MOF Query / Views / Transformations

Second Revised Submission

Submitted By:

DSTC
International Business Machines
CBOP

12 January 2004

Table of Contents

1 Overview 9

1.1 Primary Contacts for the QVT submission 9

1.2 Acknowledgments 9

1.3 Structure of This Submission 10

1.4 Resolution of RFP Requirements 10

1.4.1 Mandatory Requirements 10
1.4.2 Optional Requirements 11
1.4.3 Issues to be discussed 12
1.4.4 Evaluation Criteria 14

1.5 Proof of Concept 14

1.6 Changes to other OMG Specifications 14

2 Overall Design Rationale 15

2.1 Relationship between Queries, Views, and Transformations 15

2.2 Requirements 17

2.2.1 Functional Requirements 17
2.2.2 Usability Requirements 18

2.3 Our Overall Approach 19

2.3.1 Styles of Transformation 20

2.4 Example of Transformation 21

2.4.1 Relationship between Transformation Model and EMOF and CMOF 23
2.4.2 Relationship between Transformation Model and OCL 23
2.4.3 The use of Plugin Code for Pattern Matching 23

3 Using the Transformation Model 25

3.1 Example UML, Java, and Tagging models 25

3.2 Notation 26

3.3 Transformation 26

3.4 Transformation Rules 27

3.5 MofTerms and FeatureExprs 27

3.6 Trackings and Correspondences 28

3.7 Extents 29

3.8 Pattern Definitions and Pattern Uses 29

3.9 Transformation Rule Extending and Superseding 30

3.10 Tracking Hierarchies 31

3.11 MofTerm Ordering 31

Table of Contents

3.12 Full example 33

4 Transformation Language Semantics 35

4.1 Introduction 35

4.2 The Model 35

4.2.1 VarScope 35
4.2.1 Attributes 35
4.2.1 Associations 35
4.2.1 Constraints 37
4.2.1 Semantics 37
4.2.2 AbstractVar 37
4.2.2 Attributes 37
4.2.2 Associations 37
4.2.2 Constraints 37
4.2.2 Semantics 37
4.2.3 PatternVar 37
4.2.3 Attributes 37
4.2.3 Associations 37
4.2.3 Constraints 37
4.2.3 Semantics 38
4.2.4 ExtentVar 38
4.2.4 Attributes 38
4.2.4 Associations 38
4.2.4 Constraints 38
4.2.4 Semantics 38
4.2.5 TRuleVar 38
4.2.5 Attributes 38
4.2.5 Associations 38
4.2.5 Constraints 39
4.2.5 Semantics 39
4.2.6 PatternScope 39
4.2.6 Attributes 39
4.2.6 Associations 39
4.2.6 Constraints 40
4.2.6 Semantics 40
4.2.7 PatternDefn 40
4.2.7 Attributes 40
4.2.7 Associations 40
4.2.7 Constraints 40
4.2.7 Semantics 40
4.2.8 Query 40
4.2.8 Attributes 40
4.2.8 Associations 41

4.2.8 Constraints 41
4.2.8 Semantics 41
4.2.9 Transformation....41
4.2.9 Attributes....41
4.2.9 Associations 41
4.2.9 Constraints 42
4.2.9 Semantics 42
4.2.10 MOF::Class (used as a TrackingClass)....42
4.2.10 Associations 42
4.2.11 Key....42
4.2.11 Associations 43
4.2.11 Constraints 43
4.2.11 Semantics 43
4.2.12 TRule....43
4.2.12 Attributes....44
4.2.12 Associations 44
4.2.12 Constraints 44
• Semantics....45
4.2.13 MofTerm 46
4.2.13 Associations 46
4.2.13 Constraints 46
4.2.13 Semantics 47
4.2.14 MofInstance 47
4.2.14 Attributes....47
4.2.14 Associations 47
4.2.14 Constraints 47
4.2.14 Semantics 48
4.2.15 Term....48
4.2.15 Attributes....48
4.2.15 Associations 48
4.2.15 Constraints 48
4.2.15 Semantics 48
4.2.16 CompoundTerm 48
4.2.16 Attributes....49
4.2.16 Associations 49
4.2.16 Constraints 49
4.2.16 Semantics 49
4.2.17 AndTerm....49
4.2.17 Attributes....49
4.2.17 Associations 49
4.2.17 Constraints 50
4.2.17 Semantics 50
4.2.18 OrTerm....50

Table of Contents

4.2.18 Attributes........50
4.2.18 Associations........50
4.2.18 Constraints........50
4.2.18 Semantics........51
4.2.19 NotTerm........51
4.2.19 Attributes........51
4.2.19 Associations........51
4.2.19 Constraints........51
4.2.19 Semantics........51
4.2.20 IfTerm........51
4.2.20 Attributes........52
4.2.20 Associations........52
4.2.20 Constraints........52
4.2.20 Semantics........52
4.2.21 SimpleTerm........52
4.2.21 Attributes........53
4.2.21 Associations........53
4.2.21 Constraints........53
4.2.21 Semantics........53
4.2.22 TrackingUse........53
4.2.22 Attributes........54
4.2.22 Associations........54
4.2.22 Constraints........54
4.2.22 Semantics........55
4.2.23 PatternUse........55
4.2.23 Attributes........55
4.2.23 Associations........55
4.2.23 Constraints........55
4.2.23 Semantics........55
4.2.24 Condition........55
4.2.24 Attributes........55
4.2.24 Associations........55
4.2.24 Constraints........56
4.2.24 Semantics........56
4.2.25 Expression........56
4.2.25 Attributes........56
4.2.25 Associations........56
4.2.25 Constraints........56
4.2.25 Semantics........56
4.2.26 VarUse........56
4.2.26 Attributes........57
4.2.26 Associations........57
4.2.26 Constraints........57

4.2.26 Semantics 57
4.2.27 SimpleExpr 57
4.2.27 Attributes....57
4.2.27 Associations 57
4.2.27 Constraints 57
4.2.27 Semantics 57
4.2.28 StringConstant 57
4.2.28 Attributes....58
4.2.28 Associations 58
4.2.28 Constraints 58
4.2.28 Semantics 58
4.2.29 IntConstant....58
4.2.29 Attributes....58
4.2.29 Associations 58
4.2.29 Constraints 58
4.2.29 Semantics 58
4.2.30 BooleanConstant....58
4.2.30 Attributes....58
4.2.30 Associations 59
4.2.30 Constraints 59
4.2.30 Semantics 59
4.2.31 EnumConstant....59
4.2.31 Attributes....59
4.2.31 Associations 59
4.2.31 Constraints 59
4.2.31 Semantics 59
4.2.32 CompoundExpr 59
4.2.32 Attributes....59
4.2.32 Associations 59
4.2.32 Constraints 60
4.2.32 Semantics 60
4.2.33 CollectionExpr 60
4.2.33 Attributes....60
4.2.33 Associations 60
4.2.33 Constraints 60
4.2.33 Semantics 60
4.2.34 FunctionExpr....60
4.2.34 Attributes....61
4.2.34 Associations 61
4.2.34 Constraints 61
4-3 Semantics....62
4.2.35 FeatureExpr....62
4.2.35 Attributes....63

Table of Contents

4.2.35 Associations 63
4.2.35 Constraints 63
4.2.35 Semantics 63
4.2.36 InstanceRef 63
4.2.36 Attributes 63
4.2.36 Associations 63
4.2.36 Constraints 64
4.2.36 Semantics 64

5 Future Work 65

5.1 Introduction 65
5.2 Multi-directional Transformations 65
5.2.1 What is a direction? 65
5.2.2 How to write rules for multiple directions 66
5.3 Composition 67
5.4 Unique Selection 69
5.5 Use of Tracking Classes by TRules 70

6 Conformance 73

6.1 Query Conformance 73
6.2 Transformation Conformance 73
6.3 View Conformance 74
6.4 Quokka Conformance 74
6.5 Tracking Conformance 74

A: Concrete Syntax Grammar 75

B: Object Relational Mapping Example 81

C: References 87

Overview 1

DSTC, IBM and CBOP are delighted to submit this response to the ADTF's RFP for MOF 2.0 Query / Views / Transformations (QVT). We believe that this RFP addresses a vital element of the realisation of the Model-Driven Architecture (MDA).

We believe that this QVT specification offers three main benefits:

- *Expressive power*: the emphasis is on the declarative specification of the queries and transformations, rather than their implementation.
- *Semantically well-founded*: the queries and transformations can be unambiguously interpreted as they are based on a mathematical logic
- *Fully automatable*: Queries and transformations expressed in our QVT model can be executed by automated means.

1.1 Primary Contacts for the QVT submission

The primary contacts for this QVT submission are:

Keith Duddy and Michael Lawley
Senior Research Scientists
DSTC Pty Ltd
University of Queensland
Brisbane 4072, Australia
Phone: +61 7 3365 4310
Fax: +61 7 3365 4311
Email: pegamento@dstc.edu.au

Sridhar Iyengar
Distinguished Engineer
Application and Integration Middleware
IBM Ltd
Research Triangle Park, NC
U.S.A.
Phone: +1 919 486-1768
EMail: siyengar@us.ibm.com

1.2 Acknowledgments

The submitters wish to acknowledge the contributions of Keith Duddy, Anna Gerber, Sridhar Iyengar, Michael Lawley, Kerry Raymond, and Jim Steel in the preparation of this specification.

1.3 Structure of This Submission

This Chapter contains contact points and explains how the proposal addresses the RFP requirements.

Chapter 2 contains the additional requirements that the submitters had for their design of a MOF query, view and transformation language, and the explains the design rationale for the model of that language.

Chapter 3 provides an example-driven guide to using the main concepts in the transformation language.

Chapter 4 shows the model and contains the semantics of all of its elements.

Chapter 5 describes planned improvements to the transformation language presented in the previous chapters.

Chapter 6 contains conformance requirements that state which parts of the specification must be implemented to be considered conformant.

Appendix A presents an EBNF grammar for the concrete syntax used in the examples throughout this document.

Appendix B includes a full example mapping from a simplified UML model to a relational database model.

Finally, Appendix C contains a bibliography.

1.4 Resolution of RFP Requirements

This section describes how this submission meets the mandatory and optional requirements identified in the RFP.

1.4.1 Mandatory Requirements

The following mandatory requirements are taken from Section 6.5 in the RFP.

1. Language for Querying Models	The model for querying MOF-compliant repositories is a subset of the model for transforming MOF-compliant repositories (see Chapter 4 “Transformation Language Semantics”).
2. Language for transformation definitions.	The model for transformation definitions is given in Chapter 4 “Transformation Language Semantics”
3. Abstact syntax to be defined in MOF 2.0 metamodels.	This submission is based on the MOF 2.0 Core adopted submission. [MOF2Core]

4. Transformation definition can be automated	DSTC's prototype automates the execution of the transformation from information contained in the Transformation Model in Chapter 4 "Transformation Language Semantics"
5. Tranformation definition can create views	Transformations and views are both defined with the Transformation Model. The only difference between a transformation and a view is the underlying implementation. For a transformation, the target extent is independent of source extent; its objects, links and values are implemented by storing them. For a view, the target extent remains dependent on the source extent; its objects, links and values are computed using the source extent. The definition of transformations and views is the same (the specification of source and target models and the relationships between them).
6. Transformation definitions shall be declarative.	Our model for transformation definitions is declarative and was partly inspired by F-logic [KiLaWu95].
7. Mechanisms shall operate using MOF 2.0.	This submission is designed for the transformation of MOF 2.0 Core models [MOF2Core].

1.4.2 *Optional Requirements*

The following optional requirements are taken from Section 6.6 in the RFP.

1. Transformations can execute in two directions.	Our Transformation Model describes declarative relationships between the source and target models. Due to the usability requirements identified in Section 2.2 of Chapter 2, however, the richness of the pattern langauage for identifying elements in the source model(s) means that only a certain class of transformations which eschew some of the language features may be automatically reversible. We do not see any advantage to reducing the expressive power of the pattern language so that all expressible transformations are then reversible.

2. Traceability of transformation executions	Our Transformation Model can embody the identification of traceability relationships between source and target models. The need for these relationships (none, some, all) is determined by the specifier of the transformation. While it is quite possible to trace all elements in a transformation, this will result in a massive amount of traceability information, much of which can be derived from other traceability information in conjunction with the definition of the transformation. These traceability relationships can be used to generate a MOF model to represent the traceability between instances in the source and target extents.
3. Reuse and extension of generic transformations	Our Transformation Model supports the definition of patterns (named queries), the extension of patterns, and the extension and overriding of transformation rules (see Section 2.3 in Chapter 2 “Overall Design Rationale”, and the definition of Pattern and TRule in Chapter 4 “Transformation Language Semantics”).
4. Transactional transformation definitions	As individual transformation rules are not required to produce complete instances in the target extent (that is, instances in the extent model may be created through a number of transformations), the transformation is assumed to be atomic.
5. Additional data as input, and defaults	Transformations can be parameterised with additional data, but there is no support for default values for those parameters in this specification.
6. Same source and target	This is not permitted in this proposal.

1.4.3 Issues to be discussed

The following issues have been taken from Section 6.7 in the RFP.

1. CWM transformation model	Our initial prototype was based on the CWM transformation model, but we found that it lacked expressive power. The Transformation Model in this specification is the result of many refinements to that initial CWM transformation model to meet our design objectives and to reflect our experiences in developing transformations. However, the extent of change is such that most people would be unlikely to recognise the CWM origins.

2. Action Semantics	The UML Action Semantics model contains explicit operations for creating, deleting and reclassifying instances. This is not compatible with our declarative approach to transformation which is based on pattern matching in the source and target extents.
3. Source not well-formed	While it is possible to apply transformations defined by the language described in this document to source models that are not well-formed it is likely that the resulting models will also not be well-formed. Additionally, since a transformation designer generally assumes (and relies on) the well-formedness of the input models, it is quite possible that the result will not be meaningful. Note that well-formedness checking of models will be addressed by responses to the MOF2.0 Facility/Object Lifecycle RFP.
Support for preconditions.	Preconditions are supported by the expression language of the model, and can be included in any transformation rule or pattern definition.
Target not well-formed	There is no guarantee that any given set of rules will result in a well-formed target model. We expect well-formed-ness checking of models will be addressed by responses to the MOF2.0 Facility/Object Lifecycle RFP.
4. Incremental changes to the source and targets	Transformations are deemed to be executed atomically, so the source and target model instances cannot change during that process. Since transformations create target model instances that are independent of the source model instances, subsequently both can be independently changed if desired, but note that this may compromise the traceability between the sources and targets. Due to the declarative nature of the Transformation Model, a model that describes an incremental change to the source can be combined with a transformation to produce a new model that describes the corresponding incremental change required for the target. For views, the target model instances will be dependent on the source model instances, and so changes to the source model instances should be reflected in changes to the target model instances. How this propogation of incremental change is performed is an issue for the implementation of the “view” objects and is outside the scope of this specification. Indeed, it may be significant point for product differentiation.

1.4.4 Evaluation Criteria

The following Evaluation Criteria are taken from Section 6.8 of the RFP.

