Revised submission for MOF 2.0 Query / Views / Transformations RFP

Q^{V}T-*Partners*

`http://qvtp.org/`

Version 1.1 (2003/08/18)

Submitted by:

Tata Consultancy Services

Supported by:

Artisan Software
Colorado State University
Kinetium
King's College London
University of York

Revised submission for MOF 2.0 Query / Views / Transformations RFP

Q^{V}T-*Partners*

1.1 (2003/08/18)

Contents

Preface

This version of the submission

This is a revised submission to the QVT RFP. Although many parts of the document are in a very polished state, other parts remain to be filled in. Later sections in this preface contain more details of the changes since the previous version of the submission, and the intended future direction respectively.

This version of the submission was generated at 20:40 BST on August 18, 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 announcement `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 3). This later section also includes a summary of our submissions compliance with existing OMG standards such as CWM (section 3.4).

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

There are also a number of appendices which include extra technical information and examples.

Statement of proof of concept

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

One of the QVT-*Partners*, TCS, has used the experience gained from its product *MasterCraft* in preparing it's contributions to the submission. *MasterCraft* has been successfully used to develop several large enterprise-scale applications that run into several millions of lines of code. By providing a modelling framework that provides support for modelling GUI, database, process, architecture and several design strategies, *MasterCraft* has been successfully used to develop large enterprise-scale applications that run into several millions of lines of code. Using a model transformation language called *SpecL*, a number of code generation tools transform models into various platform specific implementations. *MasterCraft* has been extended to support most of the visual notation and the meta-model presented in this submission: TCS is currently working on extending *MasterCraft* and *SpecL* with the new concepts and notations proposed in this version of the submission. The experience gained from the implementation of the submission in *MasterCraft* has enabled the identification and testing of the essential requirements that a good transformations framework must support. The object-relational example presented in this submission is a small part drawn from the data manager generator tool of MasterCraft.

Resolution of RFP requirements and requests

See section 3 on page 19.

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/`
Colorado State University	`http://www.cs.colostate.edu/`
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 Appukuttan	`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`

Version history

This section gives a history of all official releases of this document. The history is presented in reverse chronological order.

Revised submission (2003/08/18)

First revised submission submitted to the OMG. Colorado State University announce their support of the Q^{V}T-*Partners* submission.

This version is a major update to our initial submission, and most sections have been substantially revised. Specific major updates include:

- Addition of executive summary.
- Addition of concrete textual syntax.
- Inclusion of a new powerful pattern language.
- Complete infrastructure, including syntax and precisely defined semantics.
- Additional examples, and significant reworking of existing examples.

Initial submission (2003/03/03)

First version submitted to the OMG.

Future direction

For the next revised submission we anticipate further development of our superstructure language and a more complete superstructure to infrastructure translation.

Executive summary

This submission presents the Q^VT-*Partners* proposal for the MOF 2.0 QVT standard. The proposal consists of a number of key ingredients which we briefly discuss in this section.

Specification and implementation

A common scenario in the development of any artifact is to first create a specification of the form and behaviour of the the artifact, and then realise an implementation which satisfies the specification. The specification is characterised by a lack of implementation details, but having a close correspondence to the requirements; conversely an implementation may lack close correspondence to the requirements.

This submission maintains this important distinction. *Relations* provide a specification oriented view of the relationship between models and are specified in a language that can be easily understood. They say what it means to translate between several models but without saying precisely how the translation is achieved. Those details are realised by *mappings* which characterise the means by which models are translated. It should be noted though, that while the *mappings* language is rich enough to provide an implementation of *relations* it also manages to maintain a requirements oriented focus. This may give rise to a scenario where developers prefer to omit *relations* and directly define *mappings*.

See section 2.4 on page 8, and chapter 5 on page 27.

Scalability and reuse

Decomposition is a key approach to managing complexity. This submission provides a number of composition mechanisms whereby *relations* and *mappings* can be composed to form more complex specifications. These mechanisms also aid reuse since *mappings* and *relations* can be treated as reusable components which are composed for specific contexts.

See section 2.8 on page 12, and chapter 5 on page 27.

Usability

Diagrammatic notations have been important to the success of many OMG standards. This proposal presents a diagrammatic notation which is an extension of collaboration object diagrams and is therefore familiar to many end users. A criticism often levelled at diagrammatic notations is their scalability. This submission also presents a textual syntax, constructs of the diagrammatic notations are closely aligned with its textual counterpart.

Considering the domains of *relations* and *mappings* at the generic type level is often too limiting. Instead it often is specific-types of things that are of interest. This submission uses patterns to describe the domains of both *relations* and *mappings*. Patterns are a means of succinctly describing specific-types of model elements and enable domains of interest to be rapidly stated with ease.

See section 2.6.1 on page 10, and section 5.7 on page 42.

Semantic soundness

By definition a standard should give rise to consistency across differing implementations. It is important that an end user can get the same results on two different implementations. For this reason, this submission goes to some effort to ensure that all the constructs have a well-defined semantic basis. This is achieved by treating the submission in two parts. The *infrastructure* part has a small number of constructs which can be easily and consistently understood from informal descriptions (although a mathematical semantics is given in Appendix B for the sake of completeness and rigour). The *superstructure* part uses the *infrastructure* as its semantic basis and defines the syntax that the end user deals with. The relationship between the *superstructure* and the *infrastructure* is expressed as a translation.

See section 2.9 on page 13, and chapter 7 on page 59.

Part I.

RESPONSE TO THE RFP

Chapter 1.

Introduction

This document is the revised Q^{V}T-*Partners* 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). These three aspects of the RFP address the following wider issue: the ability to manipulate models is necessary to ensure the full realization of MDA. The QVT RFP seeks a standard solution for model manipulation.

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 take as input a model and update it or create a new model.

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 the case of a transformation. In contrast, views themselves are models and are inherently related to the model from which they are created. 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 other words, 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 a single 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 on MDA and introduces the concepts of horizontal and vertical dimensions within modelling. The vertical dimension represents changing levels of abstraction in a particular part system; the 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 from the vertical dimension.

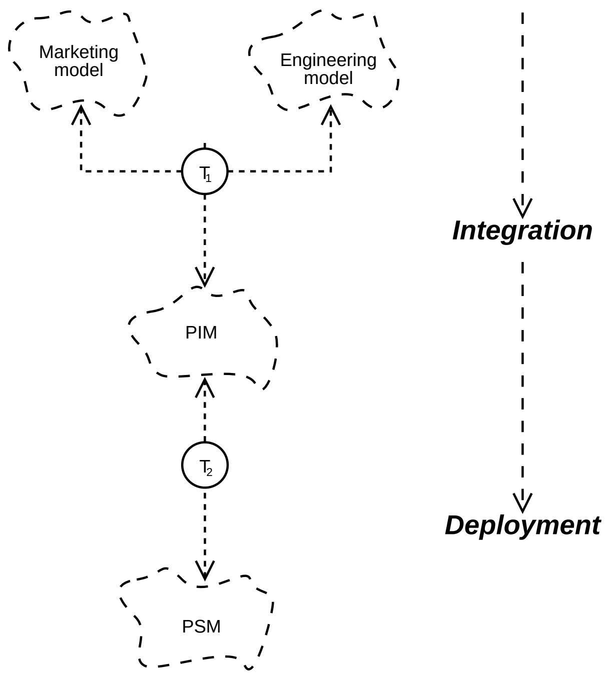


Figure 1.1.: Transformations and MDA

No matter what perspective is taken of MDA, 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. At the top, it shows a company split along the horizontal axis into multiple departments, each of which has a model of its system. Each model of a department can be considered to be a view on an overall system PIM. The system PIM can be converted into a system PSM which would form the basis of the real-world system

[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.

that the company would utilize. 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.

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

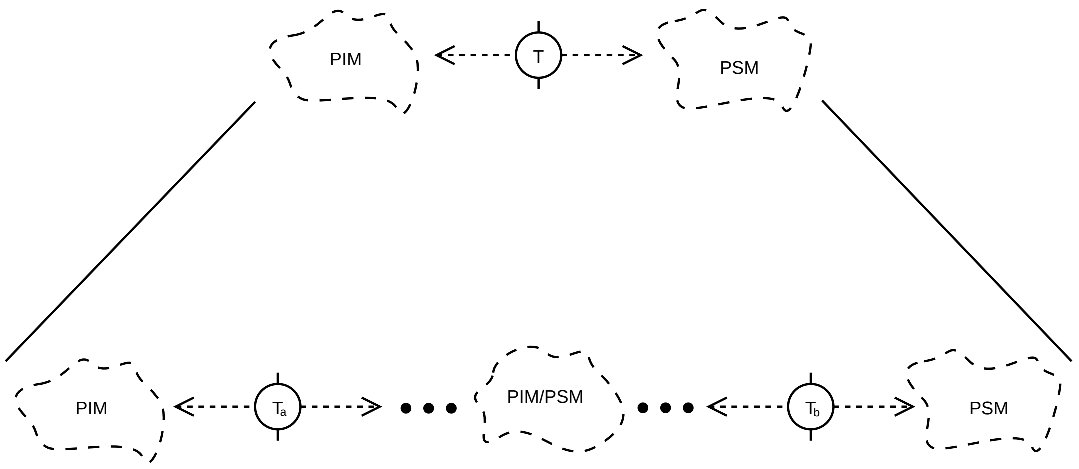


Figure 1.2.: An expanded PIM to PSM transformation

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. Applications of transformations

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' and/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 a programming language such as C++, Java or Python or a system architecture such as J2EE or .net. As shown in figure 1.2, the conversion may not necessarily happen in a single stage.
- Reverse engineering. For example, a tool which generates UML models from Java class files. Reverse engineering often involves the recovery of abstract models which are free from extraneous detail.

- Technology migration. Whereas reverse engineering is an attempt to recover lost information, technology migration is concerned with converting systems from legacy to modern platforms. For example, a tool which migrates legacy COBOL code to Java.
- Legacy tool integration. For example, the integration of various legacy backends when creating an e-commerce web site.

1.3. A general scenario

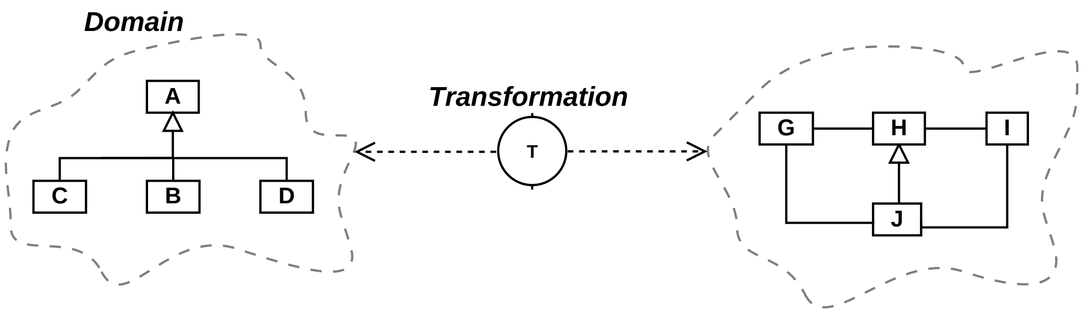


Figure 1.3.: A high level transformation

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 (in our proposal, domains can be either individual model elements or sets of model elements). Regardless of the precise definition of domain chosen, the important fact at this high level is that transformations involve an unspecified number of domains. Importantly we also do not make any assumptions about the 'heart' of the transformation itself: at this stage we have no notions of directionality or executeability.

With this abstract concept of what a transformation is in mind, the following chapter fleshes out our proposal for transformations.

Chapter 2.

The Q^{V}T-*Partners* proposal

In 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, and technically sound, definition of transformations will be the main enabling factor in the realization of the MDA vision, and that such a definition of transformations presents a considerably greater challenge than similar definitions of queries and views.

2.1. 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 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.

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

2.3. 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 2.1 shows the most important parts of concrete syntax we use in relation to

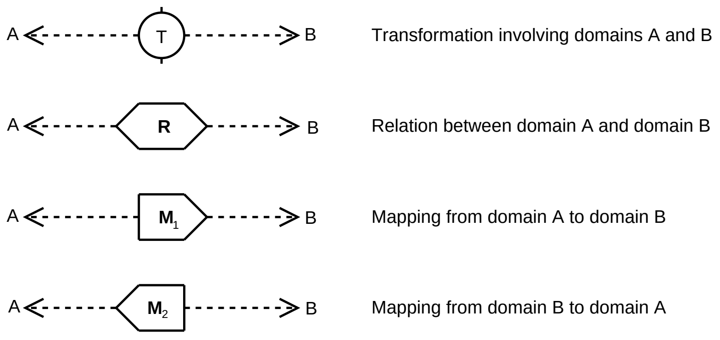

A
T
B
Transformation involving domains A and B
A
R
B
Relation between domain A and domain B
A
M1
B
Mapping from domain A to domain B
A
M2
B
Mapping from domain B to domain A

Figure 2.1.: Concrete syntax

transformations. Note that although figure 2.1 shows transformations, relations and mappings between two domains, our definitions allow each of these to be, in general, between an arbitrary number of domains.

2.4. Relations and mappings

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*[1].

Relations are multi-directional transformation specifications. Relations are not executable in the sense that they are unable to create or alter a model: they can however check two or more models for consistency against one another.

Typically relations are used in the specification stages of system development, or for checking the validity of a mapping.

Mappings are transformation implementations. Unlike relations, mappings are potentially uni-directional and can return values.

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

Figure 2.2 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 2.3 shows how transformations, relations and mappings relate to one another. 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 identification and separation of transformation specifications and implementations, as relations and mappings respectively, gives several benefits to the modeller. These include giving the modeller the ability to:

[1]Note that the terms relation and mapping are not used in their mathematical sense.

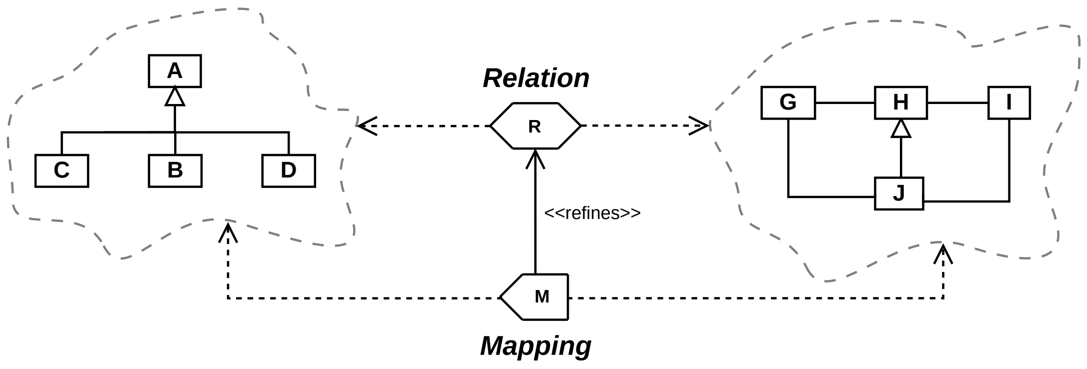


Figure 2.2.: A high level relation being refined by a directed mapping

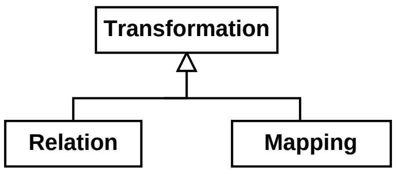


Figure 2.3.: Transformations, relations and mappings

- use relations in the initial stages of system development when implementation details may not have been fully decided upon. This allows transformations to be concretely discussed without an over-commitment to implementation details.
- write multiple mappings, possibly in completely different programming languages, which refine a single relation.
- construct a relation from a pre-existing mapping to tease out details which would otherwise be lost in the mélange.

2.5. Specification and implementation languages

The identification of relations and mappings in section 2.4 is deliberately noncommittal in regards to the languages to be used. This is partly because the concepts of specification and implementations are initially best thought about in an abstract manner, before the discussion of specific languages muddies the waters. However, there is another reason: whilst our submission provides a standard language for expressing relations and mappings (section 2.6), we also allow pre-existing transformations written in arbitrary systems and integrated into models (section 2.7).

2.6. Standard relations and mappings language

Our submission provides a standard language for relations and mappings – which we abbreviate to MTL (Model Transformation Language) in the following sections. MTL utilizes pattern matching as one of the key factors in allowing powerful transformations to be created. Providing a single language for relations and mappings is advantageous for several reasons, but most noticeably for lowering the entry barrier to transformation use. MTL comes in both graphical and textual forms, the graphical form being most useful (as with most diagrams) when used as an abstraction of the textual form.

2.6.1. Pattern matching and domains

In order to facilitate transformations we define powerful a pattern matching language which is utilized by MTL. Pattern matching is a proven concept within transformation systems such as XSLT [W3C99] and textual regular expressions à la Perl. The essential idea behind pattern matching is to allow the succinct expression of complex constraints on an input data type; data which matches the pattern is then picked out and returned to the invoker. MTL allows model fragments to be matched against meta-model patterns and used in transformations.

Pattern languages are inherently a compromise between expressivity on the one hand, and brevity and legibility on the other. As is the case with most pattern languages, the pattern language we propose is not always the best way of expressing aspects of a particular transformation. To that end, domains in our standard relations and mappings languages are comprised of patterns and conditions. By utilizing conditions, arbitrarily complicated expressions can be specified to augment patterns. Furthermore, a separate condition is scoped over all domains which allows domains to be related to one another in a natural way.

The general form of a relation when written in textual form is thus:

```
relation R {
    domain { pattern1 when condition1 }
    ...
    domain { patternn when conditionn }
    when { condition } }
```

The final `condition` is effectively a global condition which scopes over all domains. A relation is called with a number of arguments corresponding to its number of domains; each argument is either a single element, or it is a *choice*. A choice is a data set which contains zero or more objects, each of which may be tested to see if some combination of candidates satisifies the transformation – choices come in both unordered and prioritised forms. The relation is only satisfied when all domains are satisfied. This is more complex than it may initially seem as there can be constraints which hold across more than one domain. Thus an arbitrary object `o` may satisfy one domain but cause another to fail; the semantics of MTL ensure that all domains must be satisfied, which means that a runtime engine may have to try different combinations to ensure the relation is satisfied.

A specific example involving patterns and conditions is the following:

```
relation IncreasingWisdom {
    domain { (Person)[name = n, age = a, wisdom = w1] when a + 1 < 13 or
                                                           a + 1 > 19 }
    domain { (Person)[name = n, age = a + 1, wisdom = w2] }
    when { w2 > w1 }
}
```

Intuitively, this example checks that a birthday brings with it increased wisdom, except during the teenaged years when this is not always the case. `(Person)[name = n, age = a, wisdom = w1] when a > 18` is an example of an object pattern, which take the general form of:

`(Class, self)[label`$_1$` = expr`$_1$`, ... , label`$_n$` = expr`$_n$`]`

`Class` is the class the object must be an instance of, `self` is a variable which is used to refer to the object and the object pattern consists of zero or more entries. An object pattern will match successfully against any object which is an instance of `Class` and whose fields all match successfully against the object patterns fields – note that the object pattern can specify a subset of the field which `Class` defines, although it cannot define more. The pattern language contains many other constructs, such as set and sequence patterns, and patterns can nest within patterns giving huge flexibility. See section 5.6 for more details.

