MOF Query / Views / Transformations

Initial Submission

Submitted By:

DSTC
International Business Machines

3 March 2003

Copyright © 2003 DSTC, IBM Pty Ltd.

The companies listed above hereby grants a royalty-free license to the Object Management Group, Inc. (OMG) for worldwide distribution of this document or any derivative works thereof within OMG and to OMG members for evaluation purposes, so long as the OMG reproduces the copyright notices and the below paragraphs on all distributed copies.

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version.

The copyright holders listed above have agreed that no person shall be deemed to have infringed the copyright, in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

NOTICE: The information contained in this document is subject to change with notice.

The material in this document details a submission to the Object Management Group for evaluation in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification by the submitter.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The Object Management Group and the companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

All Rights Reserved. No part of the work covered by copyright hereon may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner. All copies of this document must include the copyright and other information contained on this page.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to fifty copies) for their internal use as part of the OMG evaluation process.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical, Data and Computer Software Clause at DFARS 252.227.7013

OMG® is a registered trademark of the Object Management Group, Inc.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Overview 1

DSTC and IBM 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.

Conformance requirements in Chapter 5 state which parts of the specification must be implemented to be considered conformant.

Finally, Chapter 6 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.	As the MOF 2.0 RFP is still in progress, this initial submission is based on the MOF 2.0 Core proposal [MOF2Core]
4. Transformation definition can be automated	DSTC’s prototype, MOFLog 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 is based on F-logic [KiLaWu95].
7. Mechanisms shall operate using MOF 2.0.	As far as possible this submission anticipates the features of MOF 2.0, based mainly on the the MOF 2.0 Core proposal [MOF2Core]. Where possible we use terminology and approaches that we anticipate will be in an adopted MOF 2.0 Core. A Revised submission will post-date the adoption of MOF 2.0, and any inconsistencies will be corrected.

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	Garbage in, garbage out. Well-formed-ness 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 directly 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”).
3. Extendable transformation definitions	Although not contained in this initial submission, our revised submission will introduce a concept of a transformation package, which will support inheritance, import, 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 based on the Transformation Model presented in this submission using DSTC’s dMOF product (MOF 1.3) and DSTC’s TokTok product (HUTN 1.0) and DSTC’s MOFLog prototype (based on an F-logic interpreter with MOFlet I/O support).

1.6 Changes to other OMG Specifications

No changes to other OMG specifications are required.

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.

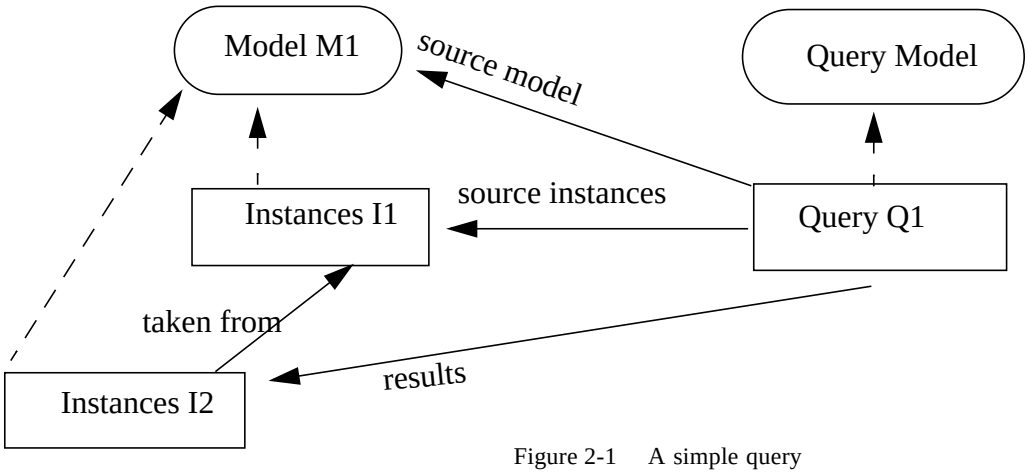

Figure 2-1 A simple query

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.