1. Support for complex transformations	Our transformation supports the matching of arbitrary patterns (Turing complete) over the source and target model instances, so transformations can be as complex as desired. However, being Turing complete is not sufficient for practical purposes. It is also important that complex transformations be capable of being represented as clearly and declaratively as possible. See Section 2.2.2, page 18 for details.
2. Reusable transformations	Our Transformation Model supports the definition of patterns (named queries), the extension of patterns, and the extension and overriding of transformation rules (see Chapter 4 “Transformation Language Semantics”). The final submission will also include support for composition of Transformations.
3. Extendable transformation definitions	Although not contained in this revised submission, our final submission will introduce a concept of a transformation package, which will support import, extension, etc, enabling re-use and composition of existing transformation packages.

1.5 Proof of Concept

DSTC Pty Ltd is currently engaged in a 7 year research programme into Enterprise Distributed Systems Technology with major projects devoted to enterprise modelling and the mapping of such models into middleware technology. DSTC Pty Ltd has extensive experience in the standardisation, implementation and use of MOF and XMI. The DSTC has been developing MOF-based transformation tools since December 2000.

DSTC has developed a prototype, Tarzan, based on the Transformation Model presented in this submission using IBM’s Eclipse Modelling Framework (EMF) as a prototype MOF 2.0 implementation.

1.6 Changes to other OMG Specifications

It is proposed that the CMOF part of MOF 2.0 be extended to include the Key class and its associated semantics. This change would also be invaluable to the traditional database oriented data modeling communities.

Overall Design Rationale 2

"Never send a human to do a machine's job" -- Agent Smith, The Matrix

Our focus is on model-to-model transformations and not with model-to-text transformations. The latter come into play when taking a final PSM model and using it to produce, for example, Java code or SQL statements. We believe that there are sufficient particular requirements and properties of a model-to-text transformation, such as templating and boilerplating, that a specialised technology be used. One such technology is Anti-Yacc [HeRaSt02].

2.1 Relationship between Queries, Views, and Transformations

The RFP (ad/2002-04-10) solicits MOF model(s) for the following:

- mappings between models defined using MOF
- querying instances of MOF models
- creating views of MOF models

We believe that queries, views, and transformations are very closely related. That is, they all operate on a set of Objects and result in a set of Objects. We see the differences as follows:

- A query is evaluated with respect to a set of source Objects defined by an Extent resulting in a set of variable bindings. These are then represented by an Extent referencing a homogenous set of ordered Lists. Each ordered List thus represents one tuple of Elements from the binding. No new Objects can be created explicitly by a query. See Figure 2-1.
- A view is evaluated with respect to a set of source Objects defined by an Extent, and results in an Extent containing a set of new heterogeneous Objects whose properties are defined relative to the View Definition and the set of source Objects. See Figure 2-2.

- A transformation is evaluated with respect to a set of source Objects defined by an Extent and a target Extent, and results in population of the Target extent by a set of newly constructed heterogeneous Objects. See Figure 2-2.

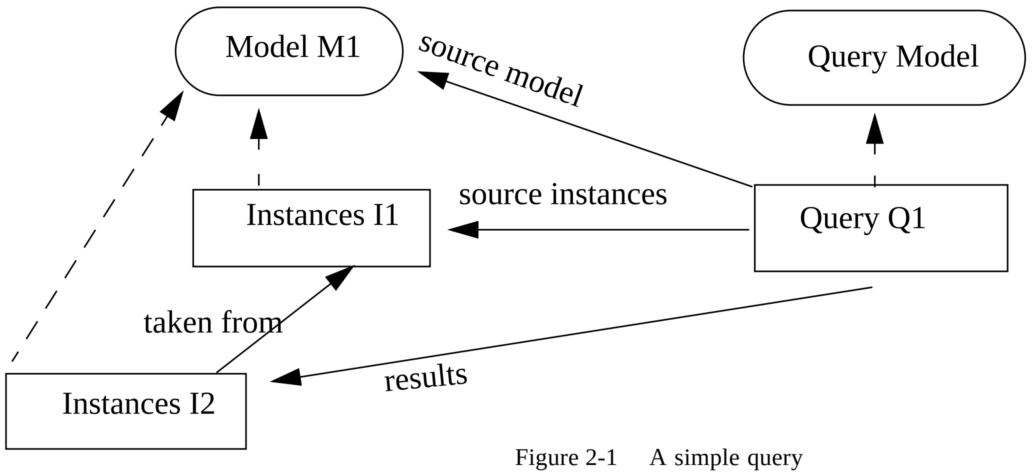

Figure 2-1 A simple query

Intuitively, a query selects existing things and a transformation describes how to construct new things from existing things. In applying a transformation, one may elect to retain the link between the inputs and outputs, resulting in a view that updates as the inputs subsequently change. Alternatively, one may elect not to retain the link, resulting in a new set of standalone objects based on an instantaneous snapshot of the inputs.

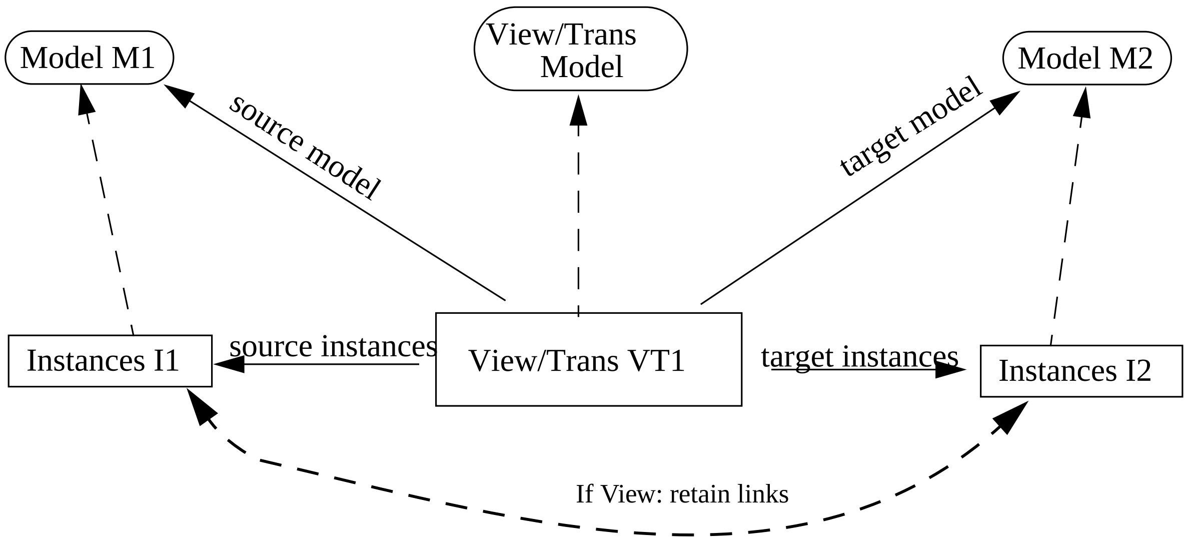

Figure 2-2 A simple view/transformation

Our proposal does not address the problem of transformation-based update. That is, a transformation that describes the new state of an extent in terms of its current state. Until more is known about the outcome of the MOF2.0 Versioning RFP we feel that it would be premature to propose an update semantics.

2.2 Requirements

DSTC has been researching model-based transformations as part of its current 7-year research programme, using a number of different approaches. Our experiments are described in [GeLaRa02]. These experiments and an examination of the experiments of others, have lead us to develop a set of requirements, in addition those given in the RFP, for a transformation language. A language satisfying these requirements will be suitable for describing transformations in a precise but readable manner; the kind of language used to describe model to model mappings needed to realise the MDA vision.

2.2.1 Functional Requirements

The transformation language must be able to:

- Match elements, and ad-hoc tuples of elements, by type (include instances of sub-types) and precise-type (exclude instances of sub-types). For example, the mapping for an EDOC-ECA ExceptionGroup is different to the mapping for its concrete supertype, OutputGroup.
- Filter the set of matched elements or tuples based on associations, attribute values, and other context. For example, an EDOC-ECA Input contained by an InputGroup is mapped differently to an Input contained by an Activity.
- Match both collections of elements and single elements. That is, rules in the language can be expressed in terms of a single element with some implied quantification, rather than needing to explicitly iterate over the elements of a collection.
- Establish associations between source and target model elements. These associations can then be used for maintaining traceability information.
- Specify ordering constraints (of ordered multi-valued attributes or ordered association links), either to match a certain pattern in the source or to establish a certain pattern in the target.
- Define a stable total order over any *unordered* multi-value attributes or *unordered* association links. This may seem counter-intuitive. However, if the same unordered collection of values is involved in a number of transformation rules, it is often important that those rules process the elements in the same order. The actual sequence itself usually does not matter (since semantically the values are unordered), but the sequence should be consistent across all rules. The need for such stable orders typically arises when different mapping rules must be applied to to the “first” and/or “last” element of some collections of values, e.g. constructing a linked list representation of a set of elements requires the “next” element to be

pointed to for all by the “last” element, and the null pointer to be set on the “last” element of the set, even though the order in which the elements occurs is logically irrelevant.

- Handle recursive structure with arbitrary levels of nesting. For example, the uniqueness semantics of the source and target metamodels may differ, thus requiring the construction of fully-qualified names with a global scope in the target model from locally-scoped names in the source model.
- Match and create elements at different meta-levels. For compact and clear specification of such transformations, it is necessary to support dynamic typing in the Transformation Model rather than relying on the explicit use of the reflective features of the MOF meta-model.
- Support both multiple source extents and multiple target extents.

2.2.2 Usability Requirements

It is desirable for readability, maintenance, and expressiveness concerns that:

- When there is no dependency of one rule on another, then the result of the transformation is not dependent on the application order of these rules, and all rules are applied to all source elements.
- Creation of target objects is implicit rather than explicit. This follows from the previous requirement; if there is no explicit total rule application order, then we cannot know which rule creates an object and are relieved of the burden of having to know. Objects are simply created on demand during execution of a transformation.
- Multiple target elements are definable in a single rule.
- A single target element should be definable by multiple rules. That is, different rules can provide property values for the same object.
- Transformation rules need only deal with collections of elements when the semantics of the transformation require it.
- Rules are able to be grouped naturally for readability and modularity.
- Transformation patterns should be definable, thus supporting modular transformation definitions.
- Embedding of conditions and expressions in the Transformation Model is explicit and seamless.
- Optional attributes should be easily handled.
- Transformations can be written in a variety of styles (typically source-driven, target-driven or aspect-driven), see Section 2.3.1.
- Transformations should be composable. This submission does not yet directly address composing Transformations but the model has been designed with this goal in mind and we intend to describe composition in the final submission.

- Rules should be useable in multiple directions; multi-directional transformations should not consist of a disjoint set of rules for each direction. This submission describes some aspects of multi-directional transformations in “Multi-directional Transformations” on page -65. However, while simple examples are a straightforward extension to directional rules, in practice there is much additional complexity that needs to be dealt with. Additionally, there appears to be only a little intertest in such a feature.

2.3 Our Overall Approach

To satisfy the requirements of the RFP and those identified above, we have developed a transformation language that allows for the declarative specification of transformations without regard for rule application order. An early version of this language has been successfully prototyped based on a modified F-Logic interpreter [KiLaWu95]. Our direct implementation, Tarzan, of the current language described here is near completion.

A declarative transformation describes what the result should be in terms of the input, but does not prescribe how to go about constructing the result. However, like Horn clauses in logic programming, instances of a transformation language should be a declarative specification, and also have an equivalent procedural interpretation, thus allowing the specification to be executed. Additionally, declarative transformations are more easily able to specify equivalences and thus be used to describe transformations that can be performed in two (or more) directions.

A transformation in our language consists of the following major concepts: pattern definitions, transformation rules, and tracking relationships.

Pattern definitions are used to label common structures that may be repeated throughout a transformation.

A pattern definition has a name, a set of parameter variables, a set of local variables, and a term. Parameter variables can also be thought of as formal by-reference parameters. Pattern definitions are used to name a query or pattern-match defined by the term. The result of applying a pattern definition via a pattern use is a collection of bindings for the pattern definition’s parameter variables.

Transformation rules are used to describe the things that should exist in a target extent based on the things that are matched in a source extent. Transformation rules can be extended, allowing for modular and incremental description of transformations. More powerfully, a transformation rule may also supersede another transformation rule. This allows for general case rules to be written, and then special cases dealt with via superseding rules. For example, one might write a naive transformation rule initially, then supersede it with a more sophisticated rule that can only be applied under certain circumstances. Superseding is not only ideal for rule optimization and rule parameterization, but also enhances reusability since general purpose rules can be tailored after-the-fact without having to modify them directly.

Tracking relationships are used to associate target elements with the source elements that lead to its creation. Since a tracking relationship is generally established by several separate rules, they allow other rules to match elements based on the tracking relationship independently of which rules were applied or how a target element was created. This allows one set of rules to define what constitutes a particular relationship, while another set depends only on the existence of the relationship without needing to know how it was defined. This kind of rule decoupling is essential for rule reuse via extending and superseding to be useful.

A very common transformation definition kind is shown in Figure 2-3. In this pattern, two classes and an association between them are transformed into two different classes with a different association between them, but the transformation is essentially structure-preserving. This means that if A1 and B1 are associated by C in the source extent, then A1's corresponding (functionally dependent) element X1 will be associated by Z with B1's corresponding (functionally dependent) element Y1 in the target extent.

Therefore, the transformation between association C and association Z can be written most conveniently in terms of the functional dependencies established by the transformations between classes A and X, and between classes B and Y.

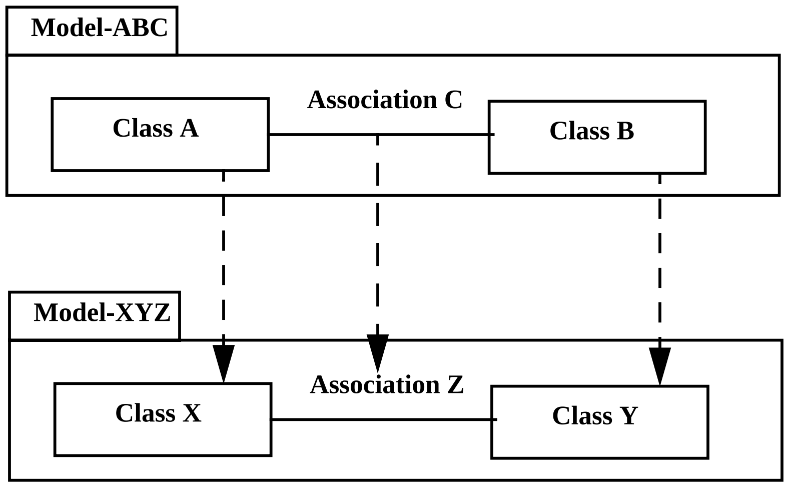

Figure 2-3 A common transformation definition kind.

2.3.1 Styles of Transformation

Our experiences have shown that there are 3 fairly common styles to structuring a large or complex transformation, reflecting the nature of the transformation:

- source-driven, in which each transformation rule is a simple pattern (often selecting a single instance of a class or association link). The matched element(s) are transformed to some larger set of target elements. This style is often used in high-level to low-level transformations (e.g. compilations) and tends to favour a traversal style of transformation specification. This works well when the source instance is tree-like, but is less suited to graph-like sources.