Note how in the `IncreasingWisdom` relation the variables `n` and `a` occur in both domains: the semantics of our proposal ensure that unless marked as local, variables with the same name in different domains of a relation must have the same value.

Figure 2.4 shows `IncreasingWisdom` in diagrammatic form. Note that this is neither a class diagram or an object diagram – it is a transformation diagram, and is most easily thought of as something in between the two.

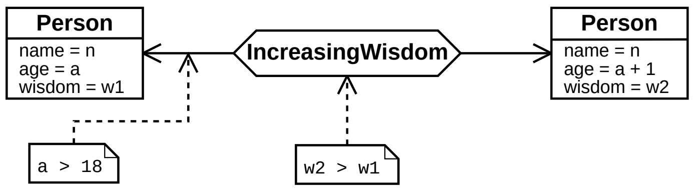


Figure 2.4.: Relation `IncreasingWisdom` in diagrammatic form

2.6.2. Differences between relations and mappings

As far as possible, the standard languages for relations and mappings share the same syntax and semantics. But by virtue of the fact that they are different concepts there are differences between the two. The most obvious difference is that whilst a relation simply consists of a number of domains and an overall constraint, mappings also have an ‘action body’.

A practical example is a mapping `IncreasingWisdomMapping` which is a refinement of the relation `IncreasingWisdom` given in section 2.6.1:

```
mapping IncreasingWisdomMapping refines IncreasingWisdom {
    domain { (Person)[name = n, age = a, wisdom = w1] when a + 1 < 13 and
                                                           a + 1 > 19 }
    body {
        p = new Person()
        p.name = n
        p.age = a + 1
        p.wisdom = w1 + 5
    }
}
```

Note how the only one of the two domains of `IncreasingWisdomMapping` has survived into the mapping: the second domain is effectively replaced by the body of the mapping. Note that the rather verbose body of `IncreasingWisdomMapping` can be rewritten using the ‘create/update object’ notation, which syntactically is intentionally identical to that used in patterns:

```
body {
    new (Person)[p.name = n, p.age = a + 1, p.wisdom = w1 + 5]
}
```

2.7. Integrating pre-existing transformations

As detailed in section 2.6, our submission provides standard languages for relations and mappings; however, we also propose a method for integrating transformations that are written in non-standard ways into models. This allows legacy transformations to be integrated into a model, as well as allowing the integration of new transformations which, for whatever reason, are not written in the standard languages.

2.8. Reusing transformations

The two main ways in which transformations can be reused are through the specialization mechanism and through transformation composition. Transformation specialization works in a similar manner to specialization of other model elements. Transformation composition however is quite different.

2.8.1. Transformation composition

Our submission provides various transformation composition operators which allow complex transformations to be built up from smaller transformations. Composition operators come in both unary and binary flavours and include operators such as `and`, `not` and `or`. To give a simple example, intuitively an `and` composite relation requires that for the composition to hold between given elements, all of the component transformations must hold as well. Composite relations often impose restrictions on the domains of its component relations.

Transformation composition has a effect on backtracking. If part of a composition fails, then our semantics force backtracking to occur. This is because at various points in a transformation, choices can be made to decide on the path forward, but any particular choice may result in the failure of a subsequent transformation. If this happens, the transformation backtracks to the last *choice point*, resets the system to its previous state and chooses a different value than was previously chosen, and then continues execution. Backtracking can occur at an arbitrary number of levels and is a powerful technique for allowing a transformation to find a successful path with minimal user intervention.

Composition is particularly useful with relations because due to their side-effect free nature, backtracking can occur at any level within the transformation with no difficulties. Backtracking with mappings is rather harder, since at certain points the side-effect body of a mapping will be executed, which creates a point at which backtracking can no longer occur. However, even the relatively limited form of backtracking available with mappings allows powerful transformations to be easily expressed.

2.8.2. Sub-transformations

Another form of transformation reuse is much simpler – when one transformation calls another. We call this the sub-transformation relationship, and it typically occurs in `when` conditions. For example a relation `R` may only be satisfied when another relation S is satisfied on sub-data that is matched within R.

Sub-transformations are a powerful mechanism that allow transformations to be reused in arbitrary ways.

2.9. A layered approach to the definition of transformations

Our definition of transformations comes in two distinct layers. Reusing terminology familiar from the UML 2 process, we name these layers *infrastructure* and *superstructure*.

We define a simple infrastructure which contains a small extension to the MOF meta-model with semantics defined in terms of existing OMG standards. The infrastructure also contains a formal language used to precisely define transformations. The infrastructure contains what we consider to be a sensible minimum of machinery necessary to support all relevant types of transformations. The infrastructure is necessarily low-level and not of particular importance to end users of transformations. Its use is as simple semantic core; its presence is also useful for tool vendors and other implementors of the standard, who require a precise reference point.

Secondly we present a superstructure which contains a much higher-level set of language constructs, which are 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. The superstructure we present 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 the superstructure, or variants thereon.

Figure 2.5 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. Information that is encoded in Q includes such things as typing and structural information.

2.10. A proposed methodology

Although our proposal lends itself to being utilised in many different ways, we propose a standard methodology which allows users to quickly and easily get the most out of our proposal. Figure 2.6 shows the suggested sequence of steps used to create an executable transformation. The essential idea is as follows:

1. Gather requirements.
2. Create relations / specifications.
3. Refine each relation into one or more mappings.

This methodolgy makes use of the fact that relations can be viewed as being test cases – a test case being a specification for a correct run through of an implementation. This becomes particularly apparent when relations

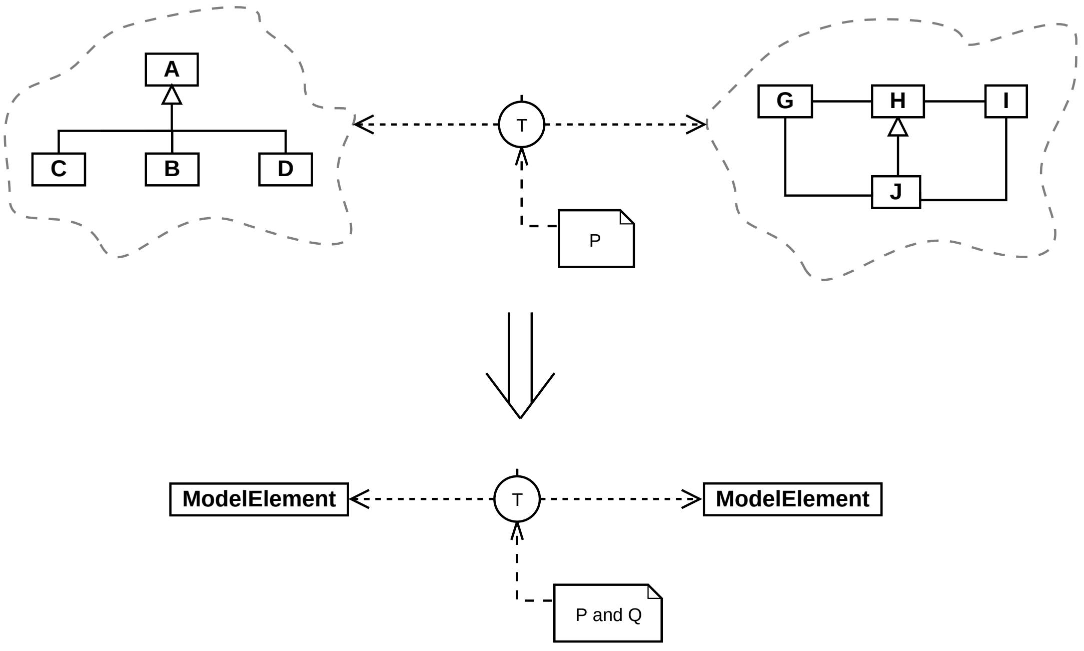


Figure 2.5.: Translating superstructure (top) to infrastructure (bottom)

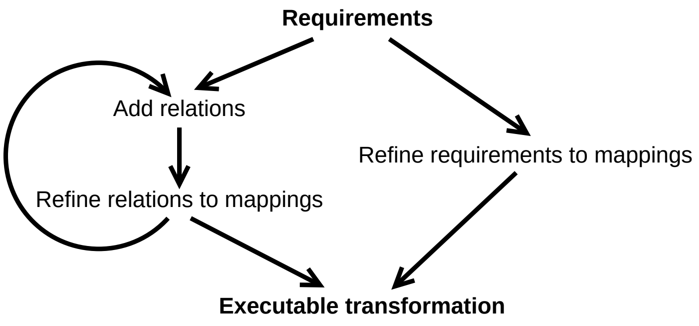

Requirements
Add relations
Refine requirements to mappings
Refine relations to mappings
Executable transformation

Figure 2.6.: Proposed methodology

contain ‘hard data’ in them. Thus in our proposed methodolgy, the development of transformations is test-case driven.

2.11. A simple example

Figure 2.7 shows a simple example of a transformation `AtoX` which transforms a UML-like attribute into an XML element. The model in figure 2.7 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.

At this point, note that we have not specified whether the transformation is a relation or a mapping – in fact, it could be either of these. If the transformation is written in the standard MTL language, then a further transformation diagram can be constructed, as illustrated in figure 2.8, where the domains are specified in more

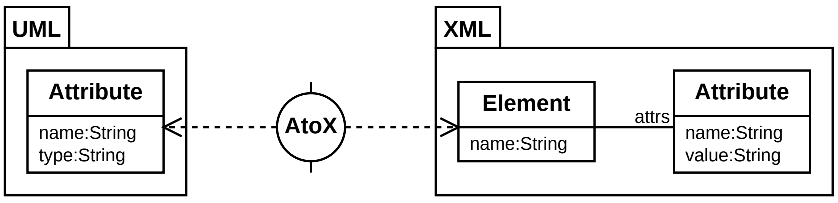


Figure 2.7.: Transforming a UML-like attribute into an XML element

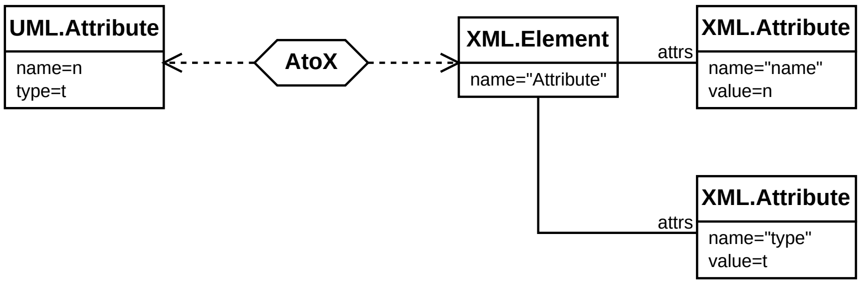


Figure 2.8.: A transformation diagram showing a relation for figure 2.7

detail.

Figure 2.9 shows an instance of the `AtoX` relation. an example where `AtoX` is a relation. When written out in its more familiar textual concrete syntax, the XML in figure 2.9 looks as follows:

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

The textual version of figure 2.8 looks as follows:

```
relation AtoX {
    domain { (UML.Attribute)[name = n, type = t] }
    domain {
        (XML.Element)[
            name = "Attribute",
            attrs = {
                (XML.Attribute)[name = "name", value = n],
```

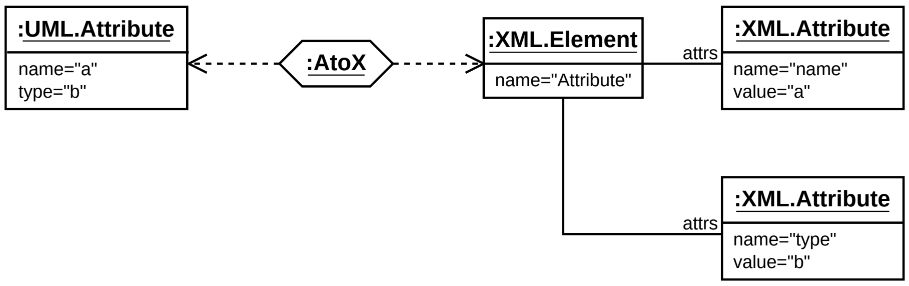


Figure 2.9.: An instance of figure 2.8

```
                (XML.Attribute)[name = "type", value = t]
            }
        ]
    }
}
```

We now present an example of a mapping which refines the relation:

```
mapping MAtoX refines AtoX {
    domain { (UML.Attribute)[name = n, type = t] }
    body {
        (XML.Element)[
            name = "Attribute",
            attrs = {
                (XML.Attribute)[name = "name", value = n],
                (XML.Attribute)[name = "type", value = t]
            }
        ]
    }
}
```

As with most mappings which refine simple relations, the mapping initially looks rather similar to the relation it refines. However, there is a fundamental difference between the relation `AtoX` and the mapping `MAtoX` that can most easily be explained by explaining the difference between the second domain of `AtoX` and the body of `MAtoX`. Syntactically the two appear to share identical contents; in reality, the second domain of `AtoX` contains a pattern and the body of `MAtoX` contains an object expression. The pattern checks that an object matches against it; an object expression creates an object. The two intentionally share the same syntax to reduce the learning curve – as they can not exist in the same context, there is never ambiguity as to which is a pattern and which is an object expression.

For more detailed examples using our proposal, please see appendix A.

2.11.1. Summary

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

- Transformation is a super type of both relation and mapping.
- Relations are fully declarative specifications. Mappings are operational implementations.
- A standard language MTL is provided, which is capable of expressing relations and mappings side by side.
- MTL utilizes pattern matching to allow the succinct expression of many transformations.
- Transformations can be composed.
- Transformations can be specialized.

- Transformations can be arbitrarily used by other transformations.
- Our definition is split into a small core infrastructure, and a richer superstructure. Superstructure is translated into infrastructure.

Chapter 3.

Resolution of RFP requirements

3.1. 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 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 infrastructure and superstructure definitions are capable of expressing all the necessary information for transformations.

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

In our proposal, 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.*

In our proposal, relations are fully declarative.

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

3.2. 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. Relations are not executable in the sense that they can not create or modify a model. Rather they execute in the sense of checking models for correctness with respect to a transformation. As relations are fully declarative, they place no restriction on which direction transformations go between.

Mappings are directed transformations, and 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, mappings can be marked as being traceable.

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 proposal. Transformations may specialize one another. Transformation composition allows large transformations to be created from smaller ones. 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 allows mappings to be marked as being transactional.

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 proposal allows additional data to be fed into the transformation process.

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 proposal does not mandate whether transformations are update-in-place or functional copy, and thus transformations can perform update-in-place if the implementor chooses.

3.3. 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, the superstructure reuses 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.

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 proposal, mappings can refine relations. Relations can thus act as specifications for mappings and there is no need for explicit pre- and post-conditions.

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

Our proposal is amenable to use by various synchronisation schemes: we do not mandate any particular such scheme.

3.4. 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 3.3.

Action Semantics Language (ASL) See section 3.3.

Part II.

TECHNICAL DETAILS

Chapter 4.

Overview

Our definition of transformations is spread over the following three chapters:

5. Superstructure The superstructure is the semantically and syntactically rich part of the proposal. It is intended for end users.

This chapter defines textual and visual concrete syntaxes which are an extension to MOF. The superstructure is the compliance point for the submission.

6. Infrastructure The infrastructure is the small semantic core of our proposal. It is not intended for end users. It is useful for tool vendors and others who require a precise reference point.

The infrastructure is not a compliance point for the submission.

7. Superstructure to infrastructure translation The semantics of the superstructure are given by its translation into the infrastructure.

This chapter presents the superstructure to infrastructure translation in terms of infrastructure transformations.

Chapter 5.

Superstructure

5.1. Overview

This chapter defines the superstructure portion of our proposal. From an end users point of view, the superstructure *is* the proposal – users write their transformations in the superstructure language, and exchange transformations with one another in the superstructure form. The semantics of the superstructure is given via a translation into the infrastructure, which is presented in section 7.

This chapter assumes a familiarity with the introduction to this document (chapter 2) that should give the reader a broad overview of our proposal that is then expanded upon in this chapter.

5.2. Initial superstructure example

In order to provide a suitable introduction to the superstructure definition, we take a well understood example, the translation from UML to XML and back again [ACR$^+$03], and use it as the basis for this section. Figures 5.1 and 5.2 show the simplified meta-models we use for UML and XML respectively in the rest of this section.

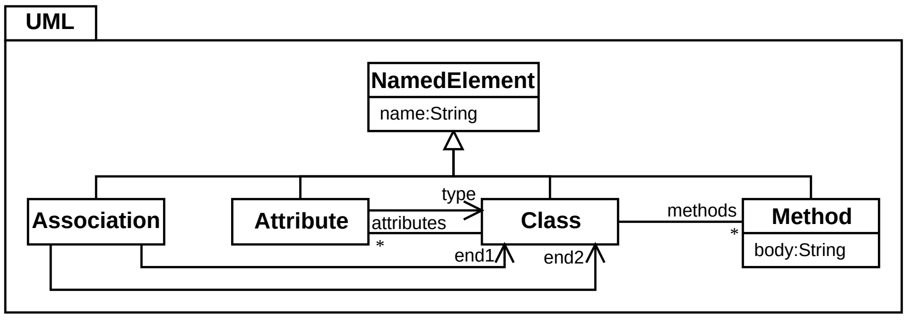


Figure 5.1.: A simplified model of UML

5.2.1. Specifying a relation

The main way of writing a transformation definition is via the superstructure textual syntax (which is defined in section 5.6). In the first part of this example, we will be chiefly concentrating on relations. The general form of a relation is as follows:

```
relation R {
  domain { pattern₁ when condition₁ }
```

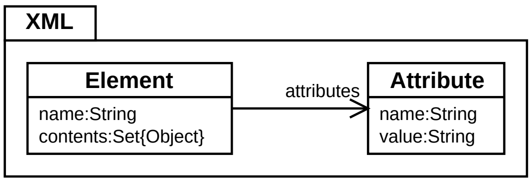


Figure 5.2.: A simplified model of XML

```
  ...
  domain { pattern_n when condition_n }
  when { condition }
}
```

There is also a visual syntax for transformations in general, and relations in particular, which is intended to be used – as are most visual notations – as an abstraction of the textual syntax.

A relation is effectively a constraint over multiple domains. Domains can be thought of as ordered inputs to a transformation, where each input has a ‘type’ given by one or more patterns, and a condition, within the domain. There is then an overarching constraint which can constrain one or more domains in parallel. A relation can return `true` or `false` according to whether a suitable combination of the input values can be found that satisfies all domains, and the overarching constraint.