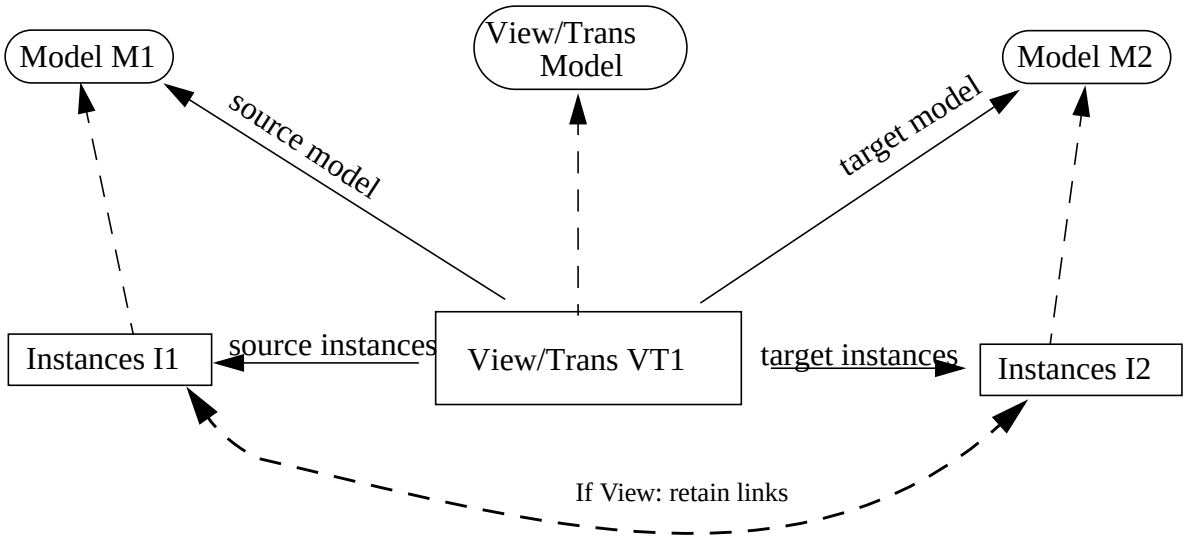

Figure 2-2 A simple view/transformation

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 and expressiveness concerns that:

- There is no dependency on the application order of the 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 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 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.

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. This language has been successfully prototyped based on a modified F-Logic interpreter [KiLaWu95].

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.

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 a target element 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.

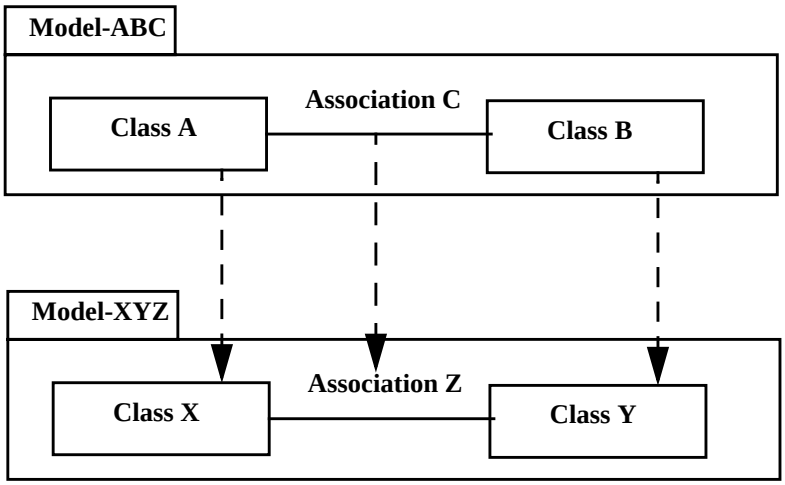

Figure 2-3 A common transformation definition kind.

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:

- 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).