- target-driven, in which each transformation rule is a complex pattern of source elements (involving some highly constrained selection of various classes and association links). The matched elements are transformed to a simple target pattern (often consisting of a single element). This style is often used for reverse-engineering (low-level to high-level) or for performing optimizations (e.g. replacing a large set of very similar elements with a common generic element).
- aspect-driven, in which the transformation rule is not structured around objects and links in either the source or target, but more typically around semantic concepts, e.g. transforming all imperial measurements to metric ones, replacing one naming system with another.

Aspect-driven transformations are a major reason why we favour implicit (rather than explicit) creation of target objects, as aspect-driven transformation rules rarely address entire objects, and thus it is extremely difficult to determine which of several transformation rules (which may or may not apply to any given object) should then have responsibility for creating the target object. Typically the target object is only required if any one of the transformation rules can be applied, but no target object should be created if none of the rules can be applied. This is extremely difficult to express if explicit creation is used.

2.4 Example of Transformation

Figure 2-4 illustrates an example of using transformation to convert between the EDOC Enterprise Collaboration Architecture (ECA) model [EDOC] and a model of workflows used to represent designs in DSTC’s workflow product Breeze [Breeze].

At the top of Figure 2-4, there are 3 models: the EDOC ECA model, the Breeze model, and the Transformation model. The ECA model includes the concepts needed to express some aspects of enterprise systems, such as business processes, expressed in terms of activities and their inputs and outputs and the data flows that connect those inputs and outputs. The ECA model is a technology-independent model. The Breeze model describes the workflows that can be directly implemented using the Breeze workflow product, expressed in terms of tasks, conditional tasks (shown as “If”) and edges that connect tasks. The Breeze model is technology-specific. The Transformation Model includes the concepts needed to describe the transformations between arbitrary MOF models, and is described more fully in Chapter 4 “Transformation Language Semantics”.

Below these models in Figure 2-4 are examples of the instances of the ECA, Breeze, and Transformation models. The ECA specification (an instance of the ECA model) shows an activity with two possible groups of outputs, one of which initiates a second activity. The Breeze specification (an instance of the Breeze model) illustrates a Breeze workflow equivalent to the example ECA specification, showing two tasks, two conditional tasks (shown as “If”) and three edges that control the initiations of tasks. The ECA2Breeze Transformation (an instance of the Transformation Model) describes how to convert from the source ECA model to the target Breeze model. It is important to note that the transformation is described in terms of the models, and not in terms of the instances (the ECA and Breeze specifications). The ECA2Breeze transformation consists of rules, each of which comprises some selection of input elements from the source ECA model and some “selection” of output elements from the target Breeze model and how the source and target elements are related.

The specifics of the relationship between source and target are too detailed to show in Figure 2-4, but are intended to capture the following relationships:

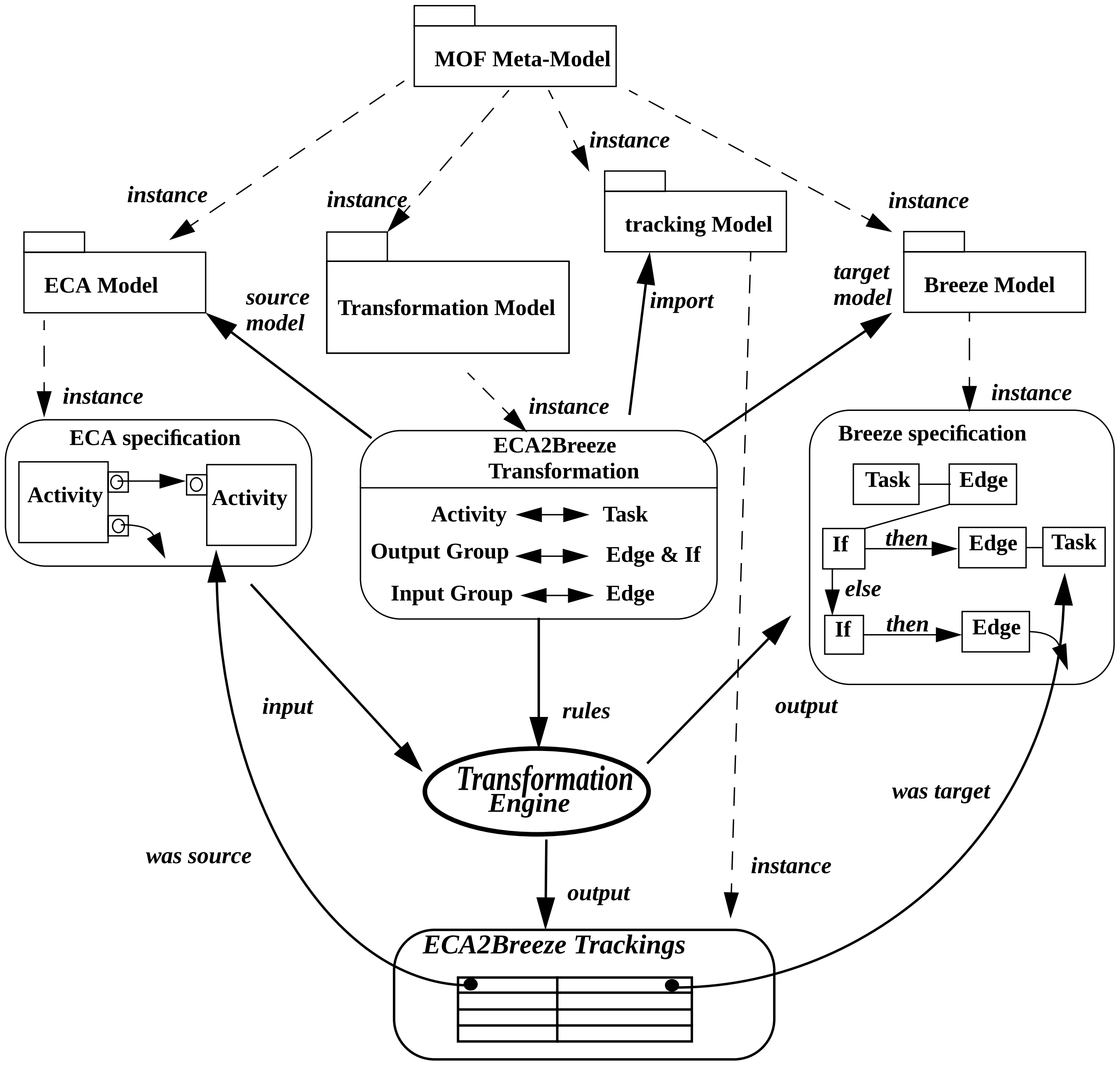

Figure 2-4 Example of transforming EDOC ECA to Breeze

- Each ECA activity corresponds to a Breeze task.
- Each ECA output group corresponds to a Breeze edge to a Breeze condition task, which is used to enable the appropriate subsequent task.
- Each ECA input group corresponds to a Breeze edge used to receive the control signals to initiate the ECA activity's task.

Having described how to transform the ECA model into the Breeze model as the ECA2Breeze Transformation, the ECA2Breeze Transformation rules are then input to a transformation engine, which will populate the target extent based on the source extent according to the transformation rules. Other inputs to the transformation engine may include user-customisation choices (e.g. object granularity, time/space trade-off preferences, preferred optimisations). These inputs may be model elements or other extents (containing instances of some arbitrary parameterization model).

2.4.1 Relationship between Transformation Model and EMOF and CMOF

The Transformation Model may be used to transform instances of both CMOF and EMOF models.

2.4.2 Relationship between Transformation Model and OCL

The Transformation Model includes Term and Expression components which allows the specification of expressions including MOF classes and their properties and associations, as well as arithmetic and logical operators and primitive literals. This overlaps significantly with the OCL metamodel, and we envisage that an adopted OCL 2.0 metamodel may be significantly reused to express the abstract syntax of transformation rules. In addition, we expect the OCL 2.0 specification to reuse the primitive type models and well as other packages from UML 2.0 Infrastructure, giving a common semantic basis for these concepts when used in OCL, and reused in the Transformation Model.

The concrete syntax used in the prototypes developed at DSTC is significantly more compact than OCL, with clearly understandable implicit quantification we consider it better suited to the dynamic typing requirements given in Section 2.2.1. However, we have not proposed any normative concrete syntax for the Transformation Model at this stage, and we envisage that the OCL syntax may be extended to include syntax for the rule and pattern model elements that use the Term model as their basis. We would be in favour of a number of concrete syntaxes, including graphical syntaxes, being available for the specification of transformations. Of course next generation XMI and HUTN syntaxes will be automatically available for use due to the model being MOF compliant.

2.4.3 The use of Plugin Code for Pattern Matching

Although this transformation language defines a powerful pattern matching capability, sometimes it is easier to use a piece of procedural code (maybe because it already exists) to identify the instances that meet a rule's matching criteria. This is facilitated in the model by the Class FeatureExpr. It has the ability to call an operation defined in some class in a model in the scope of the transformation (most likely the Tracking Model), passsing in the value of variables or literals in the scope of the rule as arguments.

Some possible examples include:

- an operation that traverses the model starting at a particular instance, and returns a bag of references to other instances, sorted into a particular order
- an operation that takes two instances references in a model and makes a complex comparison, returning a boolean.
- an operation that implements a complex mathematical or string manipulation function that is not included in the builtins.

All the operations called using this mechanism must be query-only, and may not change the models refered to by the arguments passed to them.

The *collect* attribute of the FeatureExpr Class allows multiivalued results of operation calls to be flattened; that is, each member of the collection is treated as one match of the containing rule.

Using the Transformation Model 3

“You have to see it for yourself” -- Morpheus, The Matrix

This chapter provides a guide to the use of some of the major concepts in the Transformation Model. These concepts are illustrated using a simple transformation from a UML model to a Java model. The models are presented first, followed by examples of the use of transformations, transformation rules, and other concepts from the Transformation Model. The complete example transformation is reproduced in Section 3.12, page 33.

3.1 Example UML, Java, and Tagging models

The illustrative example transformation used in this chapter is taken from a simple mapping from the UML metamodel to the Java metamodel. The relevant parts of the respective models are shown in Figure 3-1 and Figure 3-2. We also make use of a Tagging model, to parameterise the transformation. The Tagging model is shown in Figure 3-3.

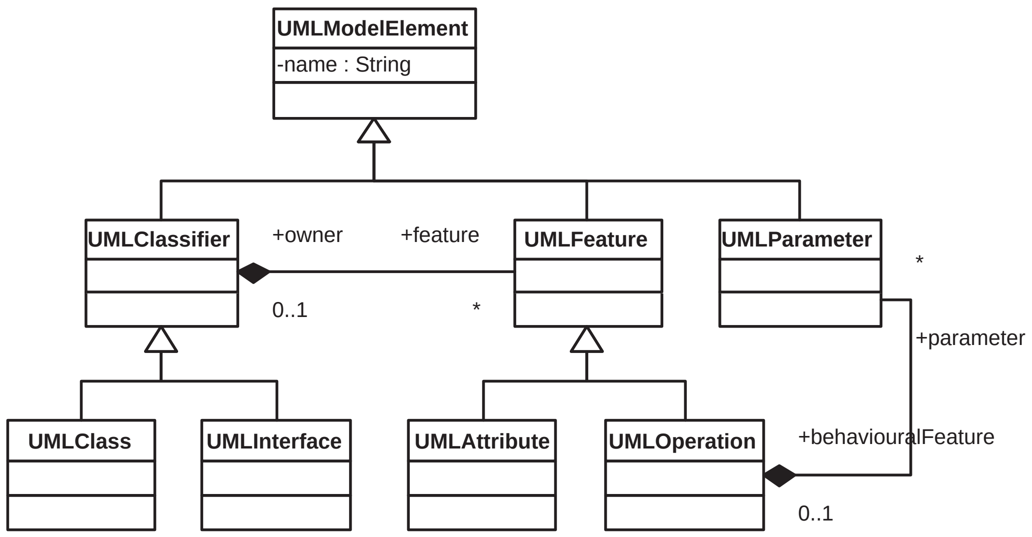

Figure 3-1 UML Model used in the example transformation.

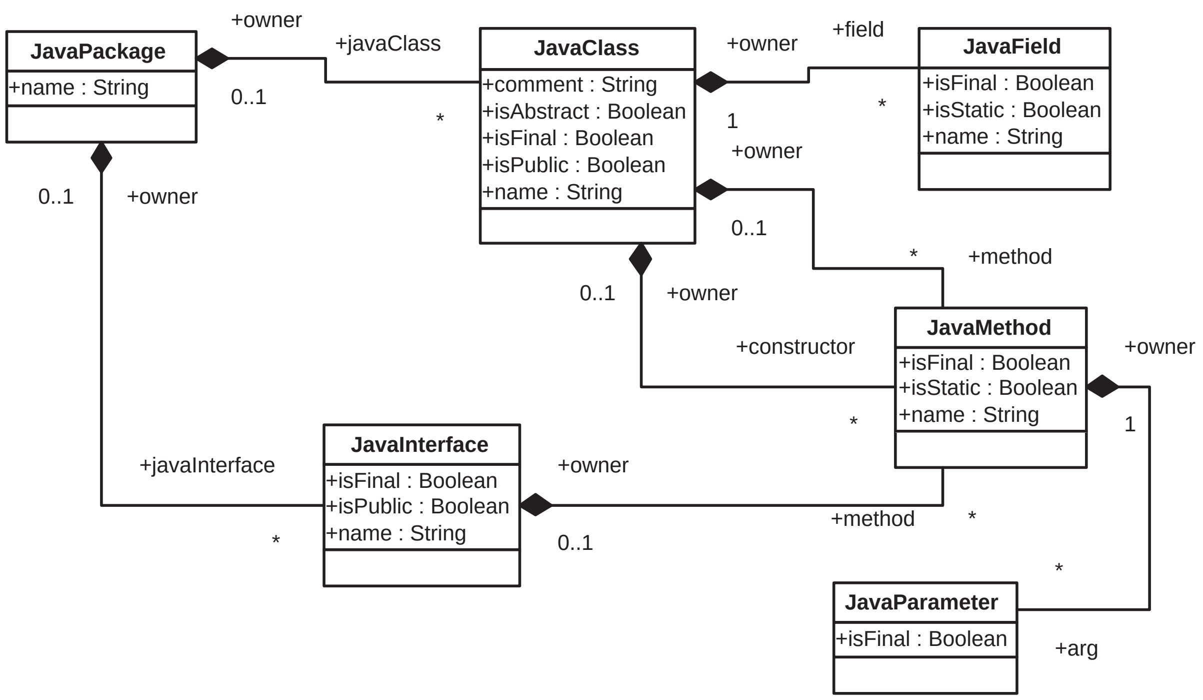

Figure 3-2 Java Model used in the example transformation.

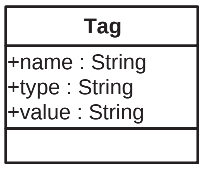

Figure 3-3 Tagging model used in the example transformation.

3.2 Notation

The notation used in this chapter is not normative, and is used only to illustrate the modelling concepts.