As a first example, we now present a relation which relates UML methods to XML elements. Note that with relations there is no notion of directionality – that is, the order of the domains carries with it no implication with it that we wish to change a UML model into XML or vice versa.

```
relation Method_And_XML {
  domain { (UML.Method)[name = n, body = b] }
  domain {
    (XML.Element)[
      name = "Method",
      attributes = {(XML.Attribute)[name = "name", value = n]},
      contents = {b}
    ]
  }
}
```

5.2.2. Patterns

The relation `Method_And_XML` contains two domains; each domain contains a single pattern. Patterns are a means of succinctly describing the ‘shape’ of model elements, without necessarily describing every aspect of the model elements in question. Patterns attempt to match against candidate objects – if the candidate object fulfils all of the patterns constraints, then the match is successful.

For example, the pattern in the first domain of `Method_And_XML` is an object pattern which will only match against an instance of the `UML.Method` class or instances of `UML.Method`’s subclasses. An object pattern specifies zero or more fields, each with its own pattern, to match against. Only the type of the object pattern, and the fields specified in the object pattern are of interest – any other fields a potential object might possess are of no importance. So if the pattern in the first domain `Method_And_XML` was faced with an object which had more fields than the object pattern specifies (in our example, this would mean that the object in question was an instance of a subclass of `UML.Method` which defined new fields), then provided it successfully matches against the fields `name` and `body`, the overall match is successful.

Literals such as strings are also valid patterns: they will only match against themselves. For example a pattern which is the string literal `"s"` will only match against another string literal `"s"`; the number `42` will only match against a number `42`.

As can be seen for the patterns specified for the fields `name` and `body` in `Method_And_XML`, a valid pattern is a single variable. Such a variable matches against anything, and the element it matches against it is assigned to that variable; subsequent occurrences of the same variable name must contain the same value in order to constitute a valid match. In our example, assuming that a UML method with the name `"m"` is matched against the first domain, then the string value `"m"` will be assigned to the variable `n`. The occurrence of `n` in the second domain will ensure that only an `XML.Attribute` instance whose value is the string `"m"` will succeed.

Bringing all this information about patterns together, the following pattern would only match successfully against a `Class` instance whose name is the string literal `"C"`:

```
(Class)[name = "C"]
```

The following pattern will match successfully against any `Class` instance, assigning whatever name it has to the variable `n`:

```
(Class)[name = n]
```

Sets are denoted between curly brackets { and }. The following pattern will match successfully against any `Class` instance which has one attribute whose name is `"a"`:

```
(Class)[attributes = {(Attribute)[name = "a"]}]
```

The following pattern will match successfully against any `Class` instance which has one attribute; the name of the class will be assigned to the variable `n` and the name of the attribute will be assigned to `m`:

```
(Class)[name = n, attributes = {(Attribute)[name = m]}]
```

These patterns are sufficient to guide us through our example; some more complex patterns are explored in section 5.3.

5.2.3. Invocation

Given the above definition of `Method_And_XML`, it is instructive to see various invocations which will succeed or fail accordingly. In the following invocations, note that what may look like an object pattern is in fact an object expression. Object expressions denote real objects – for example the object expression `(Class)[name = "C"]` is a particular instance of `Class` which has a name `C`. Since patterns can only occur within domains, the context always makes clear what is an object pattern and what is an object expression.

```
// This invocation succeeds

Method_And_XML(
  (UML.Method)[name = "m", body = ""],
  (XML.Element)[name = "Method", contents = {}, attributes =
    {(XML.Attribute)[name = "name", value = "m"]}
  ])

// This invocation fails because the two values of n in the object
// pattern do not contain the same value -- in the first domain n is
// bound to "m", whilst in the second it is bound to "f".

Method_And_XML(
  (UML.Method)[name = "m", body = ""],
  (XML.Element)[name = "Method", contents = {}, attributes =
    {(XML.Attribute)[name = "name", value = "f"]}
  ])
```

5.2.4. Choices

The ability to check specific objects for correctness with regard to a relation is useful, but often one needs to know if there is a combination of objects which are correct with respect to each other. This can be easily done by passing *choices* to a relations domain. A choice is a container of objects – the relation will attempt to find a combination of objects from different domains which satisfies the relation. Choices can be manually built by surrounding lists of objects with < and >. For example, consider the following invocation of `Method_And_XML`:

```
  Method_And_XML(
    (UML.Method)[name = "m", body = ""],
    <(XML.Element)[name = "Method", contents = {}, attributes =
        {(XML.Attribute)[name = "name", value = "f"]},
      (XML.Element)[name = "Method", contents = {}, attributes =
        {(XML.Attribute)[name = "name", value = "m"]}>
    ])
```

This invocation will succeed because although a single object is supplied for the first domain of `Method_And_XML`, two objects are supplied as choices for the second domain. The semantics of our proposal mean that all potential combinations of choices are tried until a successful one is found. In our above example, there are only two possible permutations – with larger choices, which happens naturally if the arguments to more than one domain are given as choices, the number of permutations can be much larger. It is worth noting that in practise the full number of permutations will rarely need to be tried, even if there is not a match within them, as often certain objects will fail to match anything, and thus strike out a large number of possible permutations.

All elements have a `tochoice` operation which returns the element as a choice. For elements which are a single data value, this is equivalent to wrapping the element with < and >. For sets and sequences, `tochoice` returns a choice which contains all the member elements of the set or sequence.

5.2.5. Visualizing a relation

At this point it is instructive to see a visual counterpart of the relation we have been defining up to this point. Figure 5.3 shows a transformation diagram for `Method_And_XML`. A transformation diagram is neither an object diagram or a class diagram – it is something in between, although closer to an object diagram. In this instance, as our original relation is fairly simple, we are able to display all of the relevant material in the diagram without cluttering it – larger examples will almost certainly require the eliding of detail in the diagram to keep it understandable.

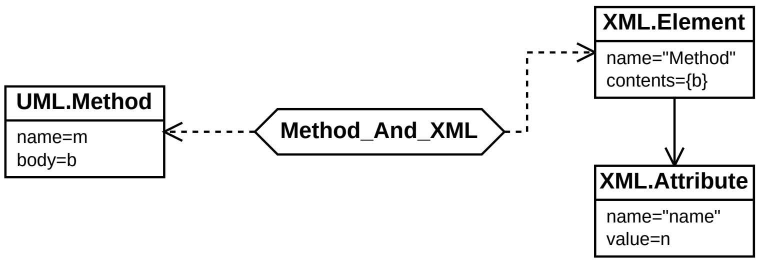


Figure 5.3.: `Method_And_XML` represented visually

5.2.6. The missing pieces

The 'complete' relation for relating UML and XML is actually a composite transformation. Before we are able to define that relation, we need to specify the rest of the relations necessary to relate the individual components of UML and XML:

```
relation Association_And_XML {
  domain { (UML.Association)[name = n, end1 = e1, end2 = e2] }
  domain {
    (XML.Element)[
      name = "Association",
      attributes = {
        (XML.Attribute)[name = "name", value = n],
        (XML.Attribute)[name = "end1", value = e1.name],
        (XML.Attribute)[name = "end2", value = e2.name]
      }
    ]
  }
}

relation Attribute_And_XML {
  domain { (UML.Attribute)[name = n, type = t] }
  domain {
    (XML.Element)[
      name = "Attribute",
      attributes = {
        (XML.Attribute)[name = "name", value = n],
        (XML.Attribute)[name = "type", value = t.name]}
      }
    ]
  }
}
```

The relation for `Class_And_XML` is more complex than those that have preceded it:

```
relation Class_And_XML {
  domain { (UML.Class)[name = n, attributes = A, methods = M] }
  domain {
    (XML.Element)[
      name = "Class",
      attributes = {
        (XML.Attribute)[name = "name", value = n],
      },
      contents = AM
    ]
  }
  when {
    A->forAll(a |
      Attribute_And_XML(a, AM.tochoice())) and
    M->forAll(M |
      Method_And_XML(m, AM.tochoice()))
  }
}
```

Whereas the other relations merely need to ensure that they correctly relate a single UML element to XML, `Class_And_XML` needs to ensure that a UML class's methods and attributes are also correctly related. The way the relation does this is to first of all bundle up the UML's attributes and methods into variables `A` and `M` respectively[1]; the contents of the XML class element are then bound to `AM`. In the overarching `when` clause for the entire relation, the relation then ensures that every attribute in the set `A` has a corresponding piece of XML in `AM` by calling the `Attribute_And_XML` with an attribute as the argument for the first domain, and `AM` converted to a choice for the second domain. `Attribute_And_XML` will return true if given a particular attribute, there is an XML element in `AM` which matches it; otherwise `Attribute_And_XML` will return false.

5.2.7. A composite relation

Having defined all the individual relations necessary, we can now create a composite relation which can check a set of UML model element instances for conformance with a set of XML model element instances. Note that in the interests of making some useful points that would otherwise have been lost, we have chosen not to have top-level elements for either UML or XML, although this would generally be the case in practise.

The relation `UML_and_XML` is defined thus:

```
relation UML_and_XML {
  domain { U }
  domain { X }
  when {
    U->forAll(u |
      disjunct(
        Association_And_XML(u, X.tochoice()),
        Class_And_XML(u, X.tochoice()))) and
```

[1]The observant reader may have noticed that we use lower-case letters to indicate variables which will contain a single value, and upper-case letters for those that will hold a set. This is purely a matter of convention, and is intended to aid readability.

```
      X->forAll(x |
        disjunct(
          Association_And_XML(U.tochoice(), x),
          Class_And_XML(U.tochoice(), x)))
    }
  }
```

Because a single top-level element is not passed to either domain of `UML_and_XML`, the two domains both expect to receive a set of elements – as attributes and methods can only exist within classes, we assume that the sets of elements that are passed in denote only classes and associations.

Composition in the form of disjunction is first used to ensure that for every UML element `u`, there is a corresponding relation with XML element(s) in `X`. Disjunction effectively means ‘or’. Therefore, what is being said in the `disjunct` application is essentially ‘do either of the relations `Association_And_XML` or `Class_And_XML` return true?’ The first one of the relations to return true leads to the immediate success of the `disjunct`; if neither are true, `disjunct` returns false.

Because ensuring that every UML element has a relation with an XML element(s) does not ensure that the converse relationship is true – for example there may be extra items in the set of XML elements that have no relation to any UML element – in the interests of completeness the check is then reversed and for every XML element `x`, the relation checks to see if there is a corresponding relation with UML element(s) in `U`.

5.3. Building a more complex relation

With the exception of `Class_And_XML`, the relations in the previous section were relatively simple. In this section, we give an example of a more complex relation which could be used as a drop in replacement for `Association_And_XML`. The idea behind this new relation is to flatten UML associations, so that they are related to attributes in XML. This has the side effect that although a model with UML associations in it can be checked for conformance against its flattened variant in XML, the flattened XML variant may also check true for a UML version where all associations have been replaced by attributes.

We define the relation `Association_And_Attributes_XML` as follows:

```
relation Association_And_Attributes_XML {
  domain { (Association)[end1 = e1, end2 = e2] }
  domain {
    (XML.Element, c1)[
      contents = {
        (XML.Element)[
          name = "Attribute",
          attributes = {(XML.Attribute)[name = "ref", c2.name]}
        ],
        |
        _
      }
    ]
    (XML.Element, c2)[
      contents = {
        (XML.Element)[
          name = "Attribute",
          attributes = {(XML.Attribute)[name = "ref", c1.name]}
        ]
        |
        _
      }
    ]
  }
  when {
    Class_And_XML(e1, c1) and Class_And_XML(e2, c2)
  }
}
```

This relation uses three major concepts which have not featured hitherto. Firstly, in the second domain the two object patterns have *self* variables `c1` and `c2`, meaning that one can refer to the entire object which matches the pattern via these variables. Secondly, both object expressions use sophisticated set patterns to match against the XML elements content's. Thirdly, the 'catch all' pseudo-variable `_` is used in the set pattern.

We will deal with these new concepts in reverse order.

5.3.1. The _ pseudo-variable

The _ pseudo-variable matches against anything – as does any variable – but unlike normal variables silently discards the result. There is no requirement that different instances of the _ pseudo-variable refer to the same value. So in the second domain of `Association_And_Attributes_XML` the two instances of _ may validly match against completely different values.

5.3.2. Set patterns

Set patterns take the general form of:

$\{v_1, \ldots, v_2 \mid S_1, \ldots, S_m\}$

Set patterns are similar to set comprehensions in some programming language. Values to the left of the vertical bar are matched as is; values to the right hand side of the bar must be sets, and are (possibly empty) subsets of the overall set. For example the following set pattern will match against a set which contains a string literal `"m"`, with any other values being assigned to the variable `S`:

```
{"m" | S}
```

If the set `"m", "a", "b"` were to be matched against the preceding pattern, there is only a single outcome possible: the match would succeed and `S` would be assigned a new set `"a", "b"`.

With a set pattern such as the following one:

```
{v | S}
```

one gets non-determinism. Given the set {`"m", "a", "b"`} to match against, the possible outcomes are as follows:

v	S
"m"	{"a", "b"}
"a"	{"m", "b"}
"b"	{"m", "a"}

With a set pattern such as:

```
{v | S, T}
```

and the set {"m", "a", "b"} to match against, all of the possible outcomes share the property that $\{v\} \cup S \cup T =$ {"m", "a", "b"} and include the following:

v	S	T
"m"	{"a", "b"}	{}
"m"	{"a"}	{"b"}
"m"	{"b"}	{"a"}
"m"	{}	{"a", "b"}
...	...	...

In the relation `Association_And_Attributes_XML` the second domain contains two of the more general form of set patterns in the `contents` field of an XML element:

```
contents = {
  (XML.Element)[
    name = "Attribute",
    attributes = {(XML.Attribute)[name = "ref", c2.name]}
  ],
  |
  _
}
```

This pattern will match successfully against a set that contains an XML element instance which matches the object pattern to the left of the `|`; the `_` pseudo-variable to the right of the `|` means that it will match against a set of any size provided that the left hand side of the set pattern is satisfied, since any extra elements will be assigned as a new set to `_` and then silently discarded. Because the value to be matched against `v` is chosen non-deterministically, there is no way of telling what path a match will take when set selection is involved. This means that relations which involve set patterns can potentially reach a verdict in a different way each time they run, although given the same data and same relation the same answer will always be reached.

5.3.3. Self variables

In the relation `Association_And_Attributes_XML` the second domain contains two instances of the following object pattern:

```
(XML.Element, s)[ ... ]
```

The second element in the 'prologue' to the object pattern is a variable which refers to the whole of the object against which the object pattern matches. In other words the variable `s` is assigned a reference to the object itself.

In Association_And_Attributes_XML, the two object expressions both have self variables, and each object expression refers to one another in order that it can determine the appropriate name of the type it refers to[2].

5.4. Creating a mapping

Whereas a relation is a specification of a transformation which can check two models for conformance with each other with respect to the relation, a mapping is an implementation that can be run on an input data model to produce an output data model. In our UML and XML example, this means that if one wished to actually transform a UML model to XML and back again, one would need a single relation to specify the transformation, but two mappings to implement it.

Mappings take the general form of:

```
mapping M {
  domain { pattern_1 when condition_1 }
  ...
  domain { pattern_n when condition_n }
  when { condition }
  body { expression }
}
```

Although a mapping may have a number of domains, as do relations, there is a fundamental difference to the way multiple domains are used with mappings and relations. With mappings, all the domains are effectively input arguments to the mapping which, providing the input data satisfies the domains, executes the body of the mapping to produce output.

A mapping can refine one or more relation, which essentially means that the mapping must be consistent with the relations: in other words, the relations the mapping refines are effectively pre and post conditions for the mapping. There is no implication that a mapping is the only refinement of a relation – multiple different mappings may refine the same relation.

Mappings which refine trivial relations often look very similar because of the intentionally close syntactic correspondence between object expressions and patterns. For example, a valid refinement of the relation Method_And_XML would be the following:

```
mapping Method_To_XML refines Method_And_XML {
  domain { (UML.Method)[name = n, body = b] }
  body {
    (XML.Element)[
      name = "Method",
      attributes = {(XML.Attribute)[name = "name", value = n]},
      contents = {b}
    ]
  }
}
```

[2]For the sake of completeness, we note the fact that Association_And_Attributes_XML could be written more simply without self variables. It is not true that this is possible in general, and it would lessen the pedagogical uses of the relation.

Despite the syntactic similarity, if the mapping `Method_To_XML` is invoked thus:

```
Method_To_XML((UML.Method)[name = "m", body = ""])
```

then rather than returning boolean `true` or `false` as the relation `Method_And_XML` would, it will return the following object:

```
(XML.Element)[
  name = "Method",
  attributes = {(XML.Attribute)[name = "name", value = "m"]},
  contents = {""}
]
```

5.5. Meta Model

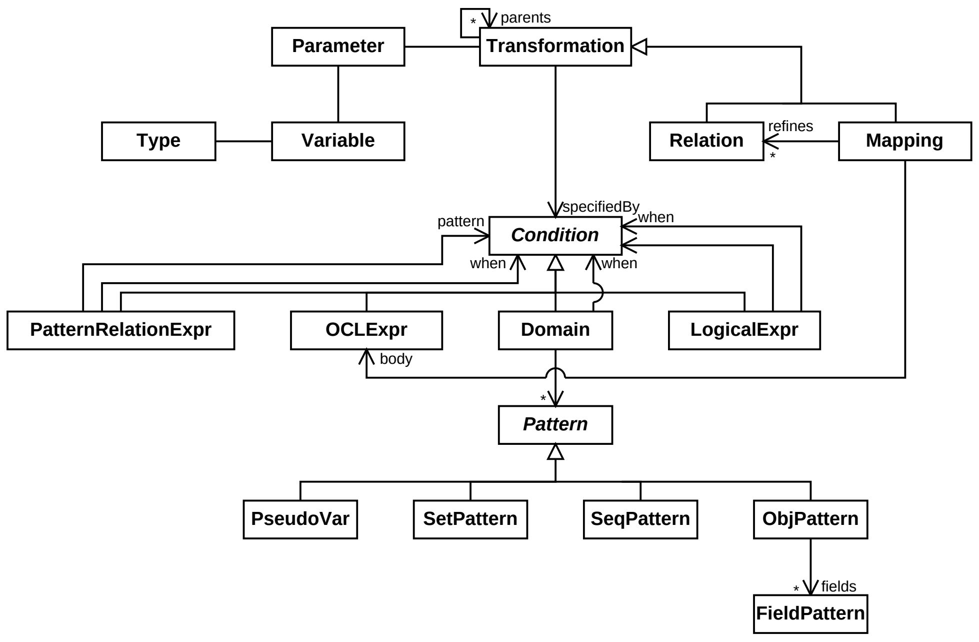


Figure 5.4.: Superstructure meta-model

Figure 5.4 shows the simplified superstructure meta-model which is an extension of MOF. `Transformation` is unsurprisingly the key element in the model. `Relation` and `Mapping` are sub-classes of `Transformation` reflecting the fact that relations and mappings are particular kinds of transformations.