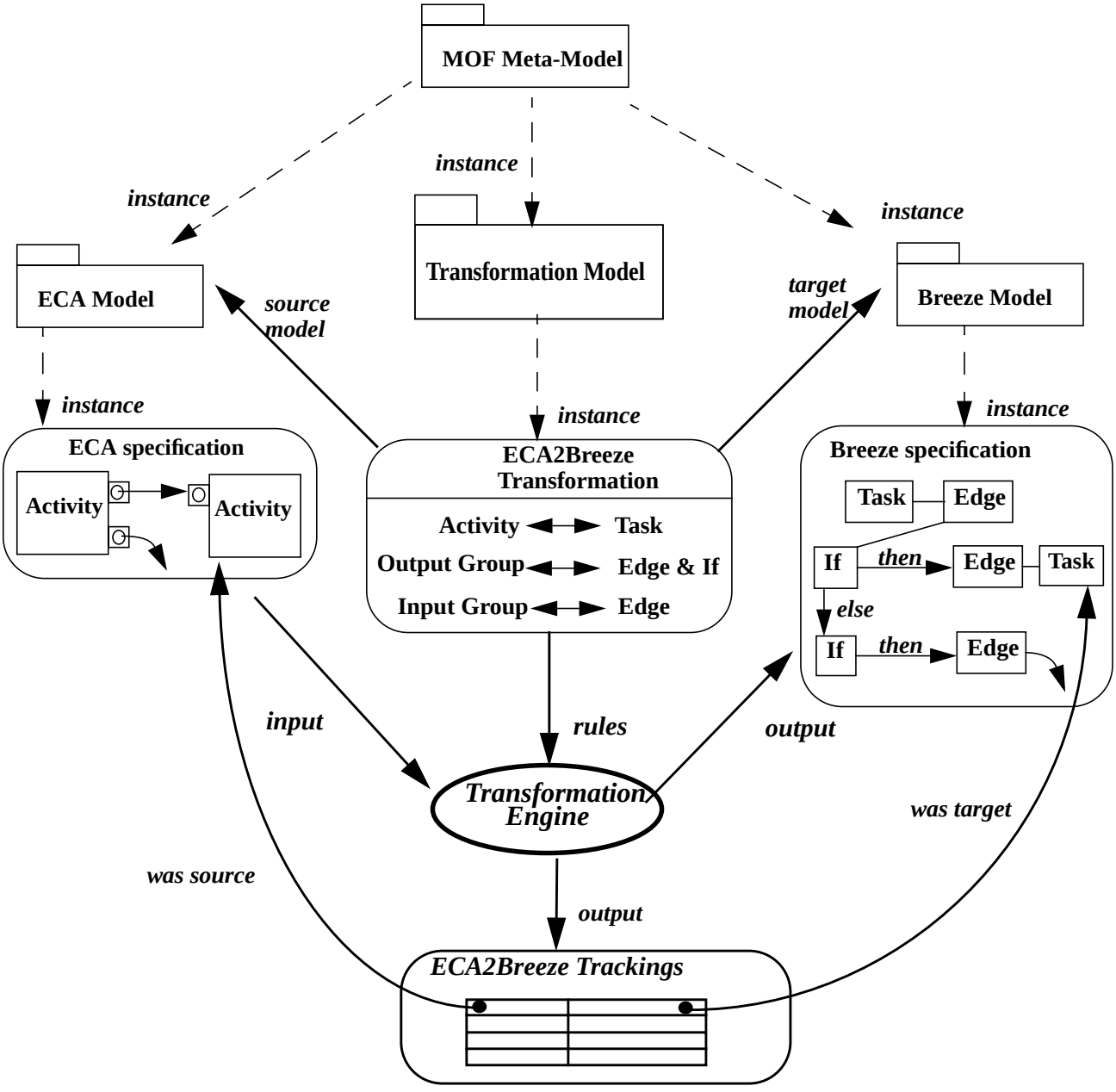

Figure 2-4 Example of transforming EDOC ECA to Breeze

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 specific 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.

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 32.

3.1 Example UML, Java, and Tag 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 Tag model, to parameterise the transformation. The Tag model is shown in Figure 3-3.

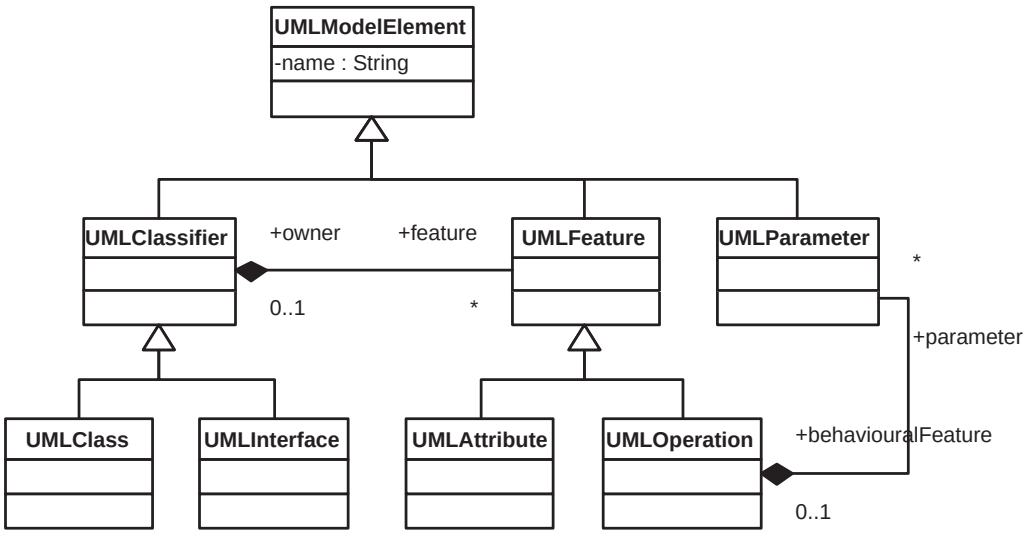

Figure 3-1 UML Model used in the example transformation.