3.3 Transformation

The topmost element of the containment tree in the Transformation Model is the Transformation. For our example transformation, we declare a single Transformation named ”Uml2Java”, that will contain all of the rules and patterns used to express our mapping from UML to Java. The transformation declaration is shown in Figure 3-4. “uml”, “java” and “tagging” are ExtentVars representing the extents over which our

Transformation will apply. In our example, the source extents are “uml”, and “tagging” and “java” is the target extent. At most one extent may be labeled as the default source or target extent to avoid needing to use the @extent syntax throught the rules (see Section 3.7, “Extents,” on page -29).

The MOF model to be used to track the Transformation is identified by TRACK USING, and the contents of this section identify the keys of the classes from that tracking model used in within the Transformation. The figure also shows the first transformation rule contained within the Transformation block.

```
TRANSFORMATION Uml2Java(SOURCE uml, TARGET java, tagging)
   TRACK USING TModel{
      KEY OF JavaClassFromUMLClassifier IS umlClassifier;
      ...
   };

   RULE UmlClassifierToJavaClass(uc,jc)
      FORALL UMLClassifier uc
      MAKE JavaClass jc,
         jc.name = uc.name
      LINKING JavaClassFromUMLClassifier jcuc
         WITH jcuc.javaClass = jc, jcuc.umlClassifier = uc;
      ...
```

Figure 3-4 A Transformation declaration containing a TRule

3.4 Transformation Rules

A transformation rule (TRule) represents the basic unit of mapping between an arrangement of source elements and an arrangement of target elements. TRules consist of a Term identifying the sources of the rule and a set of SimpleTerms that identify the targets of the rule.

TRules are used in our example transformation to express mappings from concepts in the UML model to concepts in the Java model. For example, we map each UML Classifier to a Java Class with the same name as the UML Classifier. This mapping is expressed using the TRule “UmlClassifierToJavaClass”, shown in Figure 3-4.

3.5 MofTerms and FeatureExprs

A MofTerm allows us to make statements about elements from the source and target extents. In our example we use MofInstances to make statements about the type of objects from the source and target extents that are involved in the mapping expressed by the TRule. We use FeatureExprs to make statements about the values of the object’s features.

In our TRule “UmlClassifierToJavaClass” shown in Figure 3-4, the FORALL uses a MofInstance to match UMLClassifiers from the source extent, and establish that the variable “uc” is of type UMLClassifier. In the target of the TRule, we use a

MofInstance to establish that variable “jc” is of type JavaClass, and a FeatureExpr to assert that the value of the “name” attribute for that JavaClass should be the value of tthe name of the UMLClassifier “uc”.

3.6 Trackings and Correspondences

A *correspondence* is a statement of functional dependency between target elements and a set of source elements that is characterised in the Transformation Model by tracking Classes and is represented at runtime by instances of the tracking Classes that are conatined by an (implicit) tracking Extent. We use TrackingUses that reference tracking Classes within TRules to establish and query correspondences between source and target elements. A tracking Class is a normal MOF Class that includes an extra feature “Key“ which references the set of attributes of the Class that are identifying attributes.

A TrackingUse in the target of a TRule will assert a correspondence between the source and target elements that are bound to the variables provided, whereas a TrackingUse in the source of a TRule will act as a query on all correspondence assertions, and will bind the variables to the results of the query. TrackingUses are always evaluated with respect to the (implicit) tracking Extent.

In the example we use a TrackingUse referencing “JavaClassFromUMLClassifier” in the target of TRule “UmlClassifierToJavaClass” to establish a correspondence between the UMLClassifiers matched by the rule and the Java Classes to which they are mapped. We establish the correspondence so that we may look up which Java Class was generated from each UML Classifier within other TRules.

We make use of the correspondence established in “UmlClassifierToJavaClass” in the TRule “UmlAttributeToJavaField”, shown in Figure 3-5. The TRule “umlAttributeToJavaField” maps UMLAttributes owned by each UMLClassifier to Java fields belonging to the Java Class mapped from the UMLClassifier. The TrackingUse with repsect to “JavaClassFromUMLClassifier” is used to query the correspondence between UMLClassifiers in the source extent and Java Classes in the target extent, and binds “jc” to the Java Class that was mapped from the UMLClassifier that is the owner of the matched UMLAttribute “ua”. The variable “jc” is then used in the FeatureExpr in the target of the TRule, to provide a value for the “owner” feature of the JavaField “jf” that is mapped from the UMLAttribute “ua”.

```
RULE UmlAttributeToJavaField(ua,jf)
   FORALL UMLAttribute ua
   WHERE JavaClassFromUMLClassifier jcuc AND
      jcuc.umlClassifier = ua.owner AND
      JavaClass jc = jcuc.javaClass
   MAKE JavaField jf,
      jf.owner = jc
   LINKING FieldFromAttr ffa
      WITH ffa.javaField = jf, ffa.umlAttr = ua;
```

Figure 3-5 Using TrackingUse to query correspondences

3.7 Extents

When we declared the “Uml2Java” transformation, as shown in Figure 3-4, we declared ExtentVars “uml”, “java” and “tagging” representing the extents of the transformation. We use ‘@’ in the terms in the source or target of a TRule to associate them with their context extent. We can omit the extent where this information can be determined from the context (when there is only a single source and target extent, or default extents are tagged).

In our example transformation, see Figure 3-6, we wish to comment all JavaClasses resulting from the transformation with a standard copyright statement. We use the Tagging model as the source for this copyright information, using the ‘@’ notation to explicitly specify that the MofTerm that matches instances of Tag to the variable “t” should look in the “tagging” extent.

```
RULE CopyrightToJavaClass
   FORALL Tag@tagging t
   WHERE t.name = ’Copyright’
      AND t.type = ’UMLClassifier’
      AND JavaClassFromUMLClassifier jcuc
      AND JavaClass jc = jcuc.javaClass
   MAKE jc.comment = t.value;
```

Figure 3-6 Working with extents

3.8 Pattern Definitions and Pattern Uses

A PatternDefn provides a named pattern that can be used from within source or target clauses of a TRule. Pattern definitions are generally used to simplify and modularise TRule construction, as patterns may then be used by multiple TRules, instead of each TRule duplicating the Terms used to make statements about elements from the source or target extents that are common to TRules across a Transformation.

For our example mapping, we realise that we will probably declare many TRules that refer to a UMLClassifier and its name, and also to a JavaClass and its name, so we create PatternDefns, as shown in Figure 3-7 to represent these commonly used FeatureExprs. The TRule “UmlClassifierToJavaClass” can thus be rewritten to refer to these PatternDefns, using a PatternUse.

```
DEFINE PATTERN UmlClassifierAndName(uc,name)
   FORALL UMLClassifier uc
   WHERE uc.name = name;

DEFINE PATTERN JavaClassAndName(jc,name)
   MAKE JavaClass jc AND
      jc.name = name;

RULE UmlClassifierToJavaClass(uc,jc)
   FORALL UMLClassifierAndName(uc, n)
   MAKE JavaClassAndName(jc, n)
   LINKING JavaClassFromUMLClassifier jcuc
      WITH jcuc.javaClass = jc, jcuc.umlClassifier = uc;
```

Figure 3-7 Example of PatternDefn and PatternUse

3.9 Transformation Rule Extending and Superseding

TRules can extend or supersede other TRules. The extends association means that the extender rule only applies to those elements the extended rule also applies to, i.e. the source patterns of both rules must match. The supersedes association indicates that the superseded rule applies only to those elements the superseder rules do not apply to. Superseding and extending rules are further linked by associating their variables via the Var::extends and Var::supersedes relationships respectively. This means that elements bound to a Var in the extended rule will be bound to the extending Var in the extending rule.

In our example transformation, a UMLClassifier maps to a Java Class, as expressed by TRule “UmlClassifierToJavaClass”. However, when the UMLClassifier is a UMLInterface, we need to map to a Java Interface rather than a Class.

We use TRule superseding to override the general “UmlClassifierToJavaClass” rule with a rule specific to UMLInterfaces; “UmlInterfaceToJavaInterface”, as shown in Figure 3-8.

TRule extending is used to extend the rule for UMLClassifiers to make the mapping more specific for UMLClasses, so that a constructor method is created on the Java Class in the target extent. TRule “UmlClassToJavaClass” extends “UmlClassifierToJavaClass” as shown in Figure 3-8.

Typically, extension and superseding are used in the presence of inheritance hierarchies in the source or target models.

3.10 Tracking Hierarchies

Since trackings are represented by MOF Classes, they can be arranged in hierarchies. Subtyping a tracking Class means that when a correspondence is recorded in the child tracking Class, it will also be an instance (and queryable) of the parent tracking Class. Tracking Class subtyping is often (but not always) useful in combination with transformation rule extending and superseding, or when dealing with transforming to or from inheritance hierarchies, where it is useful to talk about objects polymorphically.

The example in Figure 3-8 uses the “JavaIntfFromUMLIntf” tracking Class, which subtypes the “JavaClassFromUMLClassifer” tracking Class, so that we may query correspondences between UMLClassifiers and corresponding objects in the target extent generally, using “JavaClassFromUMLClassifier”, or more specifically, only query correspondences between UMLInterfaces and Java Interfaces, using “JavaIntfFromUMLIntf”.

```
RULE UmlInterfaceToJavaInterface(uc,jc)
   SUPERSEDES umlClassifierToJavaClass(uc,jc)
   FORALL UMLInterface uc
   MAKE JavaInterface jc,
      jc.name = uc.name
   LINKING JavaIntfFromUMLIntf jiui
      WITH jiui.javaIntf = jc, jiui.umlIntf = uc;

RULE UmlClassToJavaClass(uc,jc)
   EXTENDS UmlClassifierToJavaClass(uc,jc)
   MAKE JavaMethod m,
      m.name = uc.name,
      jc.constructor = m
   LINKING JavaConsFromUMLClass consFromClass
      WITH
         consFromClass.constructor = m,
         consFromClass.umlClass = uc;
```

Figure 3-8 Example of TRule extending and superseding.

3.11 MofTerm Ordering

It is often necessary to query and assert the order of elements involved in an ordered Property or Association. This is done using the MofOrder element. Using this element allows the modeler to establish partial orders between pairs of elements in the collection, and to thus define a total order over the entire collection of elements.

In our example transformation, we wish to maintain the order of parameters of UMLOperations in the corresponding JavaMethod. Figure 3-9 presents TRules for transforming the ordered set of UML Parameters of each UMLOperation into an ordered set of Java Parameters of the corresponding JavaMethod. The first rule establishes the existence of the corresponding Java Parameters, and the second rule ensures that they are placed in the correct order within the Method. This is done by asserting that a pair of UML Parameters in a given order in the UML Operation (the source order term) have corresponding elements in the same order in the Java Method (the target order term).

```
RULE umlParameterToJavaParameter
   FORALL UMLParameter up
   MAKE JavaParameter jp,
      up.name = jp.name
   LINKING JavaParamFromUMLParam jpup
      WITH jpup.javaParam = jp, jpup.umlParam = up;

RULE parameterOrdering
   FORALL UMLOperation uOp, UMLParameter up1,
      UMLParameter up2
   WHERE up1 BEFORE up2 IN uOp.parameter
      AND JavaMethodFromUML jmu
      AND jmu.umlOp = uOp
      AND Method m = jmu.javaMethod
      AND JavaParamFromUMLParam jpup
      AND jpup.umlParam = up1
      AND JavaParameter jp1 = jpup.javaParam
      AND JavaParamFromUMLParam jpup2
      AND jpup2.umlParam = up2
      AND JavaParameter jp2 = jpup.javaParam
   MAKE jp1 BEFORE jp2 IN m.arg;
```

Figure 3-9 Example of MofOrder

3.12 Full example

Assembling the example fragments, we arrive at Figure 3-10, representing the transformation from a simple UML model to a Java model.

```
TRANSFORMATION Uml2Java(uml, java, tagging)
   TRACK USING TModel{
      KEY OF JavaClassFromUMLClassifier IS umlClassifier;
      ...
   };

   RULE UmlClassifierToJavaClass(uc,jc)
      FORALL UMLClassifier uc
      MAKE JavaClass jc,
         jc.name = uc.name
      LINKING JavaClassFromUMLClassifier jcuc
         WITH jcuc.javaClass = jc, jcuc.umlClassifier = uc;

   RULE UmlAttributeToJavaField(ua,jf)
      FORALL UMLAttribute ua
      WHERE JavaClassFromUMLClassifier jcuc AND
         jcuc.umlClassifier = ua.owner AND
         JavaClass jc = jcuc.javaClass
      MAKE JavaField jf,
         jf.owner = jc
      LINKING FieldFromAttr ffa
         WITH ffa.javaField = jf, ffa.umlAttr = ua;

   RULE CopyrightToJavaClass
      FORALL Tag@tagging t
      WHERE t.name = ’Copyright’
         AND t.type = ’UMLClassifier’
         AND JavaClassFromUMLClassifier jcuc
         AND JavaClass jc = jcuc.javaClass
      MAKE jc.comment = t.value;

   DEFINE PATTERN UmlClassifierAndName(uc,name)
      FORALL UMLClassifier uc
      WHERE uc.name = name;

   DEFINE PATTERN JavaClassAndName(jc,name)
      MAKE JavaClass jc AND
         jc.name = name;

   RULE UmlClassifierToJavaClass(uc,jc)
      FORALL UMLClassifierAndName(uc, n)
      MAKE JavaClassAndName(jc, n)
      LINKING JavaClassFromUMLClassifier jcuc
         WITH jcuc.javaClass = jc, jcuc.umlClassifier = uc;

   RULE UmlInterfaceToJavaInterface(uc,jc)
```

```
      SUPERSEDES umlClassifierToJavaClass(uc,jc)
      FORALL UMLInterface uc
      MAKE JavaInterface jc,
         jc.name = uc.name
      LINKING JavaIntfFromUMLIntf jiui
         WITH jiui.javaIntf = jc, jiui.umlIntf = uc;

   RULE UmlClassToJavaClass(uc,jc)
      EXTENDS UmlClassifierToJavaClass(uc,jc)
      MAKE JavaMethod m,
         m.name = uc.name,
         jc.constructor = m
      LINKING JavaConsFromUMLClass consFromClass
         WITH
            consFromClass.constructor = m,
            consFromClass.umlClass = uc;

   RULE umlParameterToJavaParameter
      FORALL UMLParameter up
      MAKE JavaParameter jp,
         up.name = jp.name
      LINKING JavaParamFromUMLParam jpup
         WITH jpup.javaParam = jp, jpup.umlParam = up;

   RULE parameterOrdering
      FORALL UMLOperation uOp, UMLParameter up1,
         UMLParameter up2
      WHERE up1 BEFORE up2 IN uOp.parameter
         AND JavaMethodFromUML jmu
         AND jmu.umlOp = uOp
         AND Method m = jmu.javaMethod
         AND JavaParamFromUMLParam jpup
         AND jpup.umlParam = up1
         AND JavaParameter jp1 = jpup.javaParam
         AND JavaParamFromUMLParam jpup2
         AND jpup2.umlParam = up2
         AND JavaParameter jp2 = jpup.javaParam
      MAKE jp1 BEFORE jp2 IN m.arg;
}
```

Figure 3-10 Complete transformation example

Transformation Language Semantics 4

4.1 Introduction

This Chapter introduces the abstract syntax for the MOF query, view and transformation language. This language is described by a MOF meta-model.