We now detail some of the key model elements in more detail.

5.5.1. `Transformation`

`Transformation` is mostly used as an almost abstract superclass of `Relation` and `Mapping` as in the textual language, one expresses relations and mappings directly – there is no direct way of expressing a transformation. However in the diagrammatic syntax, transformations can be represented directly when one wishes to abstract away from whether a particular transformation is a relation or a mapping in a diagram.

`Transformation` has the following attributes:

`name:String` All transformations have a name.

`parents:Seq{Transformation}` Transformations can specialize zero or more parents.

5.5.2. Relation

`Relation` is the class which represents relations. `Relation` has no attributes.

5.5.3. Mapping

`Mapping` is the class which represents mappings.

`Mapping` has the following attributes:

`refines:Set{Relation}` A mapping can refine zero or more relations, in which case the mapping must be consistent with all of the relations it refines. Essentially this means that the relations are used as pre and post conditions for the mapping.

`traceable:bool=false` If set to true, then extra information is generated to enable debugging. The default is not to generate this information.

`transaction:bool=false` If set to true, then a mapping is considered a transactional unit. This means that when the mapping is completed, a transaction is considered to have been committed, and a commit action is implicitly performed.

5.5.4. Condition

is a boolean expression that may be used to define a `Transformation`.

`Condition` has no attributes.

5.5.5. Domain

`Domain` is the class which represents transformation domains. A domain consists of one or more patterns and a `when` condition.

`Domain` has the following attributes:

`condition:Expr` A domain contains a local condition.

`patterns:Set{Pattern}` A domain contains one or more patterns.

5.5.6. PatternRelationExpr

`PatternRelationExpr` is specified by a set of `Conditions`. Each `Condition` represents a pattern. A `PatternRelationExpression` has an additional `when` condition which may be used to specify the relationship between the different patterns.

`PatternRelationExpr` has the following attributes:

`when:Condition` This specifies the relationship between the different patterns.

5.5.7. `LogicalExpression`

`LogicalExpression` is a boolean expression connecting different conditions using boolean operators `and`, `or` and `not`.

5.5.8. `Pattern`

`Pattern` is the abstract superclass of all top-level patterns. A domain consists of one or more patterns and a `when` condition.

`Pattern` has no attributes.

5.5.9. `PseudoVar`

`PseudoVar` is the class which represents the pseudo-variable `_`. The pseudo-variable `_` matches against any value, and silently discards the value that it is matched against.

`PseudoVar` has no attributes.

5.5.10. `SetPattern`

`SetPattern` is the class which represents a set pattern. A set pattern consists of zero or more patterns which will match against single values, and zero or more patterns which will match against subsets of the set.

`SetPattern` contains the following attributes:

`values:Set{Pattern}` A set pattern contains zero or more patterns which will match against single values.

`subsets:Set{Pattern}` A set pattern contains zero or more patterns which will match against subsets.

5.5.11. `SeqPattern`

`SeqPattern` is the class which represents a sequence pattern. A sequence pattern consists of zero or More patterns which will match against single values, and zero or more patterns which will match against subsequences of the sequence.

`SeqPattern` contains the following attributes:

`values:Seq{Pattern}` A sequence pattern contains zero or more patterns which will match against single values.

`subsets:Seq{Pattern}` A seq pattern contains zero or more patterns which will match against subsequences.

5.5.12. `ObjPattern`

`ObjPattern` is the class which represents an object pattern. An object pattern consists of a class reference, an optional self variable and zero or more fields.

`ObjPattern` contains the following attributes:

`fields:Set{FieldPattern}` An object pattern contains zero or more field patterns.

`of:Pattern` An object pattern contains a class reference to indicate what class the object must be *of* in order to match.

`self:Var=null` An object pattern can optionally contain a self variable which will be assigned a reference of an object if the object matches successfully against the object pattern.

5.5.13. `FieldPattern`

`FieldPattern` is the class which represents a field pattern. A field pattern can only exist as a constituent part of an object pattern. A field pattern consists of a field name and a pattern for the field contents.

`FieldPattern` contains the following attributes:

`contents:Pattern` A field pattern has a pattern for the field contents.

`name:String` A field pattern has a name.

5.6. Textual syntax

The main way that users specify transformations in MTL is via a textual syntax. The syntax is in the C/C++/Java vein of syntax. The grammar is given in section 5.6.1.

5.6.1. Grammar

The BNF grammar for the superstructure syntax is as follows. Atoms may have zero or one suffixes: a ‘*’ indicates that zero or more occurrences of the atom is allowed; a ‘+’ indicates that one or more occurrences of the atom are allowed. Atoms can be surrounded by square brackets ‘[’ and ‘]’ to indicate that zero or one occurrences of the atom are allowed. Atoms can be grouped between ‘(’ and ‘)’.

⟨*toplevel*⟩	::=	⟨*definition*⟩* ⟨*expr*⟩+
⟨*definition*⟩	::=	⟨*relation*⟩
	|	⟨*mapping*⟩
⟨*relation*⟩	::=	‘`relation`’ ⟨*name*⟩ ⟨*extends*⟩* ‘{’ ⟨*domain*⟩+ [⟨*when*⟩] ‘}’
⟨*mapping*⟩	::=	‘`mapping`’ ⟨*name*⟩ ⟨*extends*⟩* ⟨*refines*⟩* [‘`trace`’] [‘`transaction`’] ‘{’ ⟨*domain*⟩+ [⟨*when*⟩] ‘`body`’ ‘{’ ⟨*expr*⟩ ‘}’ ‘}’
⟨*extends*⟩	::=	‘`extends`’ ⟨*name*⟩
⟨*refines*⟩	::=	‘`refines`’ ⟨*name*⟩
⟨*domain*⟩	::=	‘`domain`’ { ⟨*pattern*⟩+ [⟨*when*⟩] }
⟨*when*⟩	::=	‘`when`’ ‘{’ ⟨*expr*⟩+ ‘}’

```
⟨expr⟩           ::= ‘<’ [⟨expr⟩ (‘,’ ⟨expr⟩)*] ‘>’
                   |  ‘ordered’ ‘<’ [⟨expr⟩ (‘,’ ⟨expr⟩)*] ‘>’
                   |  ‘let’ (⟨var⟩ ‘=’ ⟨expr⟩)+ ‘in’ ⟨expr⟩ ‘end’
                   |  ⟨ocl⟩

⟨pattern⟩        ::= ‘_’
                   |  ⟨set_pattern⟩
                   |  ⟨seq_pattern⟩
                   |  ⟨obj_pattern⟩
                   |  ⟨pattern_composition⟩
                   |  ⟨expr⟩

⟨set_pattern⟩    ::= ‘{’ [⟨pattern⟩* (‘|’ ⟨pattern⟩)*] ‘}’

⟨set_pattern⟩    ::= ‘[’ [⟨pattern⟩* (‘|’ ⟨pattern⟩)*] ‘]’

⟨obj_pattern⟩    ::= [ ‘(’ ⟨var⟩ [ ‘,’ ⟨var⟩ ]] ‘[’ [⟨field_pattern⟩ (‘,’ ⟨field_pattern⟩)*] ‘]’

⟨field_pattern⟩  ::= ⟨var⟩ ‘=’ ⟨pattern⟩
```

5.6.2. Expressions in patterns

Pattern can not contain arbitrary expressions, due to the inherent problems they pose. For example, in the face of an expression such as `a + b`, where either or both of `a` or `b` is not bound, an execution engine would potentially attempt to work its way through the infinite possibilities of `a` and `b` that would satisfy the expression, even given a single integer to match against. Such freedom of expression would far too easily lead to transformations which have little practical hope of executing in a finite time.

Via a well-formedness check, we restrict patterns in expressions to be one of two things:

- A single free variable `x`.
- An expression where there are no free variables.

As patterns within domains can bind values in other domains, this means that a large class of very useful expressions within patterns are legal, and also that they are guaranteed to execute in a finite time. For example, although the expression `a + b` is not allowed on its own, in the following context it is legal:

```
relation R {
  domain { (Box)[height = a, width = b] }
  domain { a + b }
}
```

since both `a` and `b` are bound within the first domain.

The well-formedness check takes into account dependencies between domains, so the following is also valid:

```
relation R {
  domain { (Box)[height = a, width = 2 * b] }
  domain { (Kite)[height = 2 * a, length = b] }
}
```

This relation checks that a wide box is related to a tall kite – notice how both domains have one expression with a single free variable, and one expression which contains no free variables. Intuitively, an execution engine can 'pivot' around the values bound by the free variables, and can then execute the expressions which contain the bound variables.

5.6.3. Composing transformations

Complex transformations can be built by composing simpler transformations using composition functions. Several such functions are built in to the superstructure language: `disjunct`, `conjunct` and `not`. Syntactically, composition functions are accessed via standard function application, although semantically they are dealt with specially (see section 7).

5.7. Visual Notation

Diagrammatic notations have been a key factor in the success of UML, allowing users to specify abstractions of underlying systems in a natural and intuitive way. Therefore our proposal contains a diagrammatic syntax to complement the textual syntax of section 5.6.

There are two ways in which the diagrammatic syntax is used, as a way of:

- representing transformations in standard UML class diagrams.
- representing transformations, domains and patterns in a new diagram form: *transformation diagrams*.

The syntax is consistent between its two uses, the first usage representing a subset of the second.

Here we propose a visual notation to specify transformations. A transformation can be viewed as a relationship between two or more patterns. Each pattern is a collection of objects, links and values. The structure of a pattern, as specified by objects and links between them, can be expressed using UML object diagrams. Hence, we propose to use object diagrams with some extensions to specify patterns within a transformation specification.

The notation is introduced through some examples followed by detailed syntax and semantics.

Figure 5.5 specifies a transformation, `UML2Rel` from UML `classes` and `attributes` to relational `tables` and `columns`. The figure specifies that the relation, `UML2Rel` holds between a `class c` and a `table t` if and only if corresponding to every attribute of `class c`, there is a column in `table t`, with the same name. As with associations, cardinalities can be associated with a relationship.

Different relations maybe combined using boolean operators. The above relation is extended to packages in the figure 5.6. The specification states that two packages are related by the relation `UML2Rel1` iff corresponding to every `class c` in the first package, there exists a `table t`, in the second package and the `c` and `t` thus related are also related by the relation `UML2Rel`. That is, if a class c1 is related to table t1 then t1 and c1 have the same name, and corresponding to every attribute of class c1 there exists a column in table t1 with the same name as the corresponding attribute.

The above example showed how different relations can be combined using boolean operators. Similarly two pattern specifications can be combined using a boolean operator. An example is given in figure 5.7. The example specifies a relation between a UML `class c` and `table t`. The relation holds if for every pattern involving the parameter class c that matches the pattern

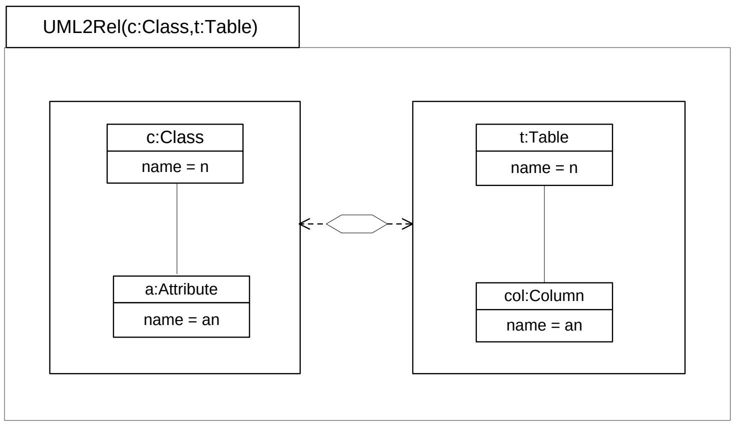


Figure 5.5.: UML Class to Relational Table Transformation

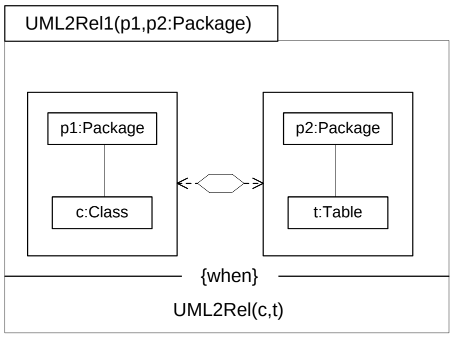


Figure 5.6.: Transformation from UML Package to Relational Package

- `class c` has `attribute a` or
- `class c` is associated to `class c1` which in turn has a primary `attribute a`

then there exists a pattern involving the parameter `table t` and having a `column col` with the same name as the `attribute a`.

In all the examples so far, the patterns comprised individual objects and links between them. The notation also supports specifications involving sets of objects. The `UML Class to Relational Table` example can be specified using sets of objects as shown in figure 5.8. In the figure `aset` and `colset`, specify the set of all attributes of class `c`, and all columns of table `t`, respectively. The specification additionally states that corresponding to every `attribute a`, in `aset`, there exists a `column col`, in `cset` having the same name.

The notation also includes support for specifying the non-existence of objects and `overriding` of relations. Figure 5.9 specifies a strange transformation from a class with no attributes to a table with exactly one column named ’empty’. The `X` annotating `Attribute` indicates that this pattern matches only if there exists no `Attribute` linked to `class c`. The specification overrides `UML2Rel3`. Thus, if the class has any attributes `UML2Rel3` should hold else `UML2Rel4` should hold. Note that a class with no attributes matches `UML2Rel3`

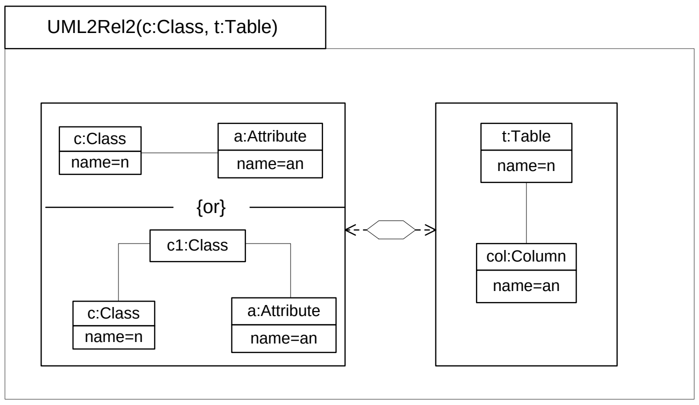


Figure 5.7.: Transformation from UML Classes and Associations to Relational Tables

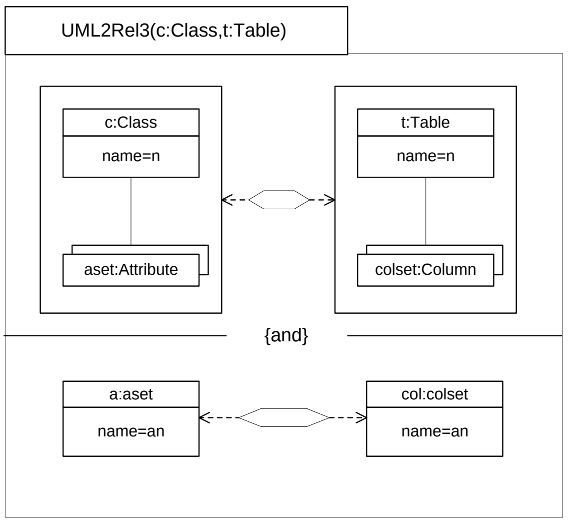


Figure 5.8.: Transformation from UML Classes to Relational Tables using Sets

too.

Two relations can also be combined using the `extends` construct. Unlike `overrides`, extends is like the {`and`} operator. Figure 5.10 gives an example of the extends construct. It extends the specification given in Figure 5.5. Figure 5.5 specifies a transformation from ⟨`class`, `attribute`⟩ to ⟨`table`, `column`⟩. Figure 5.10 extends this to include transformation of `primary key` attributes of associated classes as foreign key columns.

Not all complex conditions can be specified using only visual notations. Transformation's definitions maybe annotated with `OCL` to express these complex conditions. This is shown in Figure 5.11.

5.7.1. Visual Notation Elements

Table 5.12 gives a brief description of the various visual notation elements.

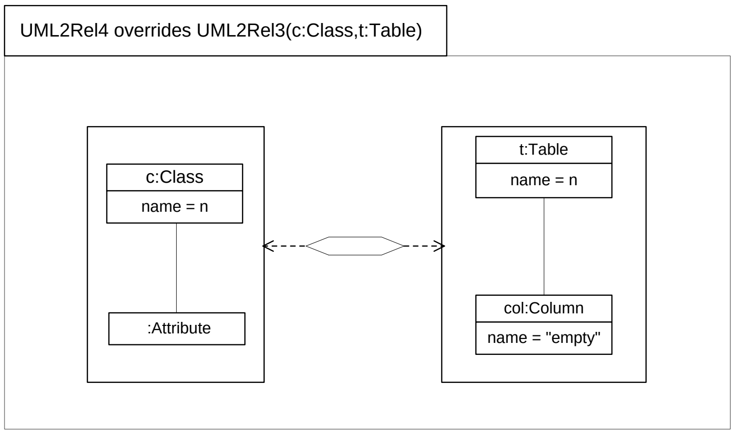


Figure 5.9.: Transformation of Empty Classes

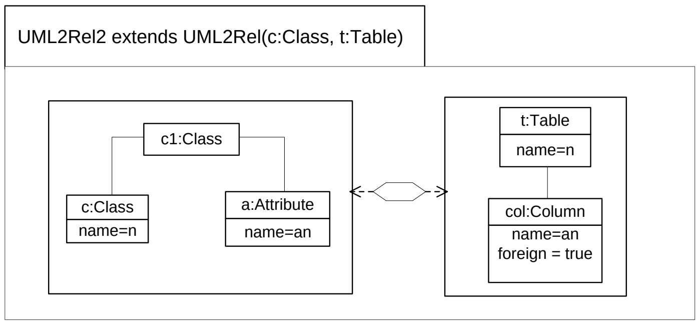


Figure 5.10.: Classes to Table Transformation Extended to Include Associations

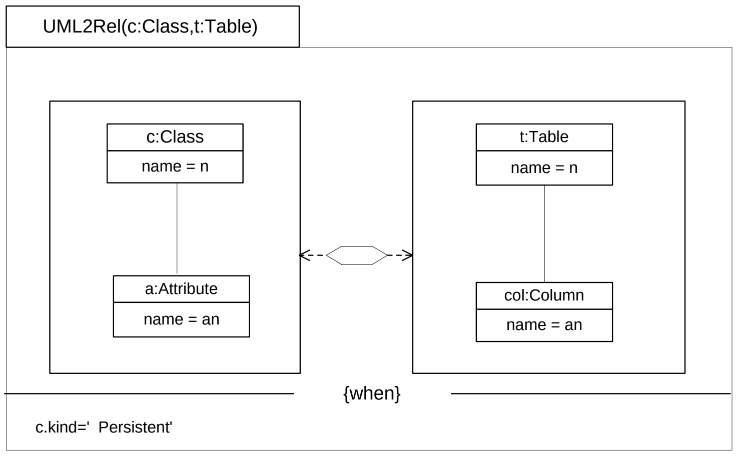


Figure 5.11.: Relation Annotated with OCL Script

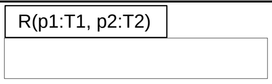
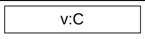
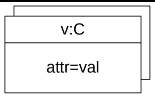
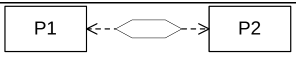
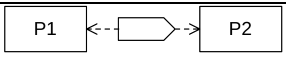
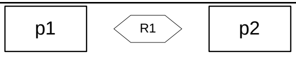
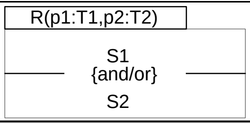
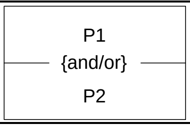
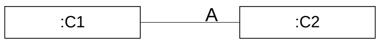
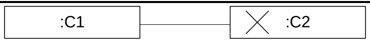
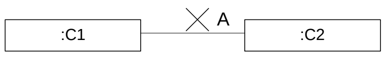
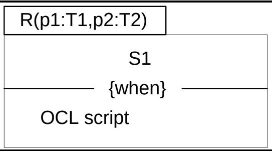
	A transformation R with two parameters. p1 of type T1 and p2 of type T2
	A pattern that matches each instance of class `C` with the attribute `attr` having the value `val`. The name `v` is optional and when given it may be used to refer to the matched instance in the rest of the specification.
	A pattern that matches all instances of class `C` with the attribute `attr` having the value `val`. The name `v` may be used to refer to the complete set of matched instances in the rest of the specification.
	Definition of a relation between `pattern P1` and `pattern P2`. The relation holds between the two patterns if for every occurrence of `pattern P1` there exists an occurrence of `pattern P2` and vice-versa.
	Definition of a mapping from `pattern P1` to `pattern P2`. Defines how each occurrence of `pattern P1` can be transformed into an occurrence of `pattern P2`.
	Transformation invocation. Evaluates to `true` if `p1` is related to `p2` by relationship `R1`.
	Definition of a `transformation R` as a composition of `S1 and/or S2`.
	Definition of a pattern as a composition of patterns `P1 and/or P2`.
	A pattern that matches each pair of instances of `Class C1` and `Class C2` that are linked to each other by an instance of `Association A`.
	A pattern that matches each instance of `Class C1` that is not linked to any instance of `Class C2` by an instance of `Association A`.
	A pattern that matches each pair of instances of `Class C1` and `Class C2` that are not linked to each other by an instance of `Association A`.
	A `Transformation S1`, annotated with an `OCL` expression.