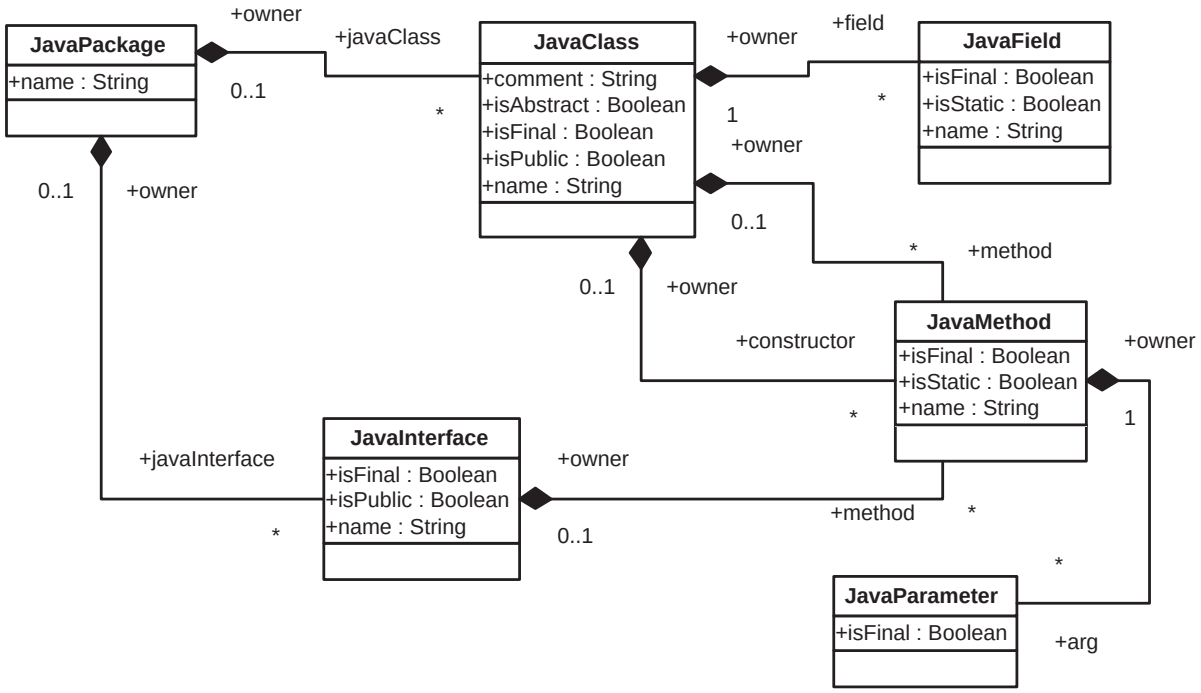

Figure 3-2 Java Model used in the example transformation.

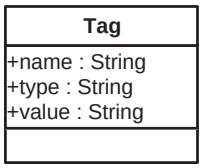

Figure 3-3 Tag model used in the example transformation.

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, patterns, and trackings used to express our mapping from UML to Java. The transformation declaration is shown in Figure 3-4. “Source”, “Target” and “Tags” are Vars representing the extents over which

our Transformation will apply. In our example, the default source extent is “Source”, “Target” is the default target extent and “Tags” represents an additional source extent that we will use to parameterise the transformation. The figure also shows the first transformation rule of transformation, contained within the Transformation block.

```
Transformation uml2java(Source, Target, Tags) {

   TRule umlClassifierToJavaClass(X,Y) {
      X:UMLClassifier[name -> N]
      ->
      Y:JavaClass[name -> N] and
      track(Y, java_class_from_uml_classifier, X)
   }
   ...
}
```

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

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, and MofFeatures to make statements about the values of the objects’ features.

In our TRule “umlClassifierToJavaClass” shown in Figure 3-4, we use a MofInstance to match UMLClassifiers from the source extent, and establish that the variable “X” is of type UMLClassifier. A MofFeature is used to bind the value of the “name” attribute of the UMLClassifier to the variable “N”. In the target of the TRule, we use a MofInstance to establish that variable “Y” is of type JavaClass, and a MofFeature to assert that the value of the “name” attribute for that JavaClass should be the value of the variable “N”.