4.2 The Model

The full model is shown in Figure 4-1 on page 36 without derived associations for the sake of clarity. In the presentation of the model below, we have followed the presentation format used in the MOF2 submission.

4.2.1 VarScope

VarScope is an abstract base type for the language constructs that may declare variables (AbstractVars) that their contained VarUses can reference. An AbstractVar may only be referenced by a VarUse that is contained (directly or indirectly) by the VarScope that contains the AbstractVar.

Attributes

name: string [0..1]	The optional name of this VarScope.

Associations

var: AbstractVar [0..*] {composite, ordered}

The set of owned variables of the scope. This is a set of names that can be referenced by VarUses multiple times within the scope. The opposite association is AbstractVar::scope.

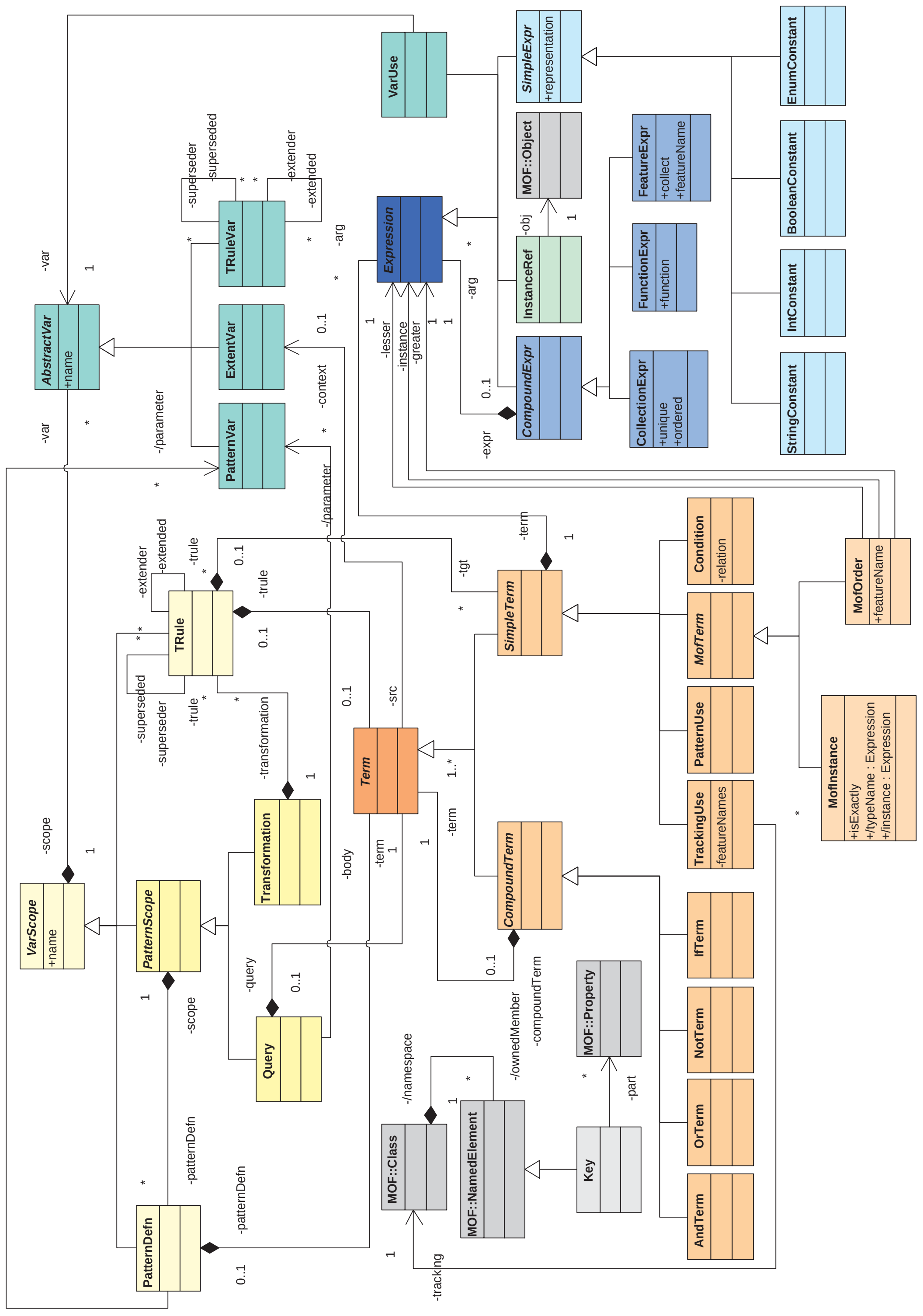

Figure 4-1 Complete Transformation Model

Constraints

Semantics

4.2.2 AbstractVar

AbstractVar is an abstract base type for the variables that may be declared within a VarScope.

Attributes

name: string [1]	The name of the variable.

Associations

scope: VarScope [1]	The scope which owns this variable. The opposite association is VarScope::var.

Constraints

Semantics

4.2.3 PatternVar

PatternVar is the declaration of a variable within a PatternDefn. PatternVar inherits from AbstractVar.

Attributes

name: string [1] (from AbstractVar)

Associations

scope: VarScope [1] (from AbstractVar)

Constraints

The scope of a PatternVar must be a PaternDefn or a Query.

Semantics

4.2.4 ExtentVar

ExtentVar is the declaration of a variable within a Transformation that will be bound to an Extent at runtime.

Attributes

name: string [1] (from AbstractVar)

Associations

scope: VarScope [1] (from AbstractVar)

Constraints

The scope of an ExtentVar must be a Transformation or a Query.

Semantics

4.2.5 TRuleVar

TRuleVar is the declaration of a variable within a TRule. TRuleVar inherits from AbstractVar.

Attributes

name: string [1] (from AbstractVar)

Associations

scope: VarScope [1] (from AbstractVar)

extended: TRuleVar [0..*]	The extended TRuleVars are the variables in the extended rules which are bound to the same value as this variable during transformation execution. The opposite association is TRuleVar::extender.
extender: TRuleVar [0..*]	The extender TRuleVars are the variables in the extender rules which are bound to the same value as this variable during transformation execution. The opposite association is TRuleVar:extended.

superseded: TRuleVar [0..*]	The superseded TRuleVars are the variables in the superseded rules which are bound to the same value as this variable during transformation execution. The opposite association is TRuleVar::superseder.
superseder: TRuleVar [0..*]	The superseder TRuleVars are the variables in the superseder rules which are bound to the same value as this variable during transformation execution. The opposite association is TRuleVar::superseded.

Constraints

The extends relationship can only exist between two variable declarations in different TRule scopes which also have an extender/extended relationship.

The supersedes relationship can only exist between two variable declarations in different TRules which are also in a superseder/superseded relationship.

The scope of a TRuleVar must be a TRule.

Semantics

4.2.6 PatternScope

PatternScope is an abstract base type for the language constructs that may declare patterns (PatternDefns). A PatternDefn may only be referenced by a PatternUse that is contained (directly or indirectly) by the PatternScope that contains the PatternDefn. PatternScope inherits from VarScope.

Attributes

name: string [0..1] (from VarScope)

Associations

var: AbstractVar [0..*] {composite, ordered} (from VarScope)

patternDefn: PatternDefn [0..*] {composite}

The set of owned pattern definitions. The opposite association is PatternDefn::scope.

Constraints

Semantics

4.2.7 PatternDefn

A PatternDefn is used to name and parameterise a Term in the language which can be reused in queries, transformation rules and other patterns, via a PatternUse, to match values in an extent. The AbstractVars owned by the PatternDefn via the parameter association act as parameters to the PatternDefn, any other AbstractVars owned by the PatternDefn are effectively local variables. PatternDefn inherits from VarScope.

Attributes

name: string [0..1] (from VarScope)

Associations

var: AbstractVar [0..*] {composite, ordered} (from VarScope)

parameter: AbstractVar [0..*] {composite, ordered} (subsets VarScope::var)

	The AbstractVars that act as parameters to the pattern, and for whom bindings will be supplied/provided when referenced by a PatternUse.
body: Term [1] {composite}	The Term which is the definition of the pattern. The opposite association is Term::patternDefn.
scope: PatternScope [1]	The pattern scope that contains this pattern definition. The opposite association is PatternScope::patternDefn.

Constraints

Semantics

4.2.8 Query

A Query is used to name and parameterise a Term which matches values in the source extents. The AbstractVars owned by the Query via the parameter association act as parameters to the Query, any other AbstractVars owned by the Query are effectively local variables. Query inherits from PatternScope (and hence from VarScope).

Attributes

name: string [0..1] (from VarScope)

Associations

var: AbstractVar [0..*] {composite, ordered} (from VarScope)

patternDefn: PatternDefn [0..*] {composite} (from PatternScope)

parameter: PatternVar [0..*] (subsets VarScope::var)

The PatternVars that act as parameters to the query, and for whom bindings will be supplied/provided when the query is invoked.

term: Term [1] {composite} The Term which is the definition of the query. The opposite association is Term::query.

Constraints

Semantics

4.2.9 Transformation

A Transformation consists of variables (ExtentVars) representing source and target extents, transformation rules (TRules), pattern definitions (PatternDefns), and tracking relationships. It is used to match elements in the source extent(s), and establish equivalences with elements in the target extent(s). Transformation inherits from PatternScope (and hence from VarScope).

Attributes

name: string [0..1] (from VarScope)

Associations

var: ExtentVar [0..*] {composite, ordered} (subsets VarScope::var)

The set of ExtentVars that will provide references to the input and output extents at runtime.

patternDefn: PatternDefn [0..*] {composite} (from PatternScope)

trule: TRule [0..*] {composite} The set of owned transformation rules. The opposite association is TRule::transformation.

Constraints

Semantics

When used to perform a transformation, every TRule contained by the transformation is evaluated to establish the equivalences between the source and target extents. The dependencies established by TrackingUses are used to identify target objects that must be created, and the equivalences are used to determine their types and property values.

4.2.10 MOF::Class (used as a TrackingClass)

A TrackingClass is a MOF Class used to specify the relationships between source metamodels and target metamodels. MOF References are used to indiacte the class or classes in the source metamodel that will be used to determine which class or classes in the target metamodel will have instances in the target extent at the end of a transformation execution.

At runtime, instances of TrackingClasses are created with references pointing to related instances in the source and target extents. In some cases it is values of properties of source classes, rather than the identity of their instances that determine which target instances (and their owned properties) must exist after a transformation execution. In these cases TrackingClasses will declare non-class typed attributes to store these source model values along with pointers to target model instances which the rules assert must exist. For example, sometimes a source model has an identifying attribute, such as a unique name, which is used as the basis for creation of target model instances.

TrackingUses are used within rules to look up the instances of TrackingClasses specified in other rules. The Key class associated with a TrackingClass has references to one or more properties owned by the TrackingClass which form a key for the purposes of such a lookup. A TrackingClass may have more than one key. A TrackingUse within a TRule must always supply enough (source model) values to satisfy one of the keys of the TrackingClass, so that the appropriate instances of the the TrackClass may be returned (usually to identify the target model instances created by another rule).

Associations

key: Key (subsets MOF::Class::ownedMember)

The key(s) by which this TrackingClass's instances can be queried in a TrackingUse that refers to this tracking class.

4.2.11 Key

A Key is a MOF NamedElement which is used to identify the set of Properties that uniquely determine the identity of the Key's containing Class.

A Key is used to constrain TrackingUses in TRules. TrackingUses act as a query over all instances of a Tracking Class, whose key(s) show which properties such a query must supply values for in order to discover the instances of interest. A Key must identify at least one property.

Associations

part: MOF::Property [1..*] {ordered}

Each Property forms part of the composite Key of the ownging Class.

Constraints

No two instances of the Class (or any subtype) that owns the Key may have the same values for each of the Properties referenced by the Key.

All Properties that are part of a Key must be owned (directly or by inheritance) by the same Class (TrackingClass).

Semantics

4.2.12 TRule

TRule is a transformation rule and a concrete subtype of VarScope. It owns a source Term, *src*, that is "matched" in the context of the source extents supplied to its containing transformation and, if the match is successful, produces a set of bindings of model elements to the Vars owned by this TRule. It also owns a set of target SimpleTerms (MofTerms, TrackingUses, and PatternUses), *tgt*, describing the model elements and their properties that should result from the application of the transformation.

TRules may *extend* other rules to refine the pattern to be matched (a conjunction of the *src* of this TRule with that of the *extended*), and add to the model elements to be created in the target. Commonly an extending TRule adds extra target model elements (or defines extra properties of target model elements) for a subset of the source model elements matched by the extended rule.

TRules may *supersede* other rules within a transformation. The superseder rule effectively restricts the set of matched elements that the (original) superseded rule is applied to (a negated conjunction of the src of this TRule with that of the superseded). Commonly a superseding TRule is used to refine the semantics of the superseded TRule, dealing with special cases not covered by the original rule.

The essential difference between extending and superseding is that extending augments the new rule's src with the old, extended, rule's src whereas superseding augments the old rule's src with a negation of the new, superseding, rule's src thus *changing* the behaviour of the old rule by reducing the set of elements it applies to.

Attributes

name: string [0..1] (from VarScope)

Associations

var: AbstractVar [0..*] {composite, ordered} (from VarScope)

	The set of TRuleVars introduced by this rule. The opposite association is AbstractVar::scope.
src: Term [0..1] {composite}	Used in the evaluation process to match model elements in the source extent. The src term will usually contain VarUses which cause variables to be bound to values in the source extent. The opposite association is Term::trule.

tgt: SimpleTerm [0..*] {composite}

	A set of SimpleTerms which define the structure of the target model elements that must exist as a result of the transformation. This expression will usually contain VarUses which allow variables bound to values in the source extent to populate the values in the target extent. The opposite association is SimpleTerm:: trule
extended: TRule [0..*]	The TRules that this rule extends. The opposite association is TRule::extender.
extender: TRule [0..*]	The TRules that extend this rule. The opposite association is TRule::extended.
superseded: TRule [0..*]	The TRules that this rule supersedes. The opposite association is TRule::superseder.
superseder: TRule [0..*]	The TRules that supersede this rule. The opposite association is TRule::superseded.
transformation: Transformation	The transformation that contains this rule. The opposite association is Transformation::trule.

Constraints

In order to bind variables to values during evaluation of the TRules in a transformation, we require that they are sufficiently constrained. Informally, this means that every variable defined by a TRule be referenced by at least one VarUse that occurs in a context that constrains the variable to be bound to one (or more) of a finite set of values. For example, the variable “p2” in the following src term is *not* sufficiently constrained:

```
FORALL Person p1
WHERE not p1.parent = p2
```

More formally, every TRuleVar owned by the TRule must occur *positively* in the src Term, or be referenced by the target VarUse of a tgt TrackingUse. A variable is said to occur positively in a term if:

- the term is an AndTerm and it occurs positively in any of the Terms directly owned by the AndTerm, or
- the term is an OrTerm and it occurs positively in all of the terms directly owned by the OrTerm, or
- the term is an IfTerm and it occurs positively in the ifTerm or in both the thenTerm and the elseTerm, or
- the term is a MofTerm and it is referenced by a VarUse owned by the MofTerm, or
- the term is a TrackingUse and it is referenced by its tgt VarUse, or
- the term is a PatternUse and the corresponding var in the PatternDefn occurs positively in the PatternDefn’s term, or
- the term is a Condition and it is referenced by a VarUse in the term’s arg Expression that constrains the variable’s possible bindings to a finite set of values.