Figure 5.12.: Visual Notation Table

Chapter 6.

Infrastructure

6.1. Introduction

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 in this chapter is to precisely define a core language for defining model transformations. The language strives to exhibit the following features:

- It is *declarative* with respect to the mechanical steps necessary to achieve transformations defined in the language. In other words, we strive to distance ourselves from viewing the language 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 semantics given separately in appendix B.3; access to the semantics is not necessary in order to use the infrastructure.

The infrastructure language is intended to be minimal in the sense that there is no unnecessary redundancy of features. Richer languages are intended to be constructed as *sugarings* of this language.

6.2. Models

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.

As a simple example, consider the model in figure 6.1. We can represent this model as an object diagram in textual form:

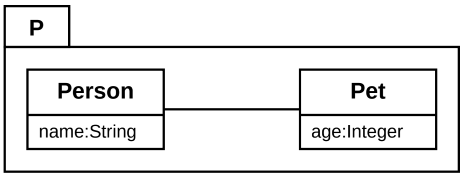


Figure 6.1.: Example model

```
(Package) [
  name = ’P’,
  contents = {
    (Class, Person) [
      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]}]}]
```

6.3. 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 infrastructure language 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 container with *immutable state* 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])]}
```

Atomic values include numbers, strings, booleans, characters etc. We assume that there is a collection of built in relations and mappings that are defined for these atomic values. Sets are unordered collections of values. We assume that there are suitable built in relations and mappings for these values. In particular we require the powerset operator `power` that maps a set to the set of all its sub-sets. A typical object has the form `(x)[g = 10; h = 100, 200]` where `x` is the type of the object. Relations and mappings (transformations) have a similar format, they are the result of evaluating relation and mapping (transformation) *definitions*. When the definition is evaluated, the current environment associated variable names with values. The environment is captured when the transformation is created in order that free references to variables can be resolved when the transformation is invoked.

A syntax for values is defined in appendix B.1. A notion of value ordering is defined in appendix B.2.

6.4. Relations

Our proposal supports both transformation *specifications* and transformation *implementations*, in the forms of relations and mappings respectively, with both using standard notation and semantics. This section describes the 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: "new Dog(a);" in Java
  class = 'Dog',
  args = {
  (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 one 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. The following relationships specify the constraints that must hold between a model and a set of names. The specification does not state *how* to calculate the names, it simply states *what* must hold between the model and the set of names:

```
PNAMES =  // Calculate the names in a package.
  (Package)[name = n, classes = C] <=> {n | N}
    when
      C->forAll(c | power(N)->exists(N’ | CNAMES(c, N’))

CNAMES = // Calculate the names in a class.
  (Class) [name = n, attributes = A] <=> {n | N}
    when
      A->forAll(a | power(N)->exists(N’ | ANAME(a, N’))

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’]
    when
     CHANGEPNAMES(C, prefix, C’)
      where
       CHANGEPNAMES = CHANGEPEMPTY + CHANGEPONE
        where
          CHANGEPEMPTY = {} <=> prefix <=> {}
          CHANGEPONE = {c | C} <=> prefix <=> {c’ | C’}
           when
             CHANGECNAME(c, prefix, c’) and
             CHANGEPNAMES(C, prefix, 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[1]:

```
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 of the problem at hand. 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 infrastructure language provides a combinator for constructing relations from multiple aspect relationships.

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) [
```

[1]This task can also be defi ned using iterate, however iterate can become unwieldy for tasks above a certain size.

```
          name = n,
          attributes = A]
        <=>
        (JavaClass) [
          name = n,
          fields = F]
        when
          ATTSTOFIELDS(A, F)
            where
              ATTSTOFIELDS = ATOFEMPTY + ATOFSTEP
                where
                  ATOFEMPTY = {} <=> {}
                  ATOFSTEP = {a | A} <=> {f|F}
                    when
                      FIELD(a, f) and
                      ATTSTOFIELDS(A, F)
                        where FIELD = ...
      METHODS =
        (Class) [
          name = n,
          attributes = A]
        <=>
        (JavaClass) [
          name = n,
          methods = { | G, U}]
        when
          ATTSTOACCESSORS(A, G) and
          ATTSTOUPDATERS(A, U)
           where
             ATTSTOACCESSORS = ...
             ATTSTOUPDATERS = ...
```

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]}]
```

6.5. Mappings

Mappings are transformations where a choice has been made regarding which are the inputs to the mapping and which are the outputs. It is sometimes the case that relations are easier to define than mappings – relations benefit from the fact that they need not be complete. In this way relations can be viewed as a transformation *specification* and mappings as the corresponding *implementation*.

The infrastructure language allows relations and mappings to be mixed and therefore specifications can be refined to implementations in an incremental manner. The language provides a syntax for mappings that is similar to that for relations. A mapping uses the same syntax for patterns and expressions; however the format of a mapping is slightly restricted compared to that of a relation. In general a binary relation has the form:

```
p1 <=> p2 when c
```

The patterns `p1` and `p2` may independently bind variables; i.e. there may be variables in `p1` that are not in `p2` and vice versa. The condition `c` will be evaluated in the scope of the variables bound by both patterns. In contrast a mapping has the form:

```
p when c => e
```

where `p` may bind variables. The condition `c` will be evaluated in the scope of the variables bound by the pattern. The expression `e` will be evaluated in the scope of the variables bound by the pattern.

Consider the specification of the relation `PNAMES` in the previous section. The relation just specifies how to check that the names of a package have been correctly calculated. A mapping for `PNAMES` describes *how* to construct the set of names:

```
PNAMES =  // Calculate the names in a package.
  (Package)[name = n, classes = C] =>
  {n | C->collect(c | CNAMES(c))->flatten}
CNAMES = // Calculate the names in a class.
  (Class)[name = n, attributes = A] =>
  {n | A->collect(a | ANAME(a))}
ANAME =  // Produce the name of an attribute.
  (Attribute)[name = n] <=> {n}
```

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])
```

6.6. Abstract syntax

Figure 6.2 shows the infrastructure abstract syntax package. This package can be merged with the standard MOF definition to produce an extended version of MOF.

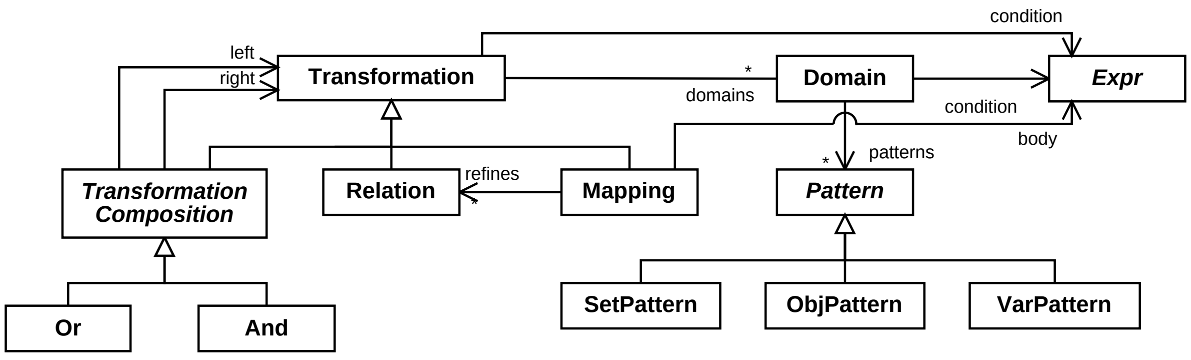


Figure 6.2.: Transformations abstract syntax extension to the MOF meta-model

Transformations contain a number of domains. Each domain has a pattern and a constraint which constrains that pattern. The Transformation itself is abstract and can not be directly instantiated: transformations are concretely realised as relations or mappings. A relation accepts values for the domains and evaluates to true if the values supplied satisfy the relation, otherwise it evaluates to false. If the relation evaluates to false, the transformation engine backtracks to a relevant choice point. Mappings accept values and if they successfully match against all the mappings domains, it evaluates to produce a target value. Mappings can refine an arbitrary number of other relations (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. Transformations are composed using `And` and `Or`, represented syntactically in the examples as `x` and `+` respectively.

The syntax of the infrastructure language is defined as follows:

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

6.7. Semantics

The semantics of the language is given in terms of an abstract QVT machine in appendix B.3. 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.

6.8. 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.

The problem posed by serialization of UML to XMI is that UML models are structured as a *graph* whereas XMI is structured as a *tree*. To allow graphs to be represented as trees, XML provides *references*. Each XML element can be provided with a unique identifier; subsequent references to the element in the XML document then use reference valued attributes to refer to the appropriate identifier.

The *flattening* of a graph to a tree is much easier to specify than to implement. To specify the flattening we assume that flattening has been performed and that we have been supplied with both the graph and the tree and then check that the tree is a faithful representation of the graph. If we also assume that whoever performed the flattening supplied us with a table containing associations between graph nodes and XML references then we can easily check the structure by traversing the tree and checking that it matches the graph modulo the reference information.

The specification of flattening makes no assumptions about what was supplied and what was synthesized. In the case of UML to XMI we might wish to specify conformance without committing to whether the transformation is to be implemented as a serialization of UML models or as a de-serialization.

If we *implement* the flattening then we must describe an algorithm for traversing a graph and building the table of reference information. There are a number of alternative strategies; we may not wish to commit to one particular strategy at this time or we may want to check that someone else has flattened the graph correctly in which case we cannot possibly guess the particular strategy that they chose. Note that in implementing a transformation we must make a choice about what is supplied and what is synthesized.

The transformation of UML models into XMI highlights the difference between relations (specifications) and implementations (mappings). Both are useful and are related to each other. We may choose to use one or the other to represent a given transformation and, when developing a large transformation we may choose to use relations and mappings to implement different parts of the whole.

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. A class may be transformed to either an XML element of type `ClassRef` or to an XML element of type `Class`. We do not with to commit at the specification stage because we don't know the flattening strategy:

```
MAPCLASS = MAPREF + MAPMODEL
  whererec
    MAPREF =
      (Element)[tag = 'ClassRef', attributes =
        {(Att)[name = 'id', value = i]}]
      <=>
      {(i, c)|T}
      <=>
      (Class, c)[name = n]
    MAPMODEL =
      (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 XML elements of type `Attribute` to attributes in the static UML model. We assume that the type of an attribute is represented as a child element in the XMI. Since the relation MAPCLASS is non-deterministic with respect to the format of the type of the attribute, we leave open many different implementation strategies:

```
MAPATTS = MAPEMPTY + MAPATTMODEL
  whererec
    MAPEMPTY = {} <=> T <=> {}
    MAPATTMODEL
      where
        MAPTYPEREF =
          {(Element)[
            tag = ’Attribute’,
             attributes = {
               (Att)[name =’name’, value = n]},
            children = {e}
          <=>
          T
          <=>
          {(Attribute)[name = n, type = t] | A}
            when
              MAPATTS(E, T, A) and
              MAPCLASS(e, T, t)
```

6.9. Application: Removing Associations

QVT must be able to address the issue of 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. Since the specification requires that the identity of the package remains the same across the relation, this specification enforces update 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})
```

Chapter 7.

Superstructure translation

This section details the translation of the superstructure into the infrastructure. This definition is given in terms of our proposal in order to show its power.

This section is not yet complete, but serves as an example of how the technique works.

In general terms, many superstructure constructs have fairly direct equivalents in the infrastructure. However several superstructure constructs have no direct equivalent in the infrastructure, and are translated into compound infrastructure constructs in order to define their semantics. The following are examples of superstructure constructs that have no direct infrastructure equivalent:

- Extension.
- Refinement.
- Choices.
- Named definitions.
- Domains.

7.1. Translation

In the following, superstructure elements are assumed to come from the `Su` meta-model, and infrastructure elements from the `In` meta-model.

```
SyntaxM = disjunct(TopLevelM, RelationM)

mapping TopLevelM {
  domain { (Su.TopLevel)[definitions = D, expressions = E] }
  body {
    let (
      bindings = D->collect(d |
        disjunct(RelationM(d), MappingM(d)))
      body = E->at(0)
      body = E->subseq(1, E->size())->iterate(e a = body |
        (In.Sequential)[left = a, right = SyntaxM(e)])
    V
    in {
      (In.LetRec)[bindings = bindings, body = body]
    }
  }
}

mapping RelationM {
  domain {
    (Su.Relation)[
      name = n,
      extends = E,
      domains = D,
      when = w]
  }
  body {
    (In.Disjunct)[
      disjuncts = E->append(
        (In.Relation)[
          name = n,
          domains = D->collect(d | DomainM(d)),
          whenn = WhenM(d)
        ]
      )
    ]
  }
}

mapping MappingM {
  domain {
    (Su.Mapping)[
      name = n,
      extends = E,
      refines = R,
      domains = D,
      when = w]
  }
  body {
    (In.Disjunct)[
      disjuncts = E->append(
        (In.Mapping)[
          name = n,
          domains = D->collect(d | DomainM(d)),
          whenn = WhenM(d)
        ]
      )
    ]
  }
}

mapping DomainM {
  domain { (Su.Domain)[pattern = p, when = w] }
  body {
    PatternM(p)
    WhenM(w)
  }
}
```

Chapter 8.

Compliance points

All tools which complain compliance with this submission must fully implement either or both of the super-structure textual syntax and visual syntax. Full support of either syntax implies that the tool supports full XML interchange of the appropriate syntax.

Part III.

APPENDICES

Appendix A.

Examples

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

A.1. Multiple to single inheritance

A.2. Classes to tables

A.3. Information system model to J2EE

A.4. Restricted Associations

This example is from the Codagen post to the `mofqvt@omg.org` mailing list[1]. The idea behind the transformation specification is to translate 'restricted associations' in a simple modelling language into pseudo-Java. In this section we use 'JM' and 'SM' to stand for Java Model and Simple Model respectively.

The meta-model for SM is shown in figure A.1, and that of JM in figure A.2.

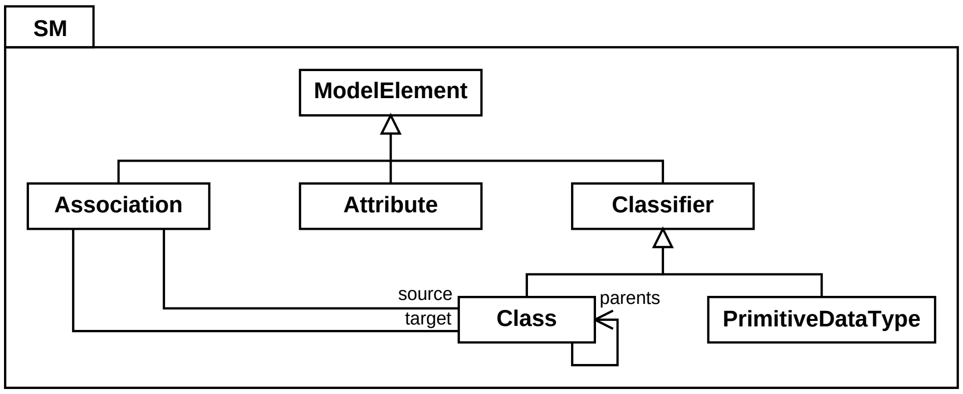


Figure A.1.: SM meta-model

Figure A.3 shows an example model with a restricted association. The association between `Manager` and `Package` is implicitly a restricted association of that between `Employee` and `Wage`. When this example is translated to code `Employee` should initially become:

[1]The message was posted Tuesday 1st July 2003 by Eric Briere.