3.6 Trackings and Correspondences

A *correspondence* is a statement of functional dependency between a target element and a set of source elements that is characterised in the Transformation Model by a named Tracking. We use TrackingUses within TRules to establish and query correspondences between source and target elements.

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.

In the example we use a TrackingUse named “java_class_from_uml_classifier” 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 “java_class_from_uml_classifier” is used to query the correspondence between UMLClassifiers in the source extent, and Java Classes in the target, to bind “JC” to the Java Class that was mapped from the UMLClassifier that is the owner of the matched UMLAttribute “X”. The variable “JC” is then used in the MofFeature in the target of the TRule, to provide a value for the “owner” feature of the JavaField “Y” that is mapped from the UMLAttribute “X”.

```
Tracking java_class_from_uml_classifier

TRule umlAttributeToJavaField {
   track(JC, java_class_from_uml_classifier, UC) and
   X:UMLAttribute[owner -> UC]
   ->
   Y:JavaField[owner -> JC] and
   track(Y, field_from_attr, X)
}
```

Figure 3-5 Using TrackingUse to query correspondences

3.7 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 3-6 to represent these commonly used MofFeatures. The TRule “umlClassifierToJavaClass” can now be rewritten to refer to these PatternDefns, using a PatternUse.

```
PatternDefn umlClassifierAndName(X,N) {
   X:UMLClassifier[name -> N]
}

PatternDefn javaClassAndName(X,N) {
   X:JavaClass [name -> N]
}

TRule umlClassifierToJavaClass(X,Y) {
   javaClassifierAndName(X,N)
   ->
   javaClassAndName(Y,N) and
   track(Y, java_class_from_uml_classifier, X)
}
```

Figure 3-6 Example of PatternDefn and PatternUse

3.8 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 3-7. The “X” and “Y” variables of “umlInterfaceToJavaInterface” supersede those of the same name in “umlClassifierToJavaClass”.

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 3-7. As variables “X” and “Y” are extended from the variables of “umlClassifierToJavaClass”, it is not necessary to specify their type again in the MofFeatures used in the TRule to bind the name of the UMLClass to the name of the constructor method in the Java Class.

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

3.9 Tracking Hierarchies

Trackings can be arranged in hierarchies. Tracking subtyping means that when a correspondence is recorded in the child tracking, it will also be present (and queryable) in the parent tracking. Tracking 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 3-7 shows the “java_intf_from_uml_intf” tracking subtyping the “java_class_from_uml_classifer” tracking, so that we may query correspondences between UMLClassifiers and corresponding objects in the target extent generally, using “java_class_from_uml”, or more specifically, only query correspondences between UMLInterfaces and Java Interfaces, using “java_intf_from_uml_intf”.

```
Tracking java_intf_from_uml_intf isa
      java_class_from_uml_classifier

TRule umlInterfaceToJavaInterface(X,Y)
   supercedes umlClassifierToJavaClass(X,Y) {
   X:UMLInterface[ name -> N ]
   ->
   Y:JavaInterface [name -> N] and
   track(Y, java_intf_from_uml_intf, X)
}

TRule umlClasstoJavaClass(X,Y)
   extends umlClassifierToJavaClass(X,Y) {
   X [name -> N]
   ->
   M:JavaMethod [name -> N] and
   Y[constructor -> M]
}
```

Figure 3-7 Example of TRule extending and superseding, and Tracking subtyping.

3.10 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 MofFeatureOrder and MofLinkOrder elements, respectively. 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 3-8 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).

```
TRule umlParameterToJavaParameter {
   X:UMLParameter[name -> N]
   ->
   Y:JavaParameter[name -> N] and
   track(Y, java_param_from_uml_param, X)
}

TRule parameterOrdering {
   order( Z[parameter -> X1], Z[parameter -> X2] ) and
   track(Y1, java_param_from_uml_param, X1) and
   track(Y2, java_param_from_uml_param, X2)
   ->
   order( M[arg -> Y1], M[arg -> Y2] )
}
```

Figure 3-8 Example of MofFeatureOrder

3.11 Extents

When we declared the “uml2java” transformation, as shown in Figure 3-4, we declared Vars “Source”, “Target” and “Tags” representing the extents of the transformation. All of the TRules so far have used “Source” as the default extent of the terms in the source of each TRule and “Target” as the extent of the terms in the target of each TRule. We can express transformation rules between elements from other extents by associating the extent with terms in the source or target of a TRule.