These conditions capture the notion of *range-restriction*.

Semantics

To evaluate this rule’s src Term, a match term is constructed that represents this rule’s src Term augmented by Terms accounting for the extends and supersedes relationships.

Loosely, the match term consists of this rule’s src Term “anded” with the match terms of each of the extended Terms and “anded” with the negation of each of the match terms of each of the superseder Terms. However, this definition is not sufficient when this Term both extends and supersedes another Term since it introduces a cyclic dependency.

More precisely, to construct the match term one must first construct the extended src term. The extended src term is an AndTerm containing this rule’s src term and (a copy of) each of the extended’s extended src terms. The match term is an AndTerm containing this rule’s extended src term and, for each superseder rule, a NotTerm containing (a copy of) the superseder’s extended src term. The extends and supersedes relationships between TRuleVars are used to correlate the TRuleVars and VarUses when performing the Term copies to construct the extended src terms and match terms.

4.2.13 MofTerm

MofTerm is an abstract base class for Terms that match MOF model elements. MofTerm inherits from SimpleTerm (and hence from Term).

Associations

context: ExtentVar [0..1] (from Term)	A MofTerm must be evaluated in the context of a particular extent, whose value is held by the ExtentVar.
arg: Expression [0..*] {composite, ordered} (from SimpleTerm)	The arguments to the evaluation of the MofTerm. The opposite association is Expression::simpleTerm.
trule: TRule [0..1] (from Term & SimpleTerm)	The rule that contains this MofTerm (if any). The opposite associations are TRule::src and TRule::tgt.
query: Query [0..1] (from Term)	The query that contains this MofTerm (if any). The opposite association is Query::term.
patternDefn: PatternDefn [0..1] (from Term)	The pattern definition that contains this MofTerm (if any). The opposite association is PatternDefn::body.
compoundTerm: CompoundTerm [0..1] (from Term)	The compoundterm that contains this MofTerm (if any). The opposite association is CompoundTerm::term.

Constraints

If context has no value then there must be a CompoundTerm that contains this MofTerm that has a value for context.

Every MofTerm must be contained by exactly one of a TRule, a Query, a PatternDefn, or a CompoundTerm.

Semantics

4.2.14 MofInstance

A MofInstance term is used to match instances of the specified type in an extent. MofInstance inherits from MofTerm (and hence from SimpleTerm and Term). Note that MofInstance is also used for matching Links in an Association, which are individually considered here as instances of the Association, with their AssociationEnds accessible as features.

Attributes

isExactly: boolean	If true, then instances of subclasses of the type are not matched. If false, then instances of subclasses are matched.

Associations

context: ExtentVar [0..1] (from Term)

trule: TRule [0..1] (from Term & SimpleTerm)

arg: Expression [0..*] {composite, ordered (from SimpleTerm)

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

typeName: Expression [1] (subsets SimpleTerm::arg)

Specifies the name of the MOF Classifier to be matched..

instance: Expression [1] (subsets SimpleTerm::arg)

Specifies the VarUse that references the Var to which the matched instances will be bound.

Constraints

There must be 2 args for a MofInstance, one of which is the typeName and the other the instance.

If context has no value then there must be a Term that contains this MofInstance that has a value for context.

Semantics

4.2.15 Term

Term is the abstract base class for terms in the pattern matching expression language. A term is evaluated in the context of a set of variable bindings for the variables defined by the containing TRule or PatternDefn. A Term either fails to match, or succeeds and results in a, possibly updated, set of bindings.

Attributes

Associations

context: ExtentVar [0..1]	A reference to an ExtentVar contained by the Transformation that is bound to the extent in which the matching of this term will be evaluated.
trule: TRule [0..1]	The rule that contains this term. The opposite association is TRule::src.
query: Query [0..1]	The query that contains this term. The opposite association is Query::term.
patternDefn: PatternDefn [0..1]	The pattern that contains this term. The opposite association is PatternDefn::body.
compoundTerm: CompoundTerm [0..1]	The compound term that contains this term. The opposite association is CompoundTerm::term.

Constraints

Every Term must be contained by exactly one of a TRule, a Query, a PatternDefn, or a CompoundTerm.

Semantics

4.2.16 CompoundTerm

CompoundTerm is a abstract base class for terms that compose other terms. CompoundTerm inherits from Term.

Attributes

Associations

term: Term [1..*] {composite, ordered}

The terms which are composed by this CompoundTerm. The opposite association is Term::compoundTerm.

context: ExtentVar [0..1] (from Term)

trule: TRule [0..1] (from Term)

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

Constraints

Semantics

4.2.17 AndTerm

A Term whose result depends on the successful evaluation of each of the terms referenced by the multi-valued *term* property. The set of variable bindings produced must be non-empty and is the intersection of the variable bindings of each of the contained terms. If the intersection is empty or any of the contained terms fails, then the AndTerm fails, otherwise it succeeds. AndTerm inherits from CompoundTerm (and hence from Term).

Attributes

Associations

term: Term [1..*] {composite, ordered} (from CompoundTerm)

The Terms referred to by the term aggregation are the operands to the conjunction represented by the AndTerm.

context: ExtentVar [0..1] (from Term)

trule: TRule [0..1] (from Term)

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

Constraints

An AndTerm must contain at least 2 terms.

Semantics

4.2.18 OrTerm

A Term whose result depends on the successful evaluation of any of the terms referenced by the multi-valued *term* property. The set of variable bindings produced is the union of the variable bindings of each of the successful contained terms. If none of the contained terms succeeds then the OrTerm fails. OrTerm inherits from CompoundTerm (and hence from Term).

Attributes

Associations

term: Term [1..*] {composite, ordered} (from CompoundTerm)

The Terms referred to by the term aggregation are the operands to the disjunction represented by the OrTerm.

context: ExtentVar [0..1] (from Term)

trule: TRule [0..1] (from Term)

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

Constraints

An OrTerm must contain at least 2 terms.

Semantics

4.2.19 NotTerm

A Term whose result is the negation of the contained Term. All Vars from the containing VarScope that are referenced by VarUses in the contained term must have bindings available. This condition may affect the evaluation order of a containing AndTerm. NotTerm inherits from CompoundTerm (and hence from Term).

Attributes

Associations

term: Term [1] (subsets CompoundTerm::term)

The term to be negated.

context: ExtentVar [0..1] (from Term)

trule: TRule [0..1] (from Term)

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

Constraints

A NotTerm must contain exactly 1 term.

Semantics

4.2.20 IfTerm

An IfTerm is semantically equivalent to an OrTerm containing two AndTerms where one AndTerm contains the ifTerm and the thenTerm, and the other AndTerm contains the negation of the ifTerm and the thenTerm. IfTerm inherits from CompoundTerm (and hence from Term).

Note, there may be variable bindings for which the ifTerm succeeds and others for which if fails in which case both the thenTerm and the elseTerm need to be evaluated.

Attributes

Associations

term: Term [1..*] {composite, ordered} (from CompoundTerm)

The Terms referred to by the term aggregation are the operands to the “if”.

context: ExtentVar [0..1] (from Term)

trule: TRule [0..1] (from Term)

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

ifTerm: Term [1] (subsets term from CompoundTerm)

The term representing the condition we are testing.

thenTerm: Term [1] (subsets term from CompoundTerm)

The term that we match when the condition is true.

elseTerm: Term [1] (subsets term from CompoundTerm)

The term that we match when the condition is false.

Constraints

An IfTerm must contain 3 terms, one of which is the ifTerm, another the thenTerm and another the elseTerm.

Semantics

4.2.21 SimpleTerm

SimpleTerm is an abstract base class for Terms denoting MOF model elements TrackingUses, PatternUses, and boolean valued Expressions. SimpleTerm inherits from Term.

Attributes

Associations

arg: Expression [0..*] {composite, ordered}

An ordered set of expressions that providing literal values or variable references to populate the MOF elements, TrackingUses, PatternUses and Conditions of its subtypes. The opposite association is Expression::simpleTerm.

context: ExtentVar [0..1] (from Term)

trule: TRule [0..1] (from Term)

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

Constraints

Semantics

4.2.22 TrackingUse

A TrackingUse Term occurring in the *tgt* of a TRule identifies a MOF Class (TrackingClass) which will be instantiated for each match that the rule's *src* Term makes. This establishes a functional dependency between the values of the Key Properties of the TrackingClass instance and the identity of model elements that need to be created in the target extent(s) by the transformation (which will become the values of target reference Properties of the same tracking instance). In the simplest case, the TrackingClass has a single Key, which nominates a single Property, which refers to a source extent object, and this determines the existence of an object in the target extent refered to by another Property of the TrackingClass. It is also possible that a Key can be data-valued Property of the Tracking Class, which indicates that the value of this Property is derived from the Property value(s) of some source extent object(s), rather than being based on the identity of a source object. In more complex cases a Key will indentify several reference Properties, bound to some combination of source extent objects and copies of data values from the source extent, and this combination will determine the identity of several target objects, refered to by other Properties of the Tracking instance.

A TrackingUse Term occurring in the body of a PatternDefn or the *src* of a TRule represents a query/match against the tuples that satisfy the functional dependency defined by the referenced Tracking. This effectively establishes an execution order dependency between a rules that directly (and indirectly via PatternUse/PatternDefns) reference a TrackingClass via the *src* association and those that reference the TrackingClass (or a supertype) via their *tgt* association.

TrackingUse inherits from SimpleTerm (and hence from Term).

Attributes

Associations

context: ExtentVar [0..1] (from Term)

Not used for TrackingUse.

arg: Expression [0..*] {composite, ordered} (from SimpleTerm)

The values of the arg Expressions are used/bound when querying/populating the attributes (specified by featureNames) of the tracking Class instances in the tracking Extent.

featureNames: String [1..*] The list of feature names that correspond pair-wise with the supplied arg Expressions.

trule: TRule [0..1] (from Term & SimpleTerm).

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

tgt: VarUse [1] (subsets arg from SimpleTerm)

The target model element that functionally depends on the src model elements.

src: VarUse [0..*] {ordered} (subsets arg from SimpleTerm)

The source model elements that the tgt model element functionally depends on.

tracking: MOF::Class [1] The tracking Class to query/populate.

Constraints

The MOF::Class referenced by the tracking association must have a Key.

Semantics

4.2.23 PatternUse

A PatternUse term results in the evaluation of (a copy of) the body term of the reference PatternDefn. The values of each of the ordered *arg* Expressions are used as bindings for the correspondingly ordered (copies of) the parameter Vars contained by the PatternDefn.

Note, a PatternUse contained by a PatternDefn may reference that PatternDefn. For example, to define the notion of a path as consisting of an edge or, recursively, an edge connected to a path.

Attributes

Associations

context: ExtentVar [0..1] (from Term) Not used for PatternUse.

arg: Expression [0..*] {composite, ordered} (from SimpleTerm)

The arguments to be supplied to the PatternDefn's parameter PatternVars.

trule: TRule [0..1] (from Term & SimpleTerm).

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

defn: PatternDefn [1] The PatternDefn to be evaluated in this context.

Constraints

Semantics

4.2.24 Condition

A Condition evaluates its boolean-valued Expression argument. For example, "X > 5" or "member(Y, YList)". Condition inherits from SimpleTerm (and hence from Term).

Attributes

Associations

context: ExtentVar [0..1] (from Term) Not used for Condition.

arg: Expression [1] {composite} (subsets SimpleTerm::arg)	
	The argument to be evaluated by this Condition.

trule: TRule [0..1] (from Term & SimpleTerm).

query: Query [0..1] (from Term)

patternDefn: PatternDefn [0..1] (from Term)

compoundTerm: CompoundTerm [0..1] (from Term)

Constraints

The arg Expression must be boolean valued.

Semantics

4.2.25 Expression

An argument to a SimpleTerm or CompoundExpr which provides values for the evaluation of the containing SimpleTerm or CompoundExpr.

Attributes

Associations

term: SimpleTerm [0..1]	The simple term that may contain this Expression. The opposite association is SimpleTerm::arg.
expr: CompoundExpr [0..1]	The compound expression that may contain this Expression. The opposite association is CompoundExpr::arg.

Constraints

An Expression must be contained by either a SimpleTerm or a CompoundExpr.

Semantics

4.2.26 VarUse

A VarUse term is the use of a variable in an expression. The value of a VarUse is stored in a binding in the evaluation context. This value may be determined from the result of matching a MofTerm in an extent, or by evaluating certain FunctionExprs and NamedExprs. For example, an equality FunctionExpr between a SimpleExpr and a VarUse of an unbound Var will result in a binding of the Var to the value of the

SimpleExpr whereas testing that the variable is less than 5 will not be able to produce a binding and must be delayed until a binding is available as a result of evaluating some other Term with a reference to the variable.

Attributes

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

var: Var [1]	The declaration of the variable used.

Constraints

Semantics

4.2.27 SimpleExpr

A term providing a literal string, number, boolean or enumerator label.

Attributes

representation: string [1]	The string representation of the value of the concrete subtypes of SimpleExpr.

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

Constraints

Semantics

4.2.28 StringConstant

A literal string value.

Attributes

representation: string [1] (from SimpleExpr)

The value of the StringConstant.

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

Constraints

Semantics

4.2.29 IntConstant

A literal integer value.

Attributes

representation: string [1] (from SimpleExpr)

The string representation of the IntConstant.

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

Constraints

Representation must be as per ANSI C.

Semantics

4.2.30 BooleanConstant

A literal boolean value.

Attributes

representation: string [1] (from SimpleExpr)

The string representation of the BooleanConstant.

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

Constraints

May contain the strings “true” and “false” only.

Semantics

4.2.31 EnumConstant

A literal enum label value.

Attributes

representation: string [1] (from SimpleExpr)

The string representation of the Enum label.

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

Constraints

Semantics

4.2.32 CompoundExpr

A structured literal containing sub-expressions.

Attributes

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

arg: Expression [0..*] {composite, ordered}

An ordered set of expressions via the *arg* association which are argument expressions providing literal values or variables for values to populate the subtypes of CompoundExpr. The opposite association is Expression::expr.

Constraints

Semantics

4.2.33 CollectionExpr

A list of values contained by the arg property that make up a set, bag, list or ordered set, depending on the values of the *unique* and *ordered* properties.

Attributes

unique: boolean	True iff the literal collection defined here is to be a set.
ordered: boolean	True iff the literal collection defined here is to be a ordered.

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

arg: Expression [0..*] {composite, ordered} (from CompoundExpr)

Constraints

If *unique* is true then there may not be any two elements that have identical values in the arg list.

Semantics

4.2.34 FunctionExpr

Represents the invocation of the builtin operation named by *operator.* The arguments to this FunctionExpr will be used as arguments to the named operation. If any of the arguments to this expression are VarUses, then there must be bindings for the referenced Vars in the evaluation context.