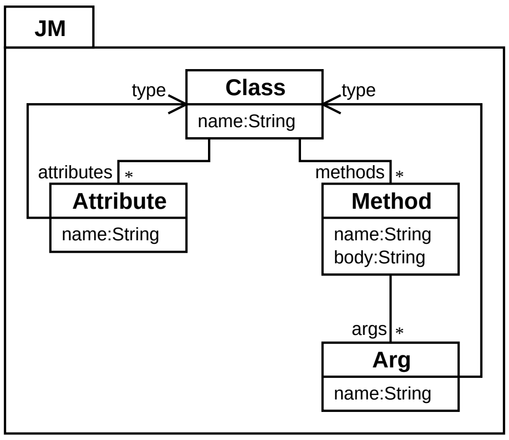


Figure A.2.: JM meta-model

```
class Employee {
  Wage aWage;
  public void setWage(Wage wage) {
    aWage = wage;
  }
}
```

When `Manager` is encountered, it should update the `Employee` class to add a static attribute `childrenHasResticted` which should be set to a value of true. `Manager` should be translated to:

```
class Manager extends Employee {
  public void setWage(Scheme scheme) {
    aWage = scheme;
  }
}
```

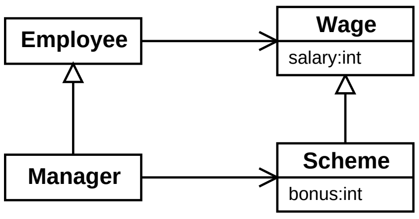


Figure A.3.: Restricted associations example

We create two versions of the transformation for restricted association: one specification with relations, another implementation with mappings. Figure A.4 shows a high level model of the transformation as a set of relations.

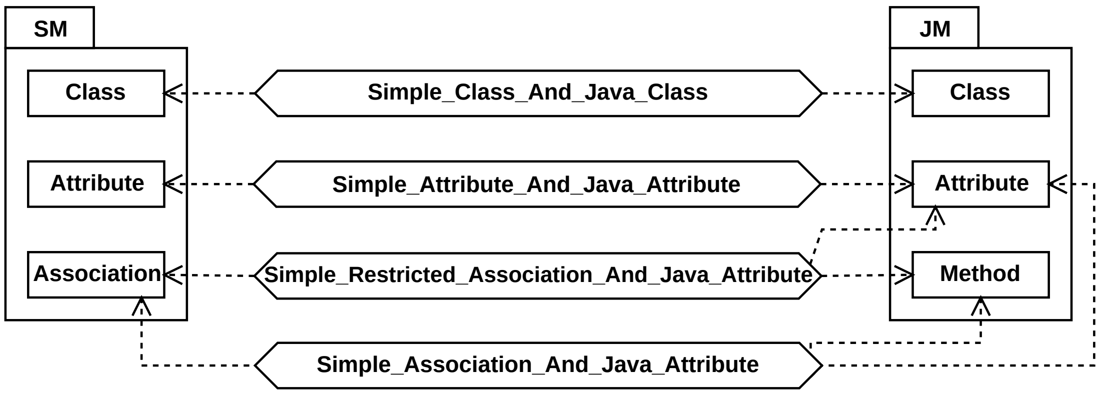

SM
Class
Attribute
Association
Simple_Class_And_Java_Class
Simple_Attribute_And_Java_Attribute
Simple_Restricted_Association_And_Java_Attribute
Simple_Association_And_Java_Attribute
JM
Class
Attribute
Method

Figure A.4.: Simplified model of Restricted Association transformation

A.4.1. Relations

Firstly we create a series of relations for the transformation. These relations are fairly heavily annotated in an attempt to better explain to the reader what the transformation is trying to achieve.

```
// Enforce that every SM class’s JM mirror contains valid mirrors of its attributes and
// that every parent SM class is mirrored in the JM version too.

relation Simple_Class_And_Java_Class {
  domain { (SM.Class)[name = n, attributes = sA, parents = sP] }
  domain { (JM.Class)[name = n, attributes = jA, parents = jP] }
  when {
    sA->forAll(a | Simple_Attribute_And_Java_Attribute(a, jA.toChoice())) and
    sP->forAll(p | Simple_Class_And_Java_Class(p, jP.toChoice()))
  }
}

// Enforce that every SM attribute is mirrored by a correct JM attribute. For this to
// be the case, the type of the SM attribute will have to have been mirrored correctly
// as well.

relation Simple_Attribute_And_Java_Attribute {
  domain { (SM.Attribute)[name = n, type = sT] }
  domain { (JM.Attribute)[name = n, type = jT] }
  when {
    Simple_Class_And_Java_Class(sT, jT)
  }
}

// This relation has three domains. The intuitive idea is that the first domain is
// a fixed SM association and the second domain is a choice of size >= 1 of
// associations.
//
// The first domain will match an association whose ends are subclasses of the
// association ends in the second domain.
//
// Given that those two match, the third domain then ensures that the correct
// attributes and methods exist in the classes that the associations are translated
// to.
//
// If this relation doesn’t succeed, then the intention is that the user should fall
// back on Simple_Association_To_Java_Attribute.

relation Simple_Restricted_Association_And_Java_Attribute {
  domain { (SM.Association)[source = csS, destination = csD] }
  domain { (SM.Association)[source = psS, destination = psD] }
  domain {

// In the parent class, we need an attribute for the association name, and also an
// attribute 'childrenHasRestictedAssociation' to show that this class has restricted
// associations.

    (JM.Class, pC)[
      attributes = {
        (JM.Attribute)[name = "a" + psD.name, type = pD],
        (JM.Attribute)[
          name = "childrenHasRestictedAssociation",
          is_static = true,
          default_value = "true"]
        | _
      },
      methods = {(JM.Method)[
        name = "set" + psD.name,
        args = [(JM.Arg)[name = psD.name.lower(), type = cD]],
        body = "a" + psD.name + " = a" + psD.name.lower()]
        | _ }
    ],

// In the class which the restricted association itself gets translated to, we merely
// need a method.

    (JM.Class, cC)[
      methods = {(JM.Method)[
        name = "set" + psD.name,
        args = [(JM.Arg)[name = csD.name.lower(), type = psD]],
        body = "a" + psD.name + " = a" + csD.name.lower()]
        | _ }
    ],

// We need to make sure that cD and pD are classes, but don't care much more about them.

    (JM.Class, cD)[],
    (JM.Class, pD)[]
  }
  when {
    csS.hasParent(psS.toChoice()) and csD.hasParent(psD.toChoice()) and
    Simple_Class_And_Java_Class(psS, pC) and
    Simple_Class_And_Java_Class(csD, cD)
    Simple_Class_And_Java_Class(psD, pD)
    Simple_Class_And_Java_Class(csS, cC)
  }
}

// This ensures that associations which don't have subsequent restrictions get
// mirrored correctly. Note that this doesn't explicitly check that the association
// does not get restricted, but assumes that by the semantics of disjunction, if
// Simple_Restricted_Association_And_Java_Attribute succeeded then this relation will
// not be called upon.

relation Simple_Association_And_Java_Attribute {
  domain { (SM.Association)[name = n, source = sS, destination = sD] }
```

```
    domain { (SM.Class, jS)[attributes = { (SM.Attribute)[name = n, type = jD] | A}] }
    when {
      Simple_Class_And_Java_Class(sS, jS) and
      Simple_Class_And_Java_Class(sD, jD)
    }
  }

  // This is the main relation. The first domain expects to receive a set of SM instances,
  // the second domain expects to receive a set of JM instances.

  relation Restricted_Associations_And_Java {
    domain { S }
    domain { J }
    when {
      S->forall(s | e
        disjunct(Simple_Class_And_Java_Class(s, J.toChoice()),
          Simple_Restricted_Association_And_Java_Attribute(s, S.toChoice(), J.toChoice()),
          Simple_Association_And_Java_Attribute(s, J.toChoice())))
    }
  }
```

A.4.2. Mappings

This section presents a number of mappings which refine the relations given in the previous section. Since there is a close correspondence between these and the relations previously given, little annotation is needed in this section.

```
  mapping Simple_Class_To_Java_Class refines Simple_Class_And_Java_Class {
    domain { (SM.Class)[name = n, attributes = A] }
    body {
      (JM.Class)[
        name = n,
        attributes = A->iterate(a as = {} |
          as + Simple_Attribute_To_Java_Attribute(a))
      ]
    }
  }

  mapping Simple_Attribute_To_Java_Attribute refines Simple_Attribute_And_Java_Attribute {
    domain { (SM.Attribute)[name = n, type = sT] }
    body {
      (JM.Attribute)[name = n, type = Simple_Class_To_Java_Class(jT.toChoice())]
    }
  }

  mapping Simple_Restricted_Association_To_Java_Attribute
    refines Simple_Restricted_Association_And_Java_Attribute {
    domain { (SM.Association)[source = csS, destination = csD] }
    domain { (SM.Association)[source = psS, destination = psD] }
    when { csS.hasParent(psS) and csD.hasParent(psD) }
    body {
      (JM.Class, pC)[
        attributes = {
          (JM.Attribute)[name = "a" + psD.name, type = pD],
          (JM.Attribute)[
            name = "childrenHasRestictedAssociation",
            is_static = true,
            default_value = "true"]
        },
        methods = {(JM.Method)[
          name = "set" + psD.name,
          args = [(JM.Arg)[name = psD.name.lower(), type = cD]],
          body = "a" + psD.name + " = a" + psD.name.lower()]}
      ];

  // In the class which the restricted association itself gets translated to, we merely
  // need a method.

      (JM.Class, cC)[
        methods = {(JM.Method)[
          name = "set" + psD.name
          args = [(JM.Arg)[name = csD.name, type = psD.name.lower()]]}
      ];
      (JM.Class, cD)[];
      (JM.Class, pD)[]
    }
  }

  mapping Simple_Association_To_Java_Attribute
    refines Simple_Association_And_Java_Attribute {
    domain { (SM.Association)[name = n, source = sS, to = destination = sD] }
    body {
      (SM.Class, jS)[attributes = {
        (SM.Attribute)[name = n, type = jD] | A}]
    }
  }

  mapping Restricted_Associations refines Restricted_Associations_And_Java {
    domain { S }
    domain { J }
    when {
      S->forall(s | e
        disjunct(Simple_Class_To_Java_Class(s, J.toChoice()),
          Simple_Restricted_Association_To_Java_Attribute(s, S.toChoice(), J.toChoice()),
          Simple_Association_To_Java_Attribute(s, J.toChoice())))
    }
  }
```

A.5. More examples

A.5.1. Object Relational Mapping

This example illustrates the generation of an object-relational data manager layer from a UML class model in a series of transformation steps.

First, a UML class model is transformed into a relational model. Then, the relational model is transformed into a set of standard query models that specify insertion, selection, modification and deletion of a row in a table.

A persistent class is transformed to add a set of standard operations, Create, Get, Update and Delete, corresponding to these queries. A further set of transformations then transforms these query models into platform-specific implementations (e.g.:JDBC, ODBC, Pro*C, etc).

The example also illustrates how various aspects can be composed with the standard data manager.

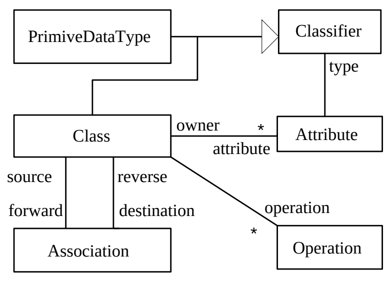


Figure A.5.: A simple UML meta-model

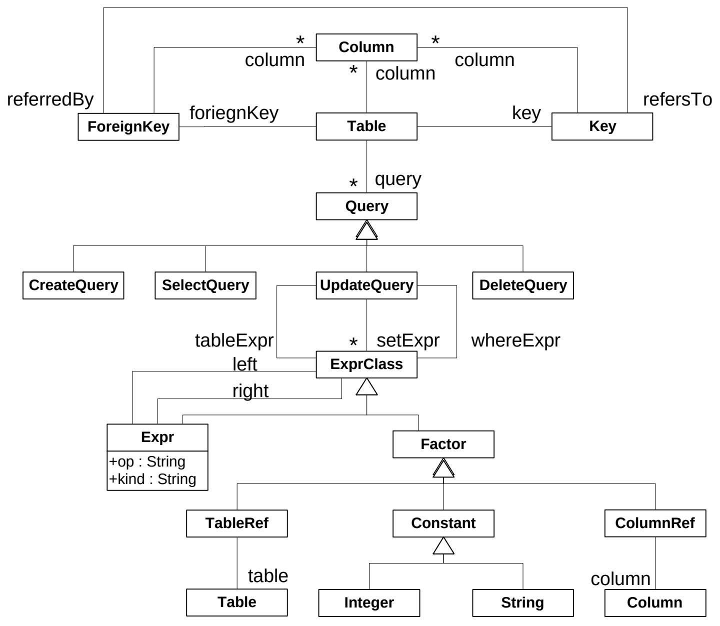


Figure A.6.: A simple RDBMS meta-model

Standard data manager

A simplified UML meta-model is shown in Figure A.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 mapping. Some attributes have the property `kind` set to `Primary` to indicate that they are the key attributes.

A sample RDBMS meta-model is shown in Figure A.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 associated table. A table has four associated queries – insert query to insert a row into the table, a select query to select a row, an update query to update a row, and a delete query to delete a row. The figure also shows a simplified meta model of the update query.

Figure A.7 specifies class to table transformation. 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.

Figure A.8 specifies the transformation of an association. 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.

Figure A.9 specifies the standard methods and corresponding queries between a class and table.

Figure A.10 specifies the generation of an update query for a table.

A typical update query has the following form:

```
update <table>
set
   <non_primary_key_column>_1 = <value>
   and
   ..
   <non_primary_key_column>_n = <value>
where
   <primary_key_column>_1 = <value>
   and
   ..
   <primary_key_column>_k = <value>
```

The figure does not show the other queries, but can be generated on similar lines.

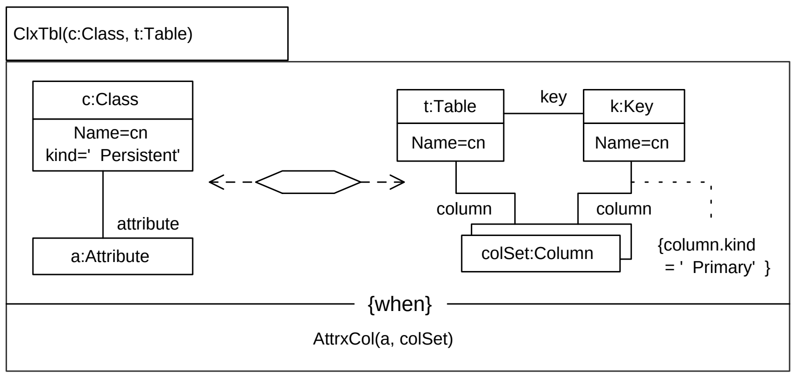
Figure A.9 shows the 'standard data manager layer' transformation that is composed from the transformations described above.

Optimistic locking aspect

Now we consider how a concurrency management aspect based on the optimistic locking strategy can be composed with the standard data manger.

The concurrency aspect we are considering handles the problem of update losses. This is typically handled by taking a lock on the record to be updated so that no other concurrent transaction can update the same record. But this has performance overheads. Optimistic locking strategy uses a 'version' field in the record that is incremented with each update. A conflicting update can then be detected by checking the version field. In a transaction, an Update() operation is preceded by a Get() operation. The Get() operation retrieves the current version number and Update() increments it. The update query with optimistic locking looks as follows:

```
update <table>
set
   <non_primary_key_column>_1 = <value>
   and
   ..
   <non_primary_key_column>_n = <value>
   and
   version = version + 1
```



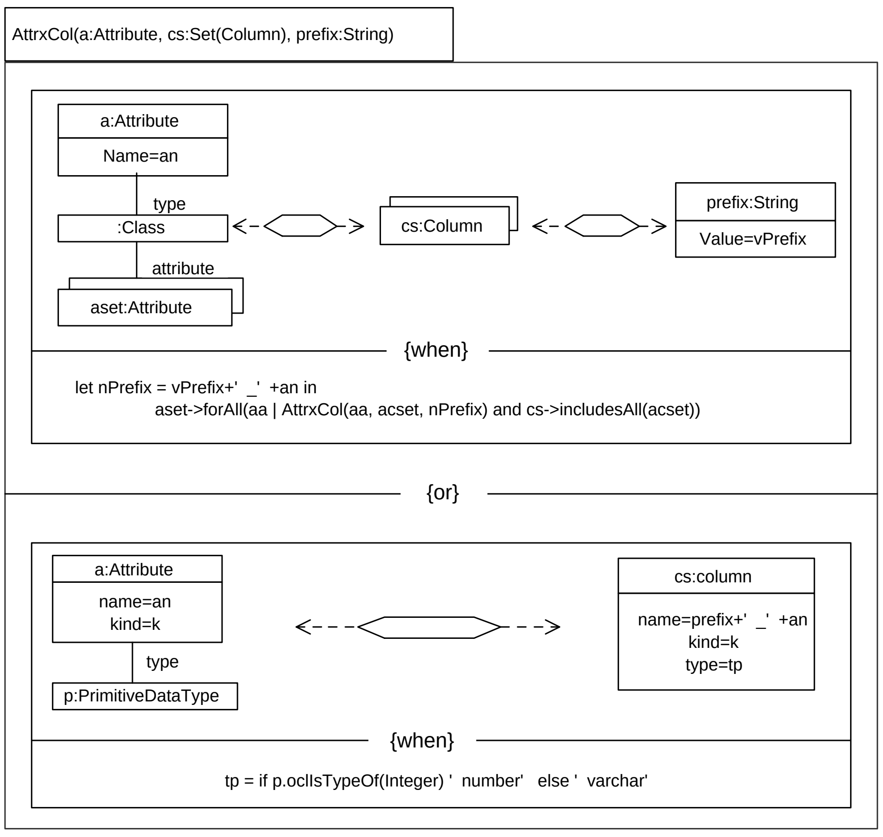

AttrxCol(a:Attribute, cs:Set(Column), prefix:String)
a:Attribute
Name=an
type
:Class
attribute
aset:Attribute
cs:Column
prefix:String
Value=vPrefix
{when}
let nPrefix = vPrefix+' _' +an in
aset->forAll(aa | AttrxCol(aa, acset, nPrefix) and cs->includesAll(acset))
{or}
a:Attribute
name=an
kind=k
type
p:PrimitiveDataType
cs:column
name=prefix+' _' +an
kind=k
type=tp
{when}
tp = if p.oclIsTypeOf(Integer) ' number' else ' varchar'