In our example transformation, see Figure 3-9, we wish to comment all JavaClasses resulting from the transformation with a standard copyright statement. We use the Tags model as the source for this copyright information, using the ‘@’ notation to explicitly specify the extent for the MofTerm that binds the value of the Tag to the variable “C”, which is then used to provide the value for the comment of each JavaClass.

```
TRule copyrightToJavaClass {
   X:Tag[name -> 'Copyright',
         type -> 'UMLClassifier',
         value -> C]@Tags and
   track(Y, java_class_from_uml_classifier, Z)
   ->
   Y[comment -> C]
}
```

Figure 3-9 Working with a non-default extent

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(Source, Target, Tags) {
   Tracking java_class_from_uml_classifier

   Tracking java_intf_from_uml_intf isa
      java_class_from_uml_classifier

   PatternDefn umlClassifierAndName(X,N) {
      X:UMLClassifier[name -> N]
   }

   PatternDefn javaClassAndName(X,N) {
      X:JavaClass[name -> N]
   }

   TRule umlClassifierToJavaClass(X,Y) {
      javaClassifierAndName(X,N)
      ->
      javaClassAndName(Y,N) and
      track(Y, java_class_from_uml_classifier, X)
   }
   TRule umlAttributeToJavaField {
      track(JC, java_class_from_uml_classifier, UC) and
      X:UMLAttribute[owner -> UC ]
      ->
      Y:JavaField[owner -> JC] and
      track(Y, field_from_attr, X)
   }

   TRule umlInterfaceToJavaInterface(X,Y)
      supercedes umlClassifierToJavaClass(X,Y) {
      X:UMLInterface[name -> N]
      ->
      Y:JavaInterface[name -> N] and
      track(Y, java_intf_from_uml_intf, X)
   }

   TRule umlClasstoJavaClass(X,Y)
      extends umlClassifierToJavaClass(X,Y) {
      X[name -> N]
      ->
      M:JavaMethod[name -> N] and
      Y[constructor -> M]
   }
```

```
   TRule umlParameterToJavaParameter {
      X:UMLParameter[name -> N]
      ->
      Y:JavaParameter[name -> N] and
      track(Y, java_param_from_uml_param, X)
   }

   TRule parameterOrdering {
      order( Z[parameter -> X1], Z[parameter -> X2] ) and
      track(Y1, java_param_from_uml_param, X1) and
      track(Y2, java_param_from_uml_param, X2)
      ->
      order( M[arg -> Y1], M[arg -> Y2] )
   }

   TRule copyrightToJavaClass {
      X:Tag[type -> 'UMLClass', value -> C]@Tags and
      track(Y, java_class_from_uml_classifier, Z)
      ->
      Y[comment -> C]
   }