Attributes

operator: string The name of the operation to be invoked.

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

arg: Expression [0..*] {composite, ordered} (from CompoundExpr)

Constraints

The value of operator must be one of the set of operations on the available types with a result of type int, collection or string, as described in Table 4-1, Table 4-2 and Table 4-3. Functions with boolean results are represented by Conditions.

Table 4-1 Operators on ints

plus(arg1: int, arg2: int, result: int)	The result is arg1 + arg2.
minus(arg1: int, arg2: int, result: int)	The result is arg1 - arg2.
times(arg1: int, arg2: int, result: int)	The result is arg1 * arg2.
divide(arg1: int, arg2: int, result: int)	The result is arg1 / arg2.
mod(arg1: int, arg2: int, result: int)	The result is arg1 modulo arg2.
exp(arg1:int, arg2: int, result: int)	The result is arg1 ^ arg2.
floor(arg: int, result: int)	The result is floor(arg).
ceiling(arg: int, result: int)	The result is ceiling(arg).
sum(arg*: int, result: int)	The result is the sum of all args.
round(arg: int, result: int)	The result is round(arg).

Table 4-2 describes the operators on collections.

Table 4-2 Operators on collections

union(arg*: collection, result: collection)	The result is the set union of all args.
intersect(arg1: collection, arg2: collection, result: collection)	The result is the set intersection arg1 / arg2.
size(arg: collection, result: int)	The result is the number of elements in arg.

Table 4-2 Operators on collections

head(arg: list, result: Object)	The result is the object at the head of the list arg.
tail(arg: list, result: list)	The result is the tail of arg, ie. arg with the head element removed.
elemAt(arg: list, index: int, result: Object)	The result is the object at the position in arg specified by index.

Table 4-3 describes the operators on strings.

Table 4-3 Operators on strings

concat(arg*: string, result: string)	The result is the string concatenation of all args.
substring-before(arg1: string, arg2: string, result: string)	The result is the string that occurs in arg1 before the substring arg2, or arg1 if arg2 does not contain arg1.
substring-after(arg1: string, arg2: string, result: string)	The result is the string that occurs in arg1 after the substring arg2, or the empty string if arg1 does not contain arg2.
substring(arg1: string, arg2: int, arg3?: int, result: string)	The result is the substring of arg1 from index arg2 to arg3 (or to the end of arg1 if arg3 is not supplied). Strings are indexed from 0.
string-length(arg1: string, result: int)	The result is the number of chars in arg1.
translate(arg1: string, arg2: string, arg3: string, result: string)	The result is equal to arg1 with characters from arg2 replaced by characters in same position from arg3.
toUpper(arg: string, result: string)	The result is arg with all characters in upper case.
toLower(arg: string, result: string)	The result is arg with all characters in lower case.

Semantics

4.2.35 FeatureExpr

Represents the value of a Property, or the invocation of an Operation named by *featureName*. When the named Feature is a Property, the *arg* list will contain a single Expression which must evaluate to an object instance. The value of the FeatureExpr is then the value, if any, of the Property named featureName. When the named Feature is an Operation, the value of the first *arg* will be used as the object on which the name operation will be invoked and the remaining *args* will be used as arguments to the named operation. If any of the arguments to this expression are VarUses, then there must be bindings for the referenced Vars in the evaluation context.

Attributes

collect: boolean	If true and the Feature is multi-valued, then the FeatureExpr evaluates to the appropriate collection. If true and the Feature is single-valued, then the FeatureExpr evaluates to a collection containing the value if there is one, otherwise it evaluates to an empty collection. If false and the Feature is multi-valued, then the FeatureExpr results in a multiple evaluations, one for each of the Feature's values. If false and the Feature is single-valued, then the FeatureExpr evaluates to the value of the Feature if it has one, otherwise evaluation fails.
featureName: string	The name of the feature.

Associations

simpleTerm: SimpleTerm [0..1] (from Expression)

expr: CompoundExpr [0..1] (from Expression)

arg: Expression [0..*] {composite, ordered} (from CompoundExpr)

Constraints

Semantics

4.2.36 InstanceRef

InstanceRef is an object reference to a MOF Object.

Attributes

Associations

obj: MOF::Object[1]	The object being referenced.

Constraints

Semantics

Future Work 5

This section describes planned improvements and extensions to the Transformation model presented in this specification. At the time of writing, these are being refined for presentation to the OMG in a future submission by developing appropriate concrete syntaxes and examples.

5.1 Introduction

The following capabilities will be added to the DSTC Transformation model along with appropriate concrete syntax changes:

- multi-directional transformations,
- composition of transformations,
- unique object selection, and
- tighter coupling between rules and tracking classes.

5.2 Multi-directional Transformations

The ongoing work not included in this submission includes Transformation definitions which may be executed in several directions. To be more precise, we envisage the ability for transformations which relate multiple model instances to define several "directions" in terms of a partitioning of the extents into a set of input extents and a set of output extents.

Note that our goal is to allow transformations to approximate isomorphisms, or statements of equivalence between model instances. Wed do not consider a transformation that consists of a set of "forward" rules and a separate set of "backward" rules to be a bi-directional transformation since it will result in duplication and redundancy in rule specification which make maintenance of the transformation more error-prone.

5.2.1 What is a direction?

An example transformation that might be defined so that it can execute in two directions is the Object to Relational mapping, which has been used as a test case during the QVT submission process.

Another common style of transformation is one in which a PIM is mapped to a PSM, and a parameter model, or marking model is used as additional input to assist certain rules in making “the right decision” as to which of several possible PSM configurations to generate from a given PIM element.

In the general case, a Transformation definition will relate N extents to one another, and when executed M (where M < N) of those extents will be used as sources to the transformation, and the remainder will be used as target extents. These extents will be parameters to the Transformation execution. A “direction” is then a named partition (into sources and targers) of the extent parameters for which the Transformation is valid.

Using our PIM, PSM and Parameter example we could define the direction “PIM to PSM” which has the PIM and Parameter models in two separate source extents, and the PSM model would be stored in another target extent. However, in order to “reverse” the transformation the partition named “PSM to PIM” would have the PSM and Parameter extents as sources, and the PIM extent as a target. One could even imagine a third “direction” called “Deduce Parameters” which has the PIM and PSM extents as sources, and the Parameter extent as a target.

5.2.2 How to write rules for multiple directions

A TRule consists of a set of terms. In a single direction transformation the terms of the rule can be divided into a set of left-hand side terms and a set of right-hand side terms. The left-hand-side terms are then conjoined to form a single term: *lhs*, and the right-hand-side terms are conjoined to form a single term: *rhs*. The rule the acts as an implication: lhs -> rhs. In other words the left-hand side “checks” the term over some set of source model instances, and if true it asserts the truth of the right hand side. This may cause the transformation engine to create some target model instances to establish or “enforce” the truth of the *rhs* assertion.

Rules which assert a simple one to one mapping will be able to be executed in reverse. That is rhs -> lhs (and, given the forward direction, this gives us lhs <-> rhs - or isomorphism).

In the general case, the terms of a rule will be partitioned for each valid direction into three four

- Checked

  Terms which only involve terms which refer to the source model elements for a particular direction.

- Enforced

  Terms which have terms which refer to target model elements for a particular direction, and typically relate them to terms referring to source model elements.

- Default

  Terms which are designed to provide values to a target model extent for a particular direction, but only in cases where there is no current value present (see the example below).

- Ignored

  These terms will be valid in some directions, but for a particular direction they are invalid, and must not be evaluated.

An example showing all of the above can be found in the Object to Relational transformation. When creating a new table from a class, we also create a key for the table with the name of the class. However if a target extent already contains a table which correctly corresponds to the class, but has a key with a different name, then we do not create any key, and don't care what the name of the pre-existing key is.

When creating a class from a pre-existing table, we also don't care whether the table key had the same name as the class or not, only that such as key is owned by the table.

So in order to write a TRule which can execute in the two directions: "ObjToRel" and "RelToObj" we must label its terms according to the direction(s) in which they are checked, enforced, or defaults.

Table 5-1 below represents the terms for a rule to transform classes to tables with keys and vice-versa.

Table 5-1

Term	**Direction: ObjToRel**	**Direction: RelToObj**
table.name = class.name	enforced	enforced
key.owner = table	enforced	checked
key.name = class.name	default	ignored

However, although there are four possibilities for how a term may be applied in a specific direction, a lot of this information can be derived from information such as the in/out nature of the scopes involved in the term. Also, some combinations of application (over the set of available directions) may not be sensible for real examples, and could be removed as possibilities. As such, we are presently investigating the language constructs necessary to provide the most useful and expressive form for these concepts.

5.3 Composition

The Transformation model currently supports the composition of rules using rule extension and superseding (see Section 4.2.12), but provides no facility for composing Transformations. The need for such a facility is two-fold: as a technique for re-use and single definition of rules, and as a technique for chaining transformations together in a "series" to form a larger transformation. We believe that these two needs can be satisfied using a single import relationship with bindings between the extent variables in the importing and imported transformation. The two motivating cases are illustrated below.

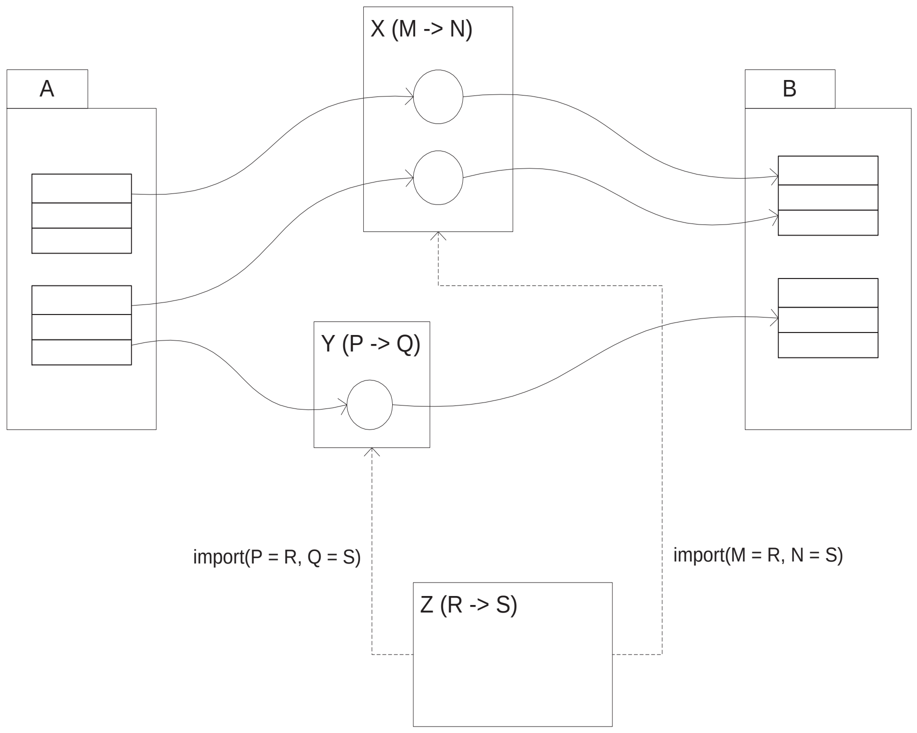

Figure 5-1 Transformation import example

Figure 5-1 presents the case where we have a transformation X from extent M to extent N, and transformation Y from extent P to extent Q. If we wish to have a third transformation Z, from extent R to extent S, in which all of the rules from both X and Y apply, we first import X binding its M extent to Z's R extent and its N to Z's S, and then import Y binding its P extent to Z's R extent, and its Q to Z's S. Transformation Z may then also define its own rules, which may extend or supersede rules from X and/or Y.

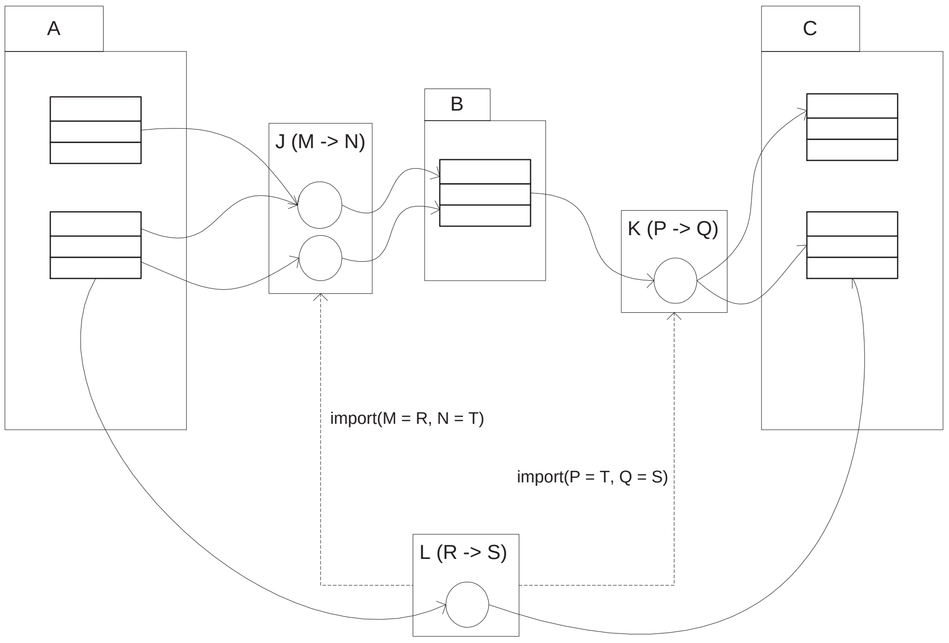

Figure 5-2 Transformation "chaining" import example

Figure 5-2 presents the case where we have a transformation J from model A to model B, and a transformation K from model B to model C, and we wish to compose these to form a transformation L from model A to model C. To do this we first declare a local or transient extent T in transformation L. We then import transformation J, binding its M extent to L's R extent and its N extent to the transient T extent in L. We also import transformation K, binding its P extent to the transient T extent in L, and its Q extent to L's S extent.

The importing of rules from another transformation makes no difference to the way that rules create instances of tracking Classes. All tracking Class instances created, either by imported rules or rules defined directly in the transformation, will exist in the same tracking Extent. Of course, since Extents are not typed, this does not place any restriction on the tracking Classes that may be instantiated by imported rules.

5.4 Unique Selection

When performing certain transformations, the ability to easily group sets of Objects into tuples may be required. In the current Transformation model the ability to match the cross product of two object types is straight forward. Creating a target model instance for each possible combination of two source model instances is straight forward. For example, a source extent or extents which contain instances of X and Y maps to a target extent that contains a Z for each combination of X and Y:

```
RULE R1
   FORALL X x, Y y
   MAKE Z z,
      z.x_attr = x.attr,
      z.y_attr = y.attr
   LINKING SomeTracking st
      WITH st.x = x, st.y = y, st.z = z;
```

It is also easy to restrict the input sets to the cross product by constraining the source model. In our example we might add the following after the FORALL:

```
   WHERE x.attr > 10 AND y.attr > x.attr
```

However, if we required a match of each X only once in combination with a Y, in the current model we would be forced to write some rule which exploits the total ordering of the X elements and Y elements to constrain the set of valid X,Y pairs that cause the creation of the Z elements so that each Y only occurred once for each X by relating the position in the order of X to that of the Y in its order, modulo the size of the set of X.