Figure A.7.: Class to Table Transformation

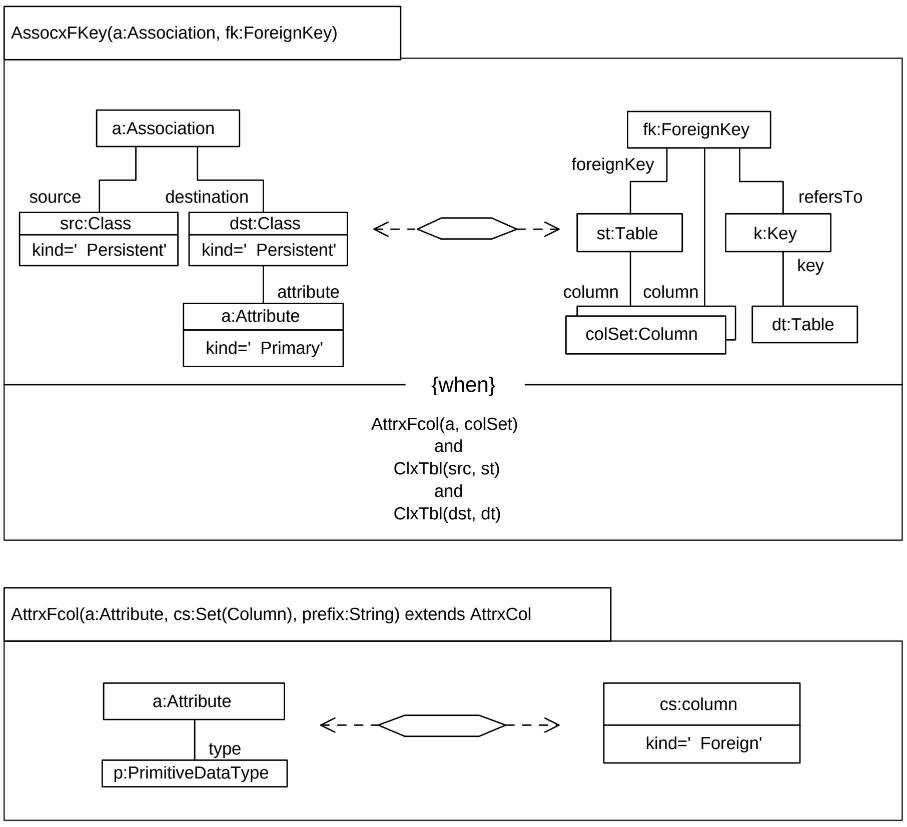


Figure A.8.: Association to Foreign Key Transformation

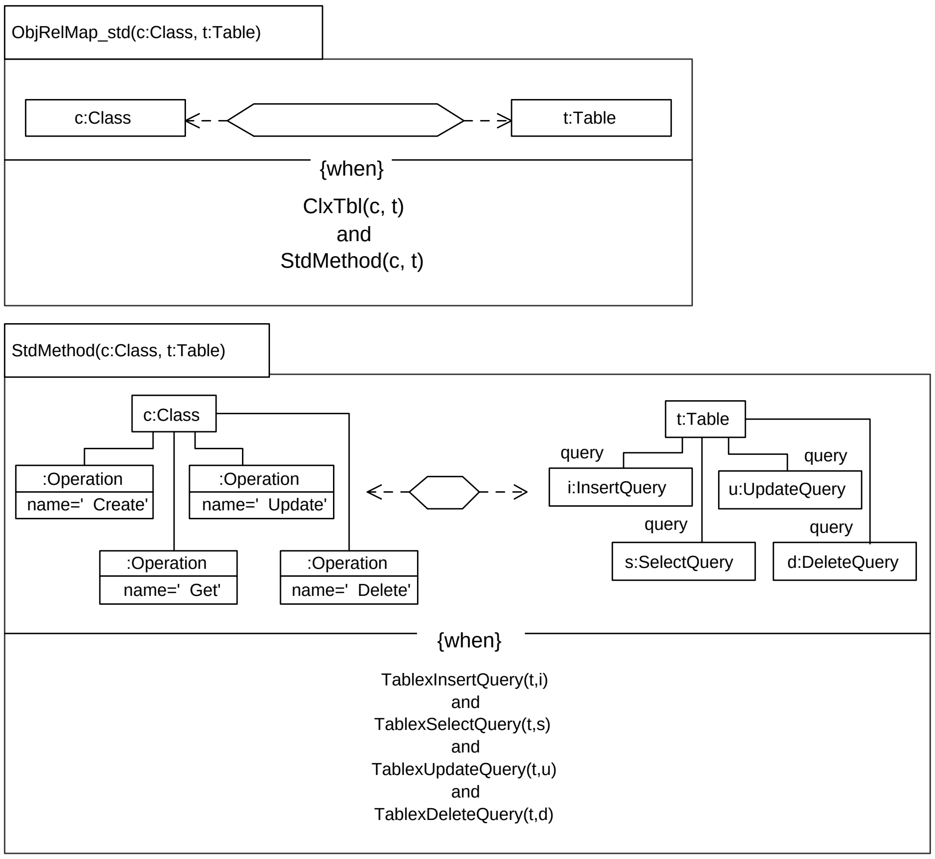


Figure A.9.: A composite transformation for standard data manager

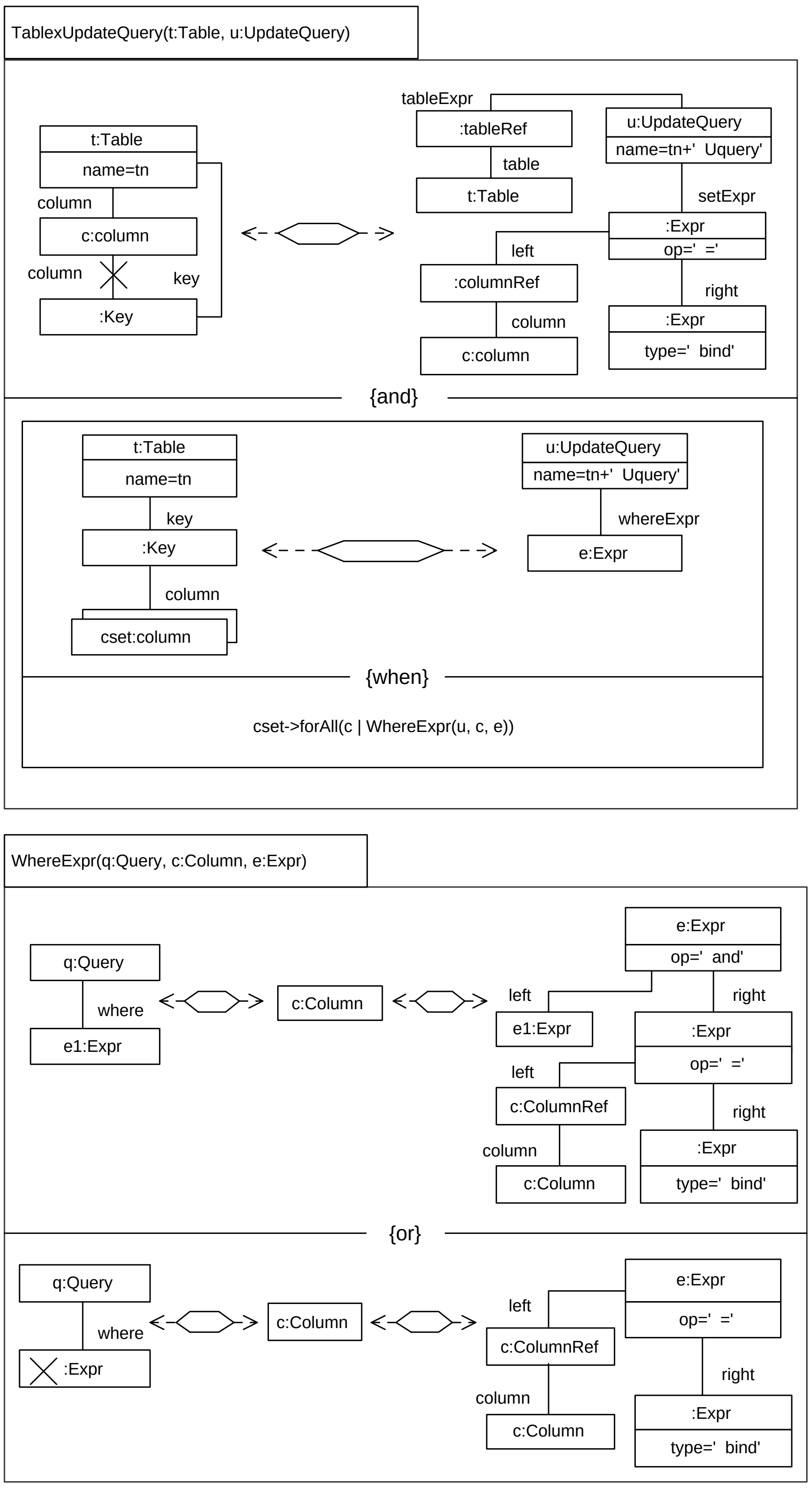


Figure A.10.: Update Query Transformation

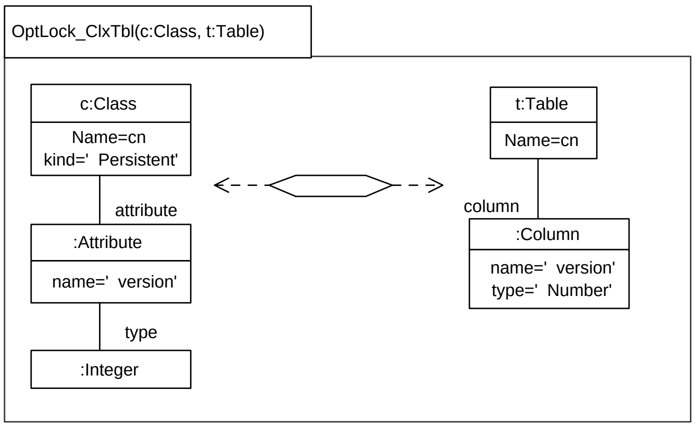


Figure A.11.: Transformation to add version field

```
where
   <primary_key_column>_1 = <value>
   and
   ..
   <primary_key_column>_k = <value>
   and
   version = <current value>
```

Figure A.11 shows the transformation that adds an attribute and a column to represent the version field.

Figure A.12 specifies the generation of the update query to add the part corresponding to the optimistic locking strategy.

Figure A.13 specifies the composite transformation that represents the optimistic locking aspect.

Figure A.14 specifies the composition of the standard data manager with the optimistic locking aspect.

A.5.2. Flattening of a hierarchical state machine

Figure A.15 shows a simplified state machine meta model.

Figure A.16 shows the transformation of a hierarchical state machine into a flat state machine. The basic step in the transformation transfers the transitions of a composite state to its child states. The transformation works recursively, in a top-down fashion, by starting with the top-level composite states until all composite states have been removed.

The transformation *FlattenStateMachine* performs in-place modifications. It is specified as a disjunction of two parts: the first part removes a composite state and invokes *FlattenStateMachine* recursively, the second part specifies the termination condition for the recursion when there are no more composite states to be removed.

The first part of the transformation is again specified as a conjunction of three sub-parts. In the first conjunct, the left pattern selects a top-level composite state and all its transitions (*tset*) and child states (*cset*), and the right pattern specifies that the parent state and its transitions are to be removed, child states (*cset*) retained, and a new set of transitions (*ttset*) added. The second conjunct specifies that corresponding to each pair of child-state

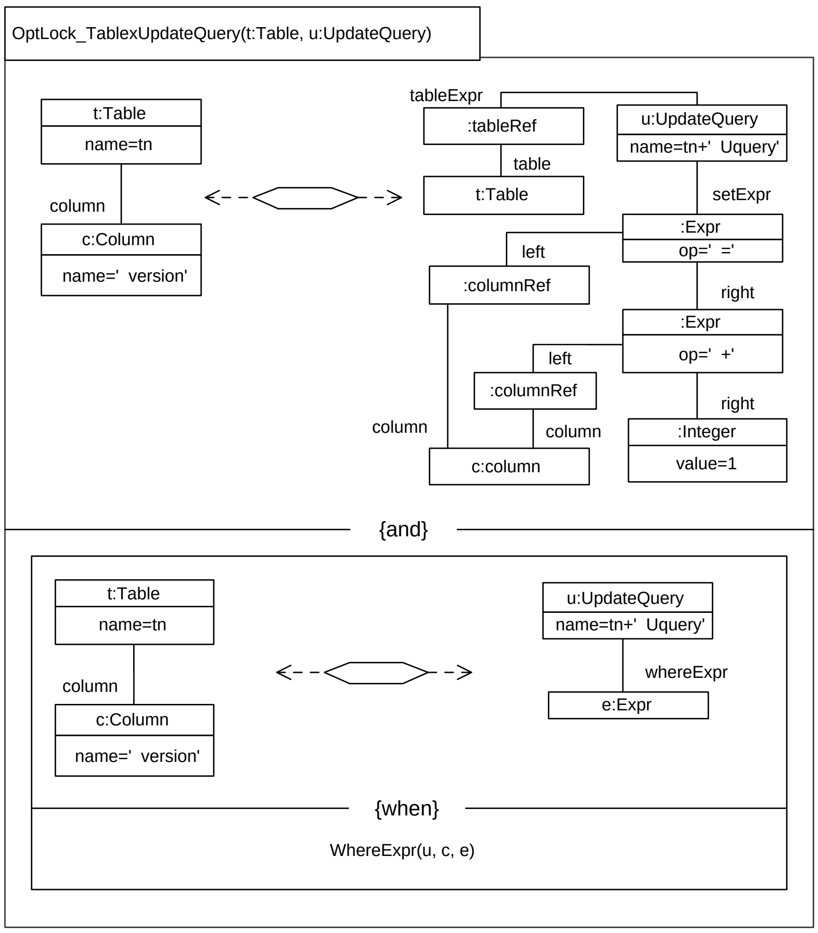


Figure A.12.: Update Query for Optimistic Locking

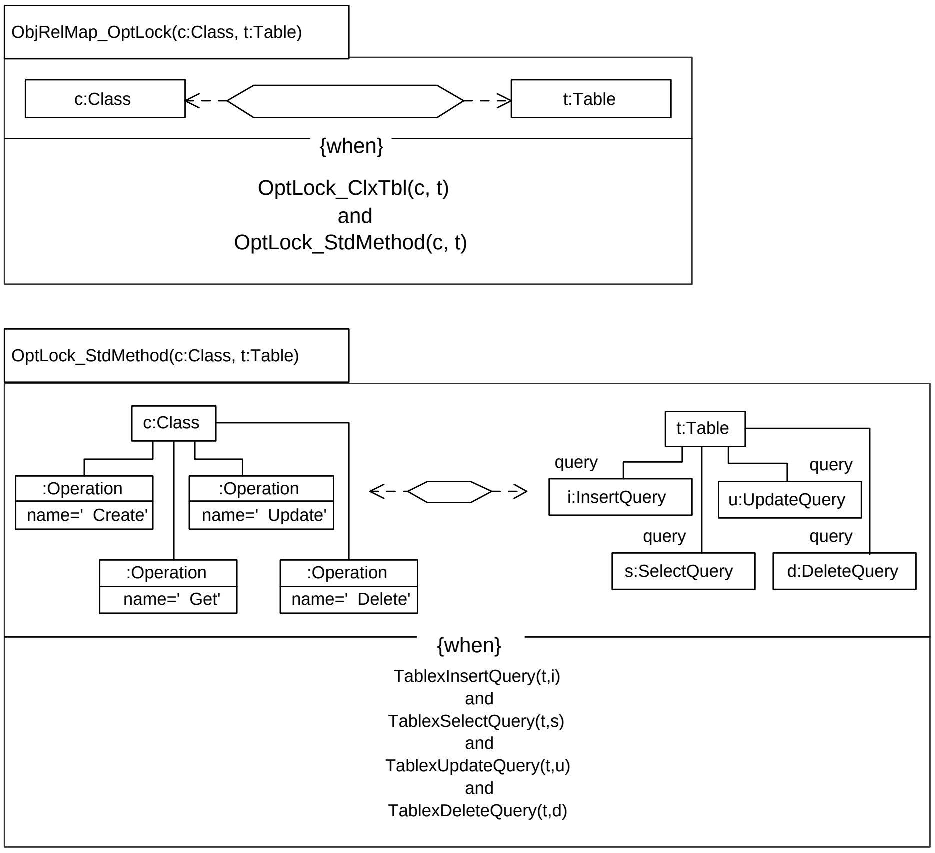


Figure A.13.: A composite transformation for optimistic locking

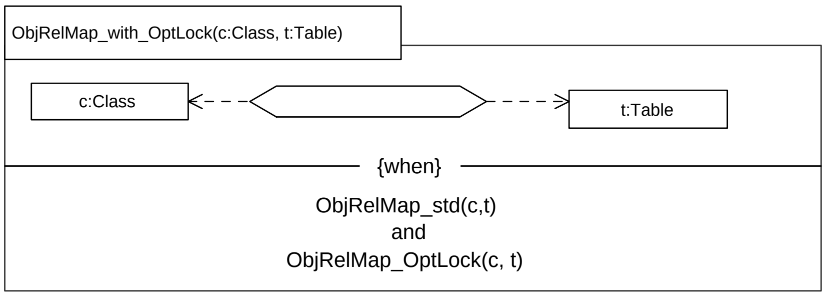


Figure A.14.: Composition of standard data manager with optimistic locking aspect

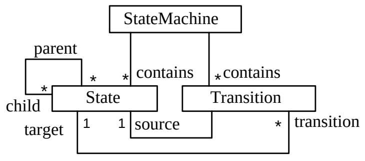


Figure A.15.: A simple state machine meta model

and transition belonging to the sets *cset* and *tset*, a new transition *tt* is to be created and added to the set *ttset*. The third conjunct specifies the recursive invocation.

In the Figure A.16 the parameter m′ refers to m after transformation.

A.5.3. DNF of a boolean expression

Figure A.17 shows the simplified meta-model of a boolean expression tree.

The problem is to transform an arbitrary boolean expression into an equivalent expression that is in disjunctive normal form (DNF).

Figure A.18 shows this transformation. It performs in-place modifications and is specified as a fix point transformation. A fix point transformation step is repeated until a fix point is reached (i.e., no further changes are possible).

