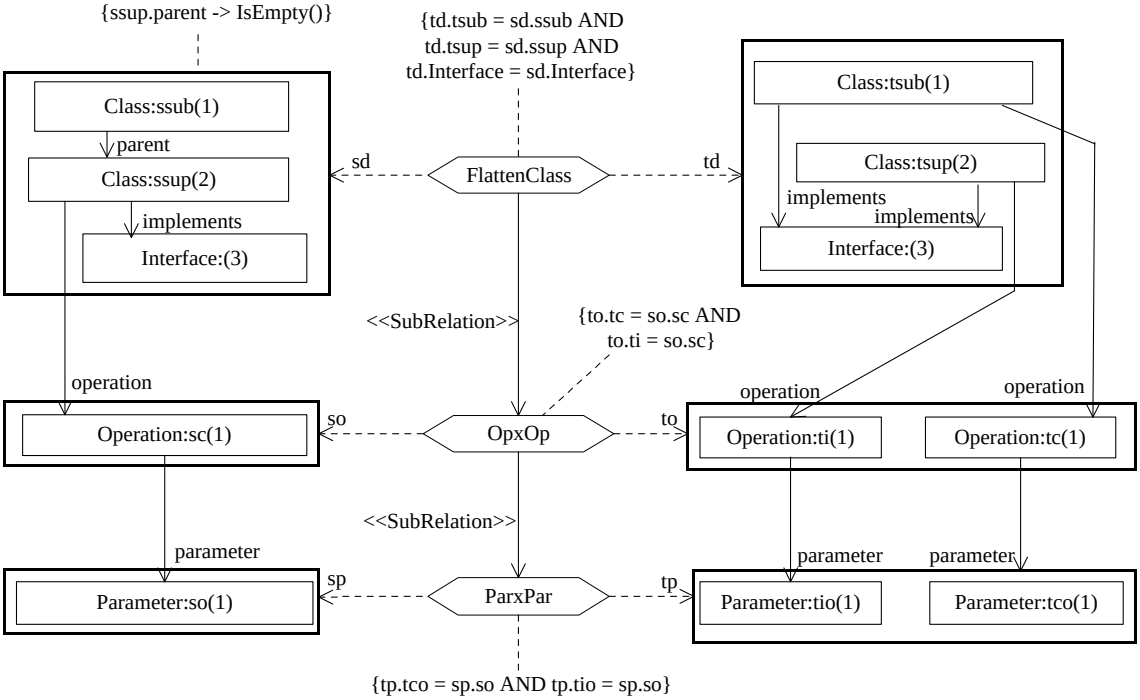

Figure 5.12.: Remove inheritance

6. Superstructure translation

This section will detail the translation of the superstructure into the infrastructure. This definition will be given in terms of our proposal in order to show the power of our proposal. The overall idea is simple: concepts which exist in the superstructure but not in the infrastructure will be translated into infrastructure concepts, with added OCL constraints or ASL as appropriate, in order to achieve the correct semantics. Such translations have precedent within the OMG community: the UML2 specification is similarly split into infrastructure and superstructure, and some submissions to UML2 such as [AD02] have given a translation from superstructure to infrastructure.

To give a small overview of how a part of superstructure might translate into infrastructure, take a relations domains. In the superstructure, these are rich pattern matching constructs, whereas in the infrastructure they are rather more spartan single model elements, with constraints. When translating from superstructure to infrastructure, a translation would first strip away all the pattern matching constructs and then analyze the resulting elements to find their most specific shared parent. This element then forms the basis of the infrastructure translation. The pattern matching constructs, and other structural information, are then translated into OCL constraints and added into the infrastructure domain in order to recover the information which has otherwise have been lost.

7. Compliance points

This chapter will list compliance points which state precisely which features implementations must support and which features implementations may optionally support.

Part III.

APPENDICES

A. Examples

This chapter will be filled with representative examples of different types of transformations.

A.1. DOS file system to UNIX file system

A.2. Superstructure operator

A.3. Multiple to single inheritance

A.4. Classes to tables

A.5. Information system model to J2EE

B. Glossary

ASL The Action Semantics Language (ASL) [OMG01] is an extension of UML which allows executeable models.

Composite relation A composite relation consists of a parent relation and a number of *component relations*. There are various types of composite relation. For example, the semantics of an `and` composite relation are that in order for the parent relation to hold, all of the component relations must hold.

Component relation A component relation is a component in a *composite relation*.

Domain Transformation are specified between a number of domains. In the infrastructure, a domain is a simple `Classifier`; in the superstructure, domains are much richer.

Infrastructure The infrastructure is the 'core kernel' of our definition. Not intended for end users, it provides a simple semantic core; it is also useful for tool vendors.

Mapping A mapping is a potentially directed transformation implementations.

MOF The Meta Object Facility is the core of the OMG's modelling work.

Pattern matching Pattern matching is a process whereby parts of a model are bound to pattern variables. The pattern can restrict exactly what models it will match against. XSLT is an example of a transformation language based on pattern macthing.

Query A query takes as input a model, and selects specific elements from that model.

Relation are multi-directional declarative transformation specifications.

Superstructure The superstructure is the rich part of the definition which end users use. The semantics of the superstructure are given by its translation into *infrastructure*.

Transformation Transformation is the umbrella term for *relation* and *mapping*.

View A view is a model that is derived from another model.

Bibliography

[AD02] Adaptive and Data Access. *2U revised submission to UML 2.0 Superstructure*, 2002. `http://cgi.omg.org/cgi-bin/doc?ad/02-12-23`.

[ADP03] Adaptive and Data Access and and Project Technology. *Unambiguous UML (2U) 3rd Revised Submission to UML 2 Infrastructure RFP*, 2003. `http://cgi.omg.org/cgi-bin/doc?ad/03-01-08`.

[BMR95] Andrew Berry, Zoran Milosevic, and Kerry Raymond. Reference model of open distributed processing: Overview and guide to use, 1995. `ftp://ftp.dstc.edu.au/pub/DSTC/arch/RM-ODP/PDFdocs/part1.pdf`.

[CEK02] Tony Clark, Andy Evans, and Stuart Kent. Engineering modelling languages: A precise meta-modelling approach. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE 2002*, volume 2306 of *Lecture Notes in Computer Science*, 2002.

[DSo01] Desmond DSouza. Model-driven architecture and integration - opportunities and challenges, 2001. `http://www.kinetium.com/catalysis-org/publications/papers/2001-mda-reqs-desmond-6.pdf`.

[MC99] Luis Mandel and Mariá Cengarle. On the expressive power of the object constraint language ocl. In *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *LNCS*, pages 854 – 874. Springer, 1999.

[OMG01] Object Management Group. *Action Semantics for the UML*, 2001. `ad/2001-08-04`.

[OMG02] Object Management Group. *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, 2002. `ad/2002-04-10`.

[W3C99] W3C. *XSL Transformations (XSLT)*, 1999. `http://www.w3.org/TR/xslt`.