The next version of this Transformation specification will include concrete syntax for unique selection of elements in ad-hoc tuples, so that transformation authors may leave the ordering and selection “tricks” to the engine. It is likely to take the form of a new keyword “UNIQUE” so that selection of arbitrary tuples can be made with no duplicate elements appearing in some tuple positions.

For example the following rule:

```
RULE R2
   UNIQUE x:X
   FORALL y:Y
   WHERE x.attr > 10 AND y.attr > x.attr
   MAKE z:Z,
      z.x_attr = x.attr,
      z.y_attr = y.attr
   LINKING SomeTracking st
      WITH st.x = x, st.y = y, st.z = z;
```

creates a single Z for each unique X with attr greater than 10, selecting some Y for each X which meets the restriction that the Y’s attr is greater than the X’s attr. The choice of Y is non-deterministic.

5.5 Use of Tracking Classes by TRules

Presently, the relationship between rules and the classes they are instantiating as trackings is somewhat indirect. It is established in the current model via a *tgt* association between a TRule and a TrackingUse that refers to a MOF class via the *tracking* association. This relationship is very important both to the user and to the implementor of a transformation engine, since the source-side and target-side uses of tracking classes is critical in the detection of implicit rule orderings. As such, we are presently investigating the possibility of further formalising the relationship by associating the TRules directly with one or more tracking classes. This would also allow the properties of the tracking class to be automatically brought into scope as variables for manipulation by terms in the rule, and for the tracking to be populated in multiple stages rather than in a single statement.

Also, at present in the language, trackings are established through the TrackingUse class, which in turn uses a VarUse with reference to a Var. Since trackings have now been modified to use MOF Classes and their Properties, the current TrackingUse semantics have a lot in common with MofTerm. Both of these imply the creation of an instance of a MOF Class. The next version of the model will rationalise this part of the model.

Future Work

Conformance 6

There are four alternative levels of conformance defined by this specification:

- Query Conformance
- Transformation Conformance
- View Conformance
- Quokka Conformance

plus an additional optional conformance point (applicable to all of the above except Query):

- Tracking Conformance

6.1 Query Conformance

Query conformance enables elements in source extents to be matched against a Query. Queries include the use of patterns but not tracking relationships. Query conformance requires support for the implementation of the following classes:

- Query
- Var
- PatternDefn
- concrete subtypes of Term (except TrackingUse)
- concrete subtypes of Expression.

6.2 Transformation Conformance

Transformation conformance enables source extents to be transformed into persistent target extents. Transformation includes the use of patterns and the use of tracking within the transformation, but there is no requirement for the tracking relationships to be made persistent after the transformation is complete. Transformation conformance requires the implementation of all concrete classes. Transformation conformance is a superset of Query Conformance.

6.3 View Conformance

View conformance enables source extents to be transformed into derived target extents which require changes in the source extents to be reflected in the derived target extents. Again, patterns and tracking relationships are included in View conformance, but the tracking relationships need not be made persistent. View conformance requires the implementation of all concrete classes, and is a superset of Query conformance.

6.4 Quokka Conformance

Quokka conformance is a superset of Transformation and View conformance, enabling the creation of both persistent and derived target extents. Quokka conformance requires implementation of all concrete classes.

6.5 Tracking Conformance

Transformation conformance, View conformance and Quokka conformance can all have an additional conformance point: tracking conformance. Tracking conformance enables the persistent storage of tracking relationships established during transformation.

Concrete Syntax Grammar A

The following grammar is generated from DSTC’s prototype implementation of a concrete syntax parser for the language used for the examples presented in this document. Please be aware that the grammar as presented here is slightly simplified for the sake of clarity (e.g., the tags SOURCE and TARGET for transformation formals have been omitted).

```
transformation
   :  "TRANSFORMATION" ID formals body
   ;

formals
   :  LBRACK ( vardecls )? RBRACK
   ;

body
   :  (  trackingImport
      |  patternDefn
      |  trule
      )*
   ;

vardecl
   :  ( "SOURCE" | "TARGET" ) ID
   ;

vardecls
   :  vardecl ( COMMA vardecl )*
   ;

trackingImport
   :  "TRACK" "USING" ID
      LCURLY trackbody RCURLY SEMI
   ;

trackbody
   :  ("KEY" “OF” cname "IS"
         feature ( COMMA feature )*
         SEMI
      )*

patternDefn
   :  "PATTERN" pname formals
      ( "FORALL" ranges
         ( "WHERE" conjunct )?
      | "MAKE" targets
      )
      SEMI
   ;
```

```
trule
    :  "RULE" rname formals
       ( relatedRules )*
       "FORALL" ranges
       ( "WHERE" conjunct )?
       "MAKE" targets
       ( "LINKING" trackingUses )?
       SEMI
    ;

tname
    :  ID
    ;

pname
    :  ID
    ;

ranges
    :  range ( COMMA range )*
    ;

conjunct
    :  disjunct ( "AND" disjunct )*
    ;

rname
    :  ID
    ;

relatedRules
    :  ( "extends"
       | "supersedes"
       | "overrides"
       )
       rname formals ( "," rname formals )*
    ;

targets
    :  target ( COMMA target )*
    ;

trackingUses
    :  trackingUse ( COMMA trackingUse )*
    ;

range
    :  mofType vname
    ;
```

```
mofType
   :  ID
   ;

vname
   :  ID
   ;

disjunct
   :  relation ( "OR" relation )*
   ;

relation
   :  LBRACK conjunct RBRACK
   |  NOT relation
   |  link
   |  patternUse
   |  linkOrder
   |  factor ( ASSIGN | RELOP ) factor
   ;

linkOrder
   :  vname "BEFORE" vname "IN" vname
   ;

patternUse
   :  pname actuals
   ;

factor
   :  constant
   |  path
   |  function
   |  vname
   ;

target
   :  link
   |  range
   |  assignment
   ;

assignment
   :  path ASSIGN factor
   ;

trackingUse
   :  "LINKING" tname vname
      "WITH" assignment (COMMA assignment)*
   ;
```

```
factors
   :  factor ( COMMA factor )*
   ;

actuals
   :  LBRACK ( factors )? RBRACK
   ;

constant
   :  STRING
   |  INT
   ;

path
   :  vname ( PERIOD feature )+
   ;

fname
   :  ID
   ;

function
   :  fname actuals
   ;

feature
   :  ID
   ;
```

Object Relational Mapping Example B

The following (incomplete) mapping requirements comes from the QVTPartners initial submission. The Transformation is provided in Figure 6-1 on page 84, while the Tracking model used for the example is provided in Figure 6-2 on page 85.

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

```
TRANSFORMATION Uml2Rel(SOURCE uml, TARGET rel)

  -- Declare the keys of tracking Model
  --
   TRACK USING uml_rel {
      KEY OF TableForClass IS cls;
      KEY OF ColumnForContainedAttr IS cls,attr, name;
      KEY OF KeyForClass IS cls;
      KEY OF ForeignKeyForContainedAttr IS cls, name;
      KEY OF ColumnForAssociation IS assoc;
      KEY OF ForeignKeyForAssociation IS assoc;
   };

  -- Produces a Table and Key
  -- for each persistent Class.
  --
  RULE Class2Table(c, t, k)
   FORALL Class c
   WHERE  c.kind = “persistent”
   MAKE   Table t, Key k,
      t.name = c.name, t.key = k,
      k.name = ‘k_’ + c.name,
   LINKING TableForClass tc
      WITH tc.cls = c AND tc.tbl = t
   LINKING KeyForClass kc
      WITH kc.cls = c, kc.key = k;
```

```
-- Produces Columns belonging to a Table
-- for each persistent Class with storable
-- Attributes.
--
RULE Attr2Column(c, a, t, k, col)
EXTENDS Class2table(c, t, k)
  FORALL Attribute a, String n
  WHERE  hasAttr(c, a, n)
  MAKE   Column col,
    col.name = n, col.owner = t
  LINKING ColumnForContainedAttr cca
    WITH cca.cls = c, cca.attr = a,
       cca.name = n, cca.col = col;

-- Immediate Attributes of a Class that are
-- primary have their corresponding Column
-- as part of the Table’s Key.
--
RULE KeyColumns
EXTENDS Attr2Column(c, a, t, k, col)
  WHERE  a.kind = “primary” AND
    inhAttr(c,a)
  MAKE   col.belongsTo = k;

-- Produces a Column and a ForeignKey for every
-- Association.
--
RULE ForeignKeyForAssoc(assoc)
 FORALL Association assoc
 WHERE  TableForClass tc AND
    tc.class = assoc.source AND Table T = tc.table AND
    KeyForClass kc AND kc.class = assoc.destination
    AND Key k = kc.key
 MAKE   Column col, ForeignKey fk,
    col.name = assoc.name, col.owner = t,
    fk.owner = t, fk.column = col, fk.refersTo = k
 LINKING ColumnForAssociation ca
    WITH ca.assoc = assoc, ca.col = col
 LINKING ForeignKeyForAssociation fka
    WITH fka.assoc = assoc, fka.foriegnKey = fk;
```

```
-- If a Class(C) has a Class(C2)-valued Attribute(A)
-- and C2 has a corresponding table (ie, it’s
-- persistent) then the Column corresponding to A must
-- have a ForeignKey
--
--
-- [Note - this appears not to be done in other’s
-- mapping examples, but is only correct if C2 does not
-- have multiple primary Attributes (otherwise the whole
-- mapping is _much_ more complicated, or surrogate keys
-- need to be introduced).]
--
RULE ForeignKeyColumns(c, a)
  FORALL Class c, Attribute a, Class c2,
  WHERE  hasAttr(c,a,n) AND
    a.type = c2 AND
    TableForClass tc AND tc.class = c AND
    Table t = tc.table AND
    ColumnForContainedAttr cca AND cca.class = c AND
    cca.name = n AND Column col = cca.col AND
    KeyForClass kc AND kc.class = c2 AND Key k = kc.key
  MAKE   ForeignKey fk,
    fk.owner = t, fk.column = Col, FK.refersTo = K
  LINKING ForeignKeyForContainedAttr fkca
    WITH fkca.cls = c, fkca.name = n, fkca.fk = fk;

-- An Attribute is “storable” if it is a simple type
-- (DataType), or it is Class-valued and that Class is
-- persistent.
--
PATTERN storable(a)
  FORALL Attribute a
  WHERE  (a.type = “Class” AND a.type.kind = “persistent”)
    OR (a.type = “DataType”);

-- A Class “inherits” an Attribute if it owns it
-- directly, or it has a superclass that “inherits” the
-- Attribute.
--
-- [Note - other’s mapping examples ignore this, but
-- what is a “class” model without inheritance?]
--
PATTERN inhAttr(c, a)
  FORALL Class c, Attribute a, Class c2
  WHERE a.owner = c
    OR (c.super = c2 AND inhAttr(c2, a));
  -- This pattern is used to handle so-called “complex
  -- types”. A Class “has” and Attribute with a fully
  -- qualified name if it “inherits” the attribute and the
  -- attribute is storable, or it “inherits” another
  -- class-valued attribute for which the class is not
  -- persistent (ie a complex type) and that class “has”
  -- the attribute.
  --
  PATTERN hasAttr(c, a, n)
    FORALL Class c, Attr a
    WHERE (
       inhAttr(c, a) AND
       a.name = n AND
       storable(a)
     )
     OR (
       inhAttr(c, a2) AND
       a2.type = c2 AND
       hasAttr(c2, a, n2) AND
       c2.kind != “persistent” AND
       n = a2.name + ‘_’ + n2
     );
```

Figure 6-1 UML-Rel Transformation

```
package Uml_Rel
  class TableForClass {
    Class cls;
    Table tbl;
  };

  class ColumnForContainedAttr {
    Class cls;
    Attribute attr;
    String name;
    Column col;
  };

  class KeyForClass {
    Class cls;
    Key key;
  };
  class ForeignKeyForContainedAttr {
    Class cls;
    String name;
    ForeignKey fk;
  };

  class ColumnForAssociation {
    Association assoc;
    Column col;
  };

  class ForeignKeyForAssociation {
    Association assoc;
    ForeignKey fk;
  };
};
```

Figure 6-2 Tracking model

References C

[Breeze]	*DSTC, "Breeze: workflow with ease", www.dstc.edu.au/Research/Projects/Pegamento/Breeze/breeze.html*
[CWM]	*Object Management Group, "Common Warehouse Metamodel", 2001, OMG formal/2001-10-01.*
[dMOF]	*DSTC, "dMOF: an OMG Meta-Object Facility Implementation", www.dstc.edu.au/Products/CORBA/MOF/*
[EDOC]	*Object Management Group, "UML Profile for Enterprise Distributed Object Computing (EDOC) Specification", 2002, OMG ptc/02-02-05.*
[HeRaSt02]	*D. Hearnden, K. Raymond and J. Steel, Anti-Yacc: MOF-to-text. In Proceedings, Sixth International Enterprise Distributed Object Computing (EDOC 2002) Conference, pages 200-211. IEEE Computing Society, Los Alamitos, CA, USA 2002.*
[HUTN]	*Object Management Group, "Human-Usable Textual Notation", 2002, OMG ptc/02-12-01.*
[GeLaRa02]	*A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozemberg, editors, Proceedings of ICGT 02, volume 2505 of Lecture Notes in Computer Science, pages 90 -105. Springer Verlag, 2002.*
[JMI]	*Sun Microsystems, "Java^{TM} Metadata Interface (JMI) Specification", 2002, http://java.sun.com/products/jmi/*
[KiLaWu95]	*M. Kifer, G. Lausen, and J.Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Journal of the ACM, 42(4):741 843, July 1995.*
[MDA]	*Object Management Group, "Model Driven Architecture: The Architecture Of Choice for a Changing World", 2001, www.omg.org/mda/*
[MOF]	*Object Management Group, "Meta-Object Facility (MOF^{TM})", 2002, OMG formal/2002-04-03.*
[MOF2]	*Object Management Group, "MOF 2.0 Core RFP", 2001, OMG ad/01-11-14*
[MOF2Core]	*Adaptive, et. al, "Meta Object Facility (MOF) 2.0 Core Proposal", 2003, OMG ad/2003-03-16*

[OCL2]	*Object Management Group, "UML 2.0 OCL RFP", 2000, OMG ad/00-09-03.*
[QVT]	*Object Management Group, "MOF 2.0 Query/View/Transformation RFP", 2002, OMG ad/02-04-10.*
[TokTok]	*DSTC, "TokTok - The Language Generator", www.dstc.edu.au/TokTok*
[UML]	*Object Management Group, "Unified Modeling Language (UML)", 2001, OMG formal/2001-09-67.*
[UML2]	*Object Management Group, "UML 2.0 Infrastructure RFP", 2000, OMG ad/00-09-01.*
[XMI]	*Object Management Group, "XML-Based Model Interchange (XMI) Specification", 2002, OMG formal/2002-01-01.*
[XSLT99]	*XSL Transformations (XSLT) Version 1.0, W3C Proposed Recommendation 8 October 1999. http://www.w3.org/TR/1999/PR-xslt-19991008.*