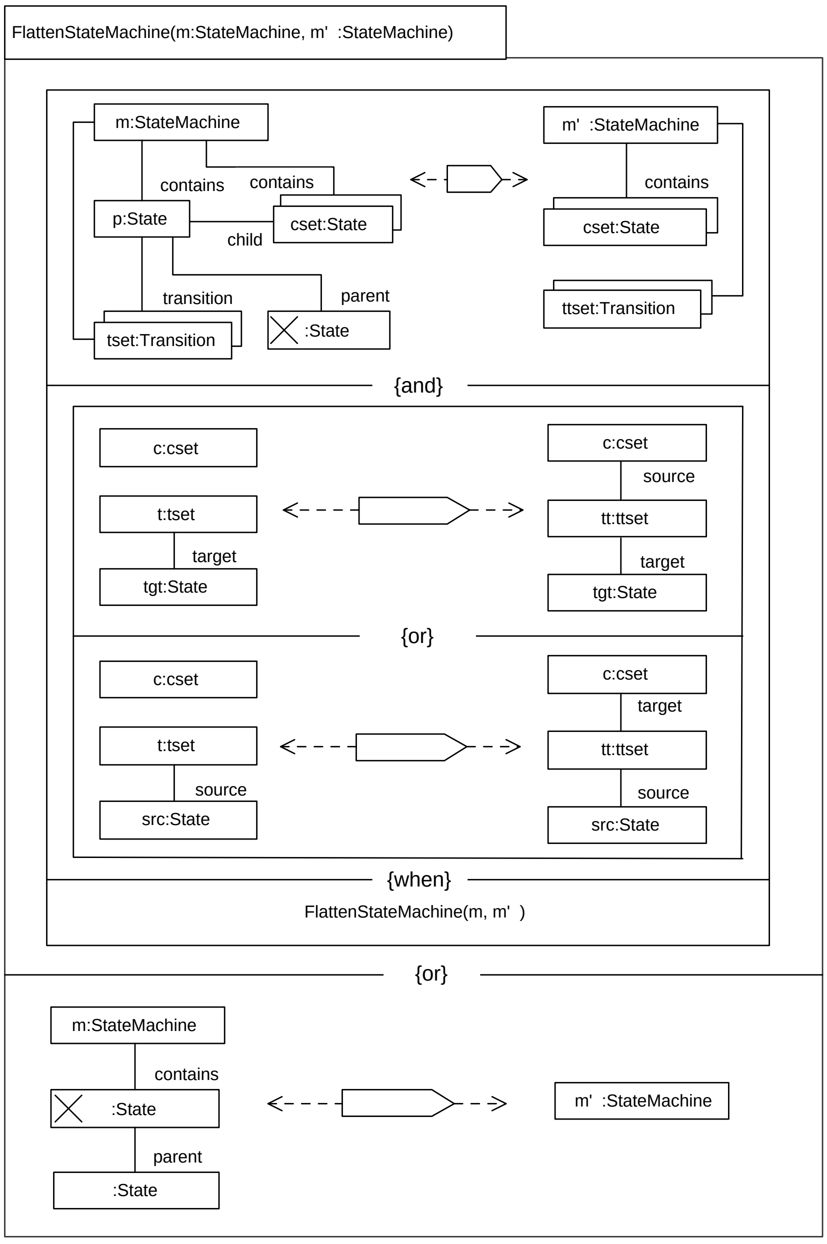


Figure A.16.: State machine flattening

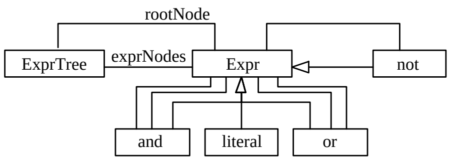


Figure A.17.: A simple boolean expression meta-model

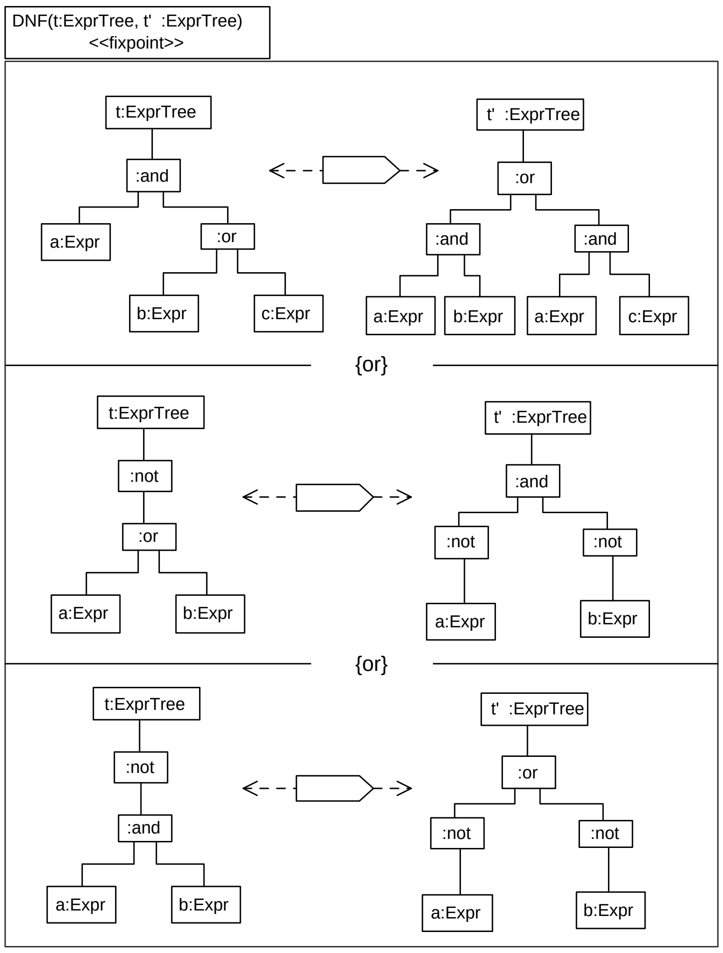


Figure A.18.: UML Class to Relational Table Transformation

Appendix B.

Infrastructure machine definition

This appendix documents the semantics of the QVT machine that provides the complete semantics for our complete infrastructure. The machine definition is given in section B.3. In order to make sense of the machine definition, we first of all define the necessary syntax for values (section B.1), as well as defining ordering and equality for values (B.2).

B.1. Values

The syntax of values is defined as follows:

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

B.2. Value Ordering and Equality

Values are ordered with respect to the amount of information content they have. If two values x and y are ordered $x \leq y$ then y contains all of x and may contain more. Value ordering is used to define value equality and to define a relation that finds the smallest data value that simultaneously contains x and y (the *least upper bound*). The least upper bound of two data values does not always exist; it is used to merge two views on the same data value.

The following rules define an ordering relation on data values. Two atomic values are ordered when they are the same:

$$a \leq a \tag{B.1}$$

Two sets are ordered $P \leq Q$ when for each element of $p \in P$ there is an element $q \in Q$ such that $p \leq q$:

$$\frac{\forall p \in P \bullet \exists q \in Q \bullet p \leq q}{P \leq Q} \tag{B.2}$$

Two objects are ordered when the corresponding classes and field values are ordered:

$$\frac{\bigwedge_{i\in 0\ldots m}(\bigvee_{j\in 0\ldots n} v_i = w_j \;\&\; x_i \leq y_i) \qquad c_1 \leq c_2}{(c_1)[v_i = x_i{}^{i\in 0\ldots m}] \leq (c_2)[w_i = y_i{}^{i\in 0\ldots n}]} \tag{B.3}$$

Note that relations and mappings cannot be compared using the $\leq$ relation. Equality of data value is defined as follows:

Definition 1 *Two values x and y are equal $x = y$ when $x \leq y$ and $y \leq x$.*

Definition 2 *Two values x and y have a least upper bound $z = x \sqcup y$ when this is the smallest data value such that $x \leq z$ and $y \leq z$.*

B.3. Machine definition

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

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

The semantics of the infrastructure language is defined by a machine. Notational conventions are as follows:

- 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 $\mathrm{mkset}(i)$, 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, h, c, f, d) where:

- s is the stack used to contain values. It is a sequence.
- ρ is the current lexical environment.
- h is the heap containing mappings between object identifiers (values) and objects.
- 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:

$$\begin{array}{c}\text{span}(\{1,2,3\},2) = \\ [[\emptyset,\{1,2,3\}], \\ [\{1\},\{2,3\}], \\ [\{2,3\},\{1\}],\ldots]\end{array}$$

Each state transition rule is defined below:

A constant evaluates to itself:

$$(s, \rho, h, k : c, f, d) \longrightarrow (k : s, \rho, h, c, f, d) \tag{B.4}$$

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

$$(s, \rho, h, v : c, f, d) \longrightarrow ((\rho.v) : s, \rho, h, c, f, d) \tag{B.5}$$

A relation definition evaluates to produce a relation. Note that free variables in a relation constraint are satisfied when the relation is invoked by capturing the current lexical environment in the relation:

$$(s, \rho, h, (ps, e) : c, f, d) \longrightarrow ((ps, \rho, e) : s, \rho, h, c, f, d) \tag{B.6}$$

A mapping definition evaluates to produce a mapping. Note that free variables in a mapping constraint and body are satisfied when the mapping is invoked by capturing the current lexical environment in the mapping:

$$(s, \rho, h, (ps, e_1, e_2) : c, f, d) \longrightarrow ((ps, \rho, e_1, e_2) : s, \rho, h, c, f, d) \tag{B.7}$$

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:

$$(s, \rho, h, \{es \mid\ ds\} : c, f, d) \longrightarrow (s, \rho, h, es + ds + [\mathrm{mkset}(|es|, |ds|)] + c, f, d) \tag{B.8}$$

$$(v + w + s, \rho, h, \mathrm{mkset}(i, j) : c, f, d) \longrightarrow ((\bigcup_{k \in 0 \ldots i} \{v_k\} \cup \bigcup_{k \in 0 \ldots j} w_k) : s, \rho, h, c, f, d) \tag{B.9}$$

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:

$$(s, \rho, h, (e_1, e_2)[fs] : s, f, d) \longrightarrow (s, \rho, h, [e_1, e_2] + fs + [\mathrm{mkobj}(|fs|)] + c, f, d) \tag{B.10}$$

When a mkobj instruction is performed we expect some field values fs above a type v_2 above an identity v_1. There are three possible outcomes:

1. The identity does not currently exist in the heap. In this case we just add the new object to the heap and return the identity as a handle for subsequent object references.
2. The identity currently exists in the heap and the field values and type can be merged with the object in the heap to produce a new object with that identity in the heap. In this case the identity is returned as a handle for subsequent object references.
3. The identity currently exists in the heap and the two objects cannot be merged. In this case the machine backtracks.

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

$$\begin{cases} (v_2 : s, \rho, h[v_2 \mapsto (v_1)[fs]], c, f, d) & \text{when } v_2 \notin \mathrm{dom}(h) \\ (v_2 : s, \rho, h[v_2 \mapsto o], c, f, d) & \text{when } o = (v_1)[fs] \sqcup h(v_2) \\ f & \text{otherwise} \end{cases} \tag{B.11}$$

Object field expressions are evaluated separately to produce field values at the head of the stack. the mkobj instruction expects to find a sequence of fields at the head of the stack:

$$(s, \rho, h, (v = e) : c, f, d) \longrightarrow (s, \rho, h, e : \mathrm{mkfield}(v) : c, f, d) \tag{B.12}$$

A field is constructed by associating a value with a field name:

$$(w : s, \rho, h, \mathrm{mkfield}(v) : c, f, d) \longrightarrow ((v = w) : s, \rho, h, c, f, d) \tag{B.13}$$

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:

$$(s, \rho, h, e(es) : c, f, d) \longrightarrow (s, \rho, h, es + [e, \mathrm{apply}(|es|)] + c, f, d) \tag{B.14}$$

If we apply a relation then we 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`:

$$\begin{gathered}((ps,\rho_1,e):(w+s),\rho_2,h,\text{apply}(i):c,f,d)\longrightarrow\\(w,\rho_1,h,ps+[\text{check}(e)],f,(s,e,c,d))\end{gathered}\tag{B.15}$$

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

$$\begin{gathered}((ps,\rho_1,e_1,e_2):(w+s),\rho_2,h,\text{apply}(i):c,f,d)\longrightarrow\\(w,\rho_1,h,ps+[\text{check}(e_1),e_2],f,(s,e,c,d))\end{gathered}\tag{B.16}$$

Check just evaluates the expression:

$$(s,\rho,h,\text{check}(e):c,f,d)\longrightarrow(s,\rho,h,e:\text{check}:c,f,d)\tag{B.17}$$

Check will backtrack if the constraint failed:

$$(v:s,\rho,h,\text{check}:c,f,d)\longrightarrow\begin{cases}(s,\rho,h,c,f,d) & \text{when } v=\text{true}\\ f & \text{otherwise}\end{cases}\tag{B.18}$$

Initial occurrences of variables just bind to values:

$$(x:s,\rho,h,v:c,f,d)\longrightarrow(s,\rho[v\mapsto x],h,c,f,d)\tag{B.19}$$

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

$$(x:s,\rho,h,!v:c,f,d)\longrightarrow\begin{cases}(s,\rho,h,c,f,d) & \text{when } \rho.v=x\\ f & \text{otherwise}\end{cases}\tag{B.20}$$

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:

$$(Q:s,\rho,h,\{ps\mid Qs\}:c,f,d)\longrightarrow((Q,\emptyset):s,\rho,h,\text{select}(ps):\{|Qs\}:c,f,d)\tag{B.21}$$

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

$$((Q,R):s,\rho,h,\text{select}(\emptyset):c,f,d)\longrightarrow(Q\cup R:s,\rho,h,c,f,d)\tag{B.22}$$

If we have run out of possibilities then fail:

$$((\emptyset,\_):s,\rho,h,\text{select}(p\to P):c,f,d)\longrightarrow f\tag{B.23}$$

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

$$\begin{gathered}((v\to V,Q):s,\rho,h,\text{select}(p\to P):c,f,d)\longrightarrow\\(v:(\emptyset,V\cup Q):s,\rho,h,p:\text{select}(P):c,((V,v\to Q):s,e,\text{select}(p\to P):c,f,d)))\end{gathered}\tag{B.24}$$

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

$$(Q : s, \rho, h, \{|Qs\} : c, f, d) \longrightarrow (\mathrm{span}(Q, |Qs|) : s, \rho, h, \mathrm{split}(Qs) : c, f, d) \tag{B.25}$$

$$([] : s, \rho, h, \mathrm{split}(Qs) : c, f, d) \longrightarrow f \tag{B.26}$$

$$\begin{gathered}((Ps : Pss) : s, \rho, h, \mathrm{split}(Qs) : s, f, d) \longrightarrow \\ (Ps + s, \rho, Qs + c, (Pss : s, \rho, h, \mathrm{split}(Qs) : s, f, d), d)\end{gathered} \tag{B.27}$$

An object pattern is matched against an object identity that references an object in the heap. The object in the heap is matched against the pattern:

$$\frac{h(v_1) = (v_1, v_2)[S]}{(v_1 : s, \rho, h, (p_1, p_2)[F] : c, f, d) \longrightarrow (v_1 : v_2 : S : s, \rho, h, p_1 : p_2 : F : c, f, d)} \tag{B.28}$$

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

$$(S : s, \rho, h, \emptyset : c, f, d) \longrightarrow (s, \rho, h, c, f, d) \tag{B.29}$$

Select the matching field:

$$\begin{gathered}((S_1 \cup \{v = x\} \cup S_2) : s, \rho, h, (F_1 \cup \{v = p\} \cup F_2) : c, f, d) \longrightarrow \\ (x : (S_1 \cup S_2) : s, \rho, h, p : (F_1 \cup F_2) : c, f, d)\end{gathered} \tag{B.30}$$

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

$$(v : \_, h, \_, [], f, (s, \rho, c, d)) \longmapsto (v : s, \rho, h, c, f, d)) \tag{B.31}$$

Evaluation of a let expression creates a new binding contour and matches the patterns against the values before evaluating the let-body:

$$(s, \rho, h, \mathrm{let}\; ps = es \;\mathrm{in}\; e : c, f, d) \longrightarrow ([], \rho, h, \mathrm{rev}(es)ps + [e], f, (s, \rho, c, f, d)) \tag{B.32}$$

A where expression is just a let expression written differently:

$$(s, \rho, h, e \;\mathrm{where}\; ps = es : c, f, d) \longrightarrow (s, \rho, h, \mathrm{let}\; ps = es \;\mathrm{in}\; e : c, f, d) \tag{B.33}$$

Recursive definitions are limited to recursive transformations:

$$\frac{(s, \rho[vs \mapsto xs], h, es + c, f, d) \longrightarrow (xs + s, \rho[vs \mapsto xs], h', c, f', d)}{(s, \rho, h, \mathrm{letrec}\; vs = es \;\mathrm{in}\; e : c, f, d) \longrightarrow ([], \rho[vs \mapsto es], h', [e], f', (s, \rho, c, f, d))} \tag{B.34}$$

A whererec expression is just a letrec expression written differently:

$$(s, \rho, h, e \;\mathrm{whererec}\; ps = es : c, f, d) \longrightarrow (s, \rho, h, \mathrm{letrec}\; ps = es \;\mathrm{in}\; e : c, f, d) \tag{B.35}$$

Composing transformations using $+$ constructs an *or*-transformation that behaves specially when it is applied to some arguments:

$$(s, \rho, h, (e_1 + e_2) : c, f, d) \longmapsto (s, \rho, h, e_1 : e_2 : + : c, f, d) \tag{B.36}$$

The $+$ instruction simply constructs an or transformation:

$$(x_1 : x_2 : s, \rho, h, + : c, f, d) \longrightarrow (or(x_1, x_2) : s, \rho, h, c, f, d) \tag{B.37}$$

Application of an or transformation applies one transformation and leaves the alternative as a choice point:

$$\begin{array}{l}(or(x_1, x_2) : s, \rho, h, \text{apply}(i) : c, f, d) \longrightarrow \\ \qquad (x_1 : s, \rho, h, \text{apply}(i) : c, (x_2 : s, \rho, h, \text{apply}(i) : c, f, d), d)\end{array} \tag{B.38}$$

Composing transformations using $\times$ constructs an *and* transformation that behaves specially when it is applied:

$$(s, \rho, h, (e_1 \times e_2) : c, f, d) \longmapsto (s, \rho, h, e_1 : e_2 : \times : c, f, d) \tag{B.39}$$

The $\times$ instruction simply constructs an and transformation:

$$(x_1 : x_2 : s, \rho, h, \times : c, f, d) \longrightarrow (and(x_1, x_2) : s, \rho, h, c, f, d) \tag{B.40}$$

Application of an and transformation applies both transformations and attempts to merge the results. If the merge cannot take place then the machine backtracks:

$$\begin{array}{l}(and(x_1, x_2) : (vs + s), \rho, h, \text{apply}(i) : c, f, d) \longrightarrow \\ \qquad x_1 : (vs + (x_2 : (vs + s))), \rho, h, \text{apply}(i) : \text{apply}(i) : \text{merge} : c, f, d)\end{array} \tag{B.41}$$

The merge operator takes two values and the heap and is defined when the two values can be merged with respect to the objects in the heap. the result (when defined) is the merged value and a new heap:

$$\begin{array}{l}(x_1 : x_2 : s, \rho, h, \text{merge} : c, f, d) \longrightarrow \\ \qquad \begin{cases} (o : s, \rho, h', c, f, d) & \text{when } (o, h') = \text{merge}(x_1, x_2, h) \\ f & \text{otherwise} \end{cases}\end{array} \tag{B.42}$$

Appendix C.

Glossary

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

Composition (of transformations) Composition of transformations is a form of reuse where transformations are composed together through composition functions such as `conjunct` and `disjunct`.

Domain Transformation are specified between a number of domains. In the superstructure a domain consists of a number of patterns and a condition.

Infrastructure The infrastructure is the 'core kernel' of our proposal. Not intended for end users, it provides a simple semantic core for those who need to have a precise reference point for the proposal.

Mapping A mapping is a potentially directed transformation implementations.

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

Pattern A pattern describes the 'shape' of an object it will match against.

Pattern matching Pattern matching is a process whereby parts of a model are matched against a pattern.

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

Relation Relations are multi-directional declarative transformation specifications.

Superstructure The superstructure is the semantically 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

[ACR$^+$03] Biju K. Appukuttan, Tony Clark, Sreedhar Reddy, Laurence Tratt, and R. Venkatesh. A model driven approach to model transformations. In *MDAFA 2003*, June 2003.

[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`.

[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`.