}
```

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 (Vars) that their contained VarUses can reference. A Var may only be referenced by a VarUse that is contained (directly or indirectly) by the VarScope that contains the Var.

Attributes

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

Associations

var: Var [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 Var::scope.

Constraints

Semantics

4.2.2 Var

Var is the declaration of a variable within a VarScope.

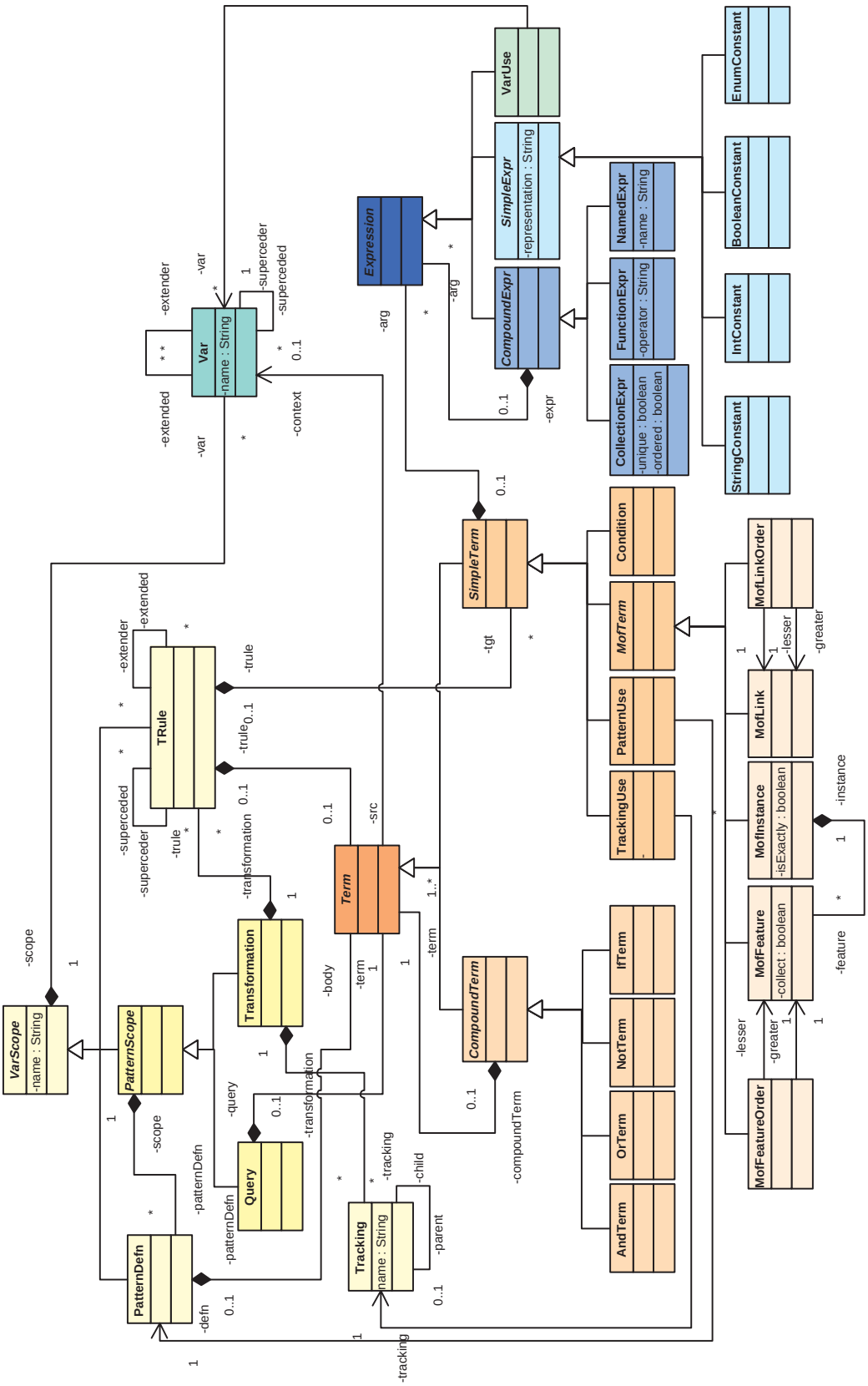

Figure 4-1 Complete Transformation Model

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.

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

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

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

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

Semantics

4.2.3 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: Var [0..*] {composite, ordered} (from VarScope)

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

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

Constraints

Semantics

4.2.4 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 Vars owned by the PatternDefn via the parameter association act as parameters to the PatternDefn, any other Vars owned by the PatternDefn are effectively local variables. PatternDefn inherits from VarScope.

Attributes

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

Associations

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

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

The Vars 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

4.2.5 Query

A Query is used to name and parameterise a Term which matches values in the source extents. The Vars owned by the Query via the parameter association act as parameters to the Query, any other Vars 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: Var [0..*] {composite, ordered} (from VarScope)

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

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

The Vars 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

4.2.6 Transformation

A Transformation consists of variables (Vars) 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: Var [0..*] {composite, ordered} (from VarScope)

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

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

tracking: Tracking [0..*] {composite}

The set of owned relationships used to track the functional dependencies between a target element and a set of source elements. The opposite association is Tracking::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 Tracking dependencies are used to identify target objects that must be created, and the equivalences are used to determine their types and property values.

4.2.7 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: Var [0..*] {composite, ordered} (from VarScope)

The set of Vars introduced by this rule. The opposite association is Var::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 “N” in the following src term is *not* sufficiently constrained:

```
P:person and not P[name -> N]
```

More formally, every Var owned by the TRule must occur *positively* in the src Term, or be referenced by the target VarUse of a tgt TrackingUse. A Var 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 Vars are used to correlate the Vars and VarUses when performing the Term copies to construct the extended src terms and match terms.

4.2.8 MofTerm

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

Associations

context: Var [0..1] (from Term) A MofTerm must be evaluated in the context of a particular extent, whose value is held by the Var.

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

4.2.9 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).

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: Var [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)

feature: MofFeature [0..*] {composite}

The (partial) set of features of the MOF Class that must also be matched. The opposite association is MofFeature::instance.

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

Specifies the name of the MOF Class.

instance: VarUse [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 variable.

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

Semantics

4.2.10 MofFeature

A MofFeature term is used to match the value(s) of a feature from the containing MofInstance's type. MofFeature inherits from MofTerm (and hence from SimpleTerm and Term).

Attributes

collect: boolean — If true, then the Var referenced by the value VarUse is bound to a collection containing all the values of the feature for the matched MofInstance. If false, then the Var referenced by the value VarUse is bound to each value of the feature for the matched MofInstance.

Associations

context: Var [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)

instance: MofInstance [1] — The MofInstance that contains this MofFeature. The opposite association is MofInstance::feature.

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

Specifies the name of the feature.

value: VarUse [1] (subsets SimpleTerm::arg)

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

Constraints

A MofFeature must have 2 args, one of which is the featureName and the other the value variable.

Semantics

4.2.11 MofLink

A MofLink term is used to match the members of an Association. MofLink inherits from MofTerm (and hence from SimpleTerm and Term).

Attributes

Associations

context: Var [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 Association.

member: VarUse [2] {ordered} (subsets SimpleTerm::arg)

Specifies the VarUses that reference the Vars to which the two ends of the matched links will be bound.

Constraints

Every MofLink must have 3 args, one of which is the typeName and the other two are VarUses for the objects linked by the association.

Semantics

4.2.12 MofFeatureOrder

A MofFeatureOrder is used to determine pair-wise ordering of feature values. If the feature is ordered, then MofFeatureOrder matches according to this order. If it is not ordered, then an implementation must provide a total order that remains constant. This is important so that the Features have a stable, albeit arbitrary, ordering when they are matched by several different TRules in a Transformation. MofFeatureOrder inherits from MofTerm (and hence from SimpleTerm and Term).

Attributes

Associations

context: Var [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)

lesser: MofFeature [1]	The MofFeature which comes before the greater MofFeature.
greater: MofFeature [1]	The MofFeature which comes after the lesser MofFeature.

Constraints

The lesser and greater MofFeatures must be owned by the same MofInstance.

Semantics

4.2.13 MofLinkOrder

A MofLinkOrder is used to determine pair-wise ordering of links. If the Association is ordered, then MofLinkOrder matches according to this order. If it is not ordered, then an implementation must provide a total order that remains constant. This is important so that the links have a stable, albeit arbitrary, ordering when they are matched by several different TRules in a Transformation. MofLinkOrder inherits from MofTerm (and hence from SimpleTerm and Term).

Attributes

Associations

context: Var [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)

lesser: MofLink [1]	The MofLink which comes before the greater MofLink.
greater: MofLink [1]	The MofLink which comes after the lesser MofLink.

Constraints

The lesser and greater MofLinks must belong to the same association.

Semantics

4.2.14 Tracking

A Tracking is used to name a functional dependency between a target model element and a set of source model elements. A TrackingUse is used to define and query the elements for which the functional dependency holds.

Any tuple of elements that belongs to a Tracking (as determined by TrackingUses in the target of a TRule) also belongs to its parent's Tracking. This is transitive up the hierarchy.

Attributes

name: string	The name of the Tracking.

Associations

transformation: Transformation [1]

The transformation that contains this tracking. The opposite association is Transformation::tracking.

parent: Tracking [0..1]	Any functional dependency that holds for this Tracking also holds for its parent tracking. The opposite association is Tracking::child.
child: Tracking [0..*]	Any functional dependency that holds for a child tracking also holds for this tracking. The opposite association is Tracking::parent.

Constraints

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: Var [0..1] — A reference to a Var 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: Var [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: Var [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: Var [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: Var [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: Var [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: Var [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 defines a functional dependency between the element bound to the *tgt* referenced variable and the set of elements bound to the *src* referenced variables. This effectively establishes the identity of model elements that need to be created in the target extent(s) by the transformation.

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. Note that to match against a Tracking, one must first evaluate all TRules that populate the Tracking.

Tracking inherits from SimpleTerm (and hence from Term).

Attributes

Associations

context: Var [0..1] (from Term) Not used for TrackingUse.

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

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: Tracking [1] The named functional dependency to query/populate.

Constraints

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: Var [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 Vars.

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: Var [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

simpleTerm: 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 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 usual set of operations on the available types: string, int, boolean, enum, and collection. For example, “>“, “>=”, “==”, “!=”, “+”, “*”, “union”, “intersection”, “and”, etc.

Semantics

4.2.35 NamedExpr

Represents the invocation of the library function in the transformation engine named *name*. The arguments to this NamedExpr will be used as arguments to the named library function.

Attributes

name: string The name of the library function to be invoked.

Associations

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

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

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

Constraints

Semantics

Conformance 5

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

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

References 6

[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™ 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™)", 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", 2002, OMG ad/02-12-10

References

[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.*