Model Transformation: A declarative, reusable patterns approach

Keith Duddy, Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel
CRC for Enterprise Distributed Systems (DSTC)
{dud,agerber,lawley,kerry,steel}@dstc.edu.au

Abstract

The MOF Query, View and Transformation RFP, issued by OMG will result in a key enabling technology for model-driven development of large distributed systems. We have designed a transformation language which will meet the requirements of this RFP, and several others besides. The language is declarative and patterns based. Transformation descriptions are explicitly reusable and modular. Rules that make up such descriptions may be aspect-driven, allowing for transformations to be written to address semantic concepts rather than structural features. This paper introduces the language and its rationale, and shows how it is used to solve a small but non-trivial MDA problem.

1 Introduction

The Meta Object Facility (MOF) [13] is a technology specification standardised by the Object Management Group in 1997. It provides an object-oriented framework for the specification of the abstract syntax of modeling languages. The benefits of using this facility for the specification of languages such as the Unified Modeling Language (UML$^{\text{TM}}$) [14] is that there are a standard mechanisms for automatically deriving

- a concrete syntax based on XML DTDs and/or schemas known as XML Model Interchange [18] (XMI)
- a customisable human-usable textual notation [16] or HUTN
- a set of interfaces in CORBA IDL or Java for programmatic access to object model repositories

However, to date, the common MOF foundation of OMG languages such as UML, the Common Warehouse Metamodel (CWM$^{\text{TM}}$) and the Enterprise Distributed Object Computing (EDOC) model has not enabled the use of a model in one language to be transformed into a model in another language, except by the following limited means:

- An XML document representing one model in the standard XMI form may be manipulated using XSLT to produce another model.
- A program may traverse the model using CORBA or Java interfaces, and populate another model in a different repository.
- Partial transformations of data may be described in the CWM.

All of these approaches have some usefulness. However, a language for describing the generic transformation of any well formed model in one MOF language into a model in some other MOF language (or perhaps in the same language) is not yet available in a standard form. The OMG has issued a Request for Proposals named "MOF Queries, Views and Transformations", known as QVT for short [22]. It requires submissions to:

- define a language for querying MOF models
- define a language for transformation definitions
- allow for the creation of views of a model
- ensure that the transformation language is declarative and expresses complete transformations
- ensure that incremental changes to source models can be immediately propagated
- express all new languages as MOF models

The DSTC Pegamento project has submitted such a proposal to the OMG [5]. The following sections provide an overview of its structure and semantics and show how it can be used to solve a small but non-trivial example of a mapping from the ECA Entity Model to an EJB/Java model.

2 Related Work

A number of partial solutions to describing and implementing model transformations are currently available. Some of these are applicable only in a limited domain, or provide very low-level abstractions for transformations.

2.1 CWM Transformation

The OMG's Common Warehouse Metamodel Specification [4] contains a model for describing Transformations. It introduces the concepts of black- and white-box transformations. Both of these transformation styles provide only a relationship between model elements which are the sources and targets of a transformation, but do not express exactly what the resulting target will consist of. White-box transformations may have a *ProcedureExpression* associated with the transformation, allowing for a program fragment in some implementation language to describe the implementation of the transformation.

2.2 Graph Transformation

Varró et al [26, 6] describe a system for model transformation based on Graph Transformations [2]. This style of transformation is based on the application of an ordered set of rules. Operators available include transitive closure, and repeated application. Rules identify sub-graphs which define before and after states, and may refer to source and target model elements and introduce associations between.

The use of this kind of transformation for software engineering would require a copy of the initial model to be kept, as the graph is modified in place. Care needs to be taken in ordering rules with repeated application, as non-termination is not guaranteed.

2.3 Generated XSLT

Peltier et al. [21, 20, 1] propose that a transformation approach that operates on textual representations of models. The rules are expressed as a model instance, and then translated into a form that manipulates the textual documents. The current implementation uses XMI [18] as the textual model format, and generates XSLT from the model of the transformation rules which can then be applied to the XMI documents.

Rules are required to be ordered, and are restricted to defining a single target element per source element. Creation of target model elements is done explicitly in the rules, which in part explains the requirement for rule ordering, as rules creating objects must execute before rules populating their contents or participation in associations.

2.4 XSLT

XSLT [27] may be used effectively for some class of transformations of MOF models, as they may be represented as XML documents via the XMI specification. However, XSLT must be written in terms of the concepts in the source XMI document (model), and object (or element) creationis explicit. The style is highly procedural and due to its XML basis, the concrete syntax is very user unfriendly. As such, it is usuitable for one of the major goals of a declarative transformation language - which is to communicate mapping specifications to human beings. It also requires a complete document as input, and is therefore not amenable to transforming incremental updates to models.

2.5 Action Semantics

Since UML 1.4 the Action Semantics language (ASL) has been a standard part of UML. ASL has been submitted to OMG [9, 25] as a candidate reponse to the MOF QVT RFP. The ASL provides a number of constructs at the level of 2GL programming language statements that can be composed to provide specifications of actions. Its imperative nature makes it rather unsuitable for describing patterns within models which are to be transformed into new model elements. At best it offers a form of pseudo-code for descrribing algorithms for walking one graph and explictly constructing another graph.

3 Additional Requirements for Transformation

The basic requirements for a transformation language are spelled out in the RFP, however, we have formulated the following additional criteria for usability of the language and reusability of the definitions expressed in it.

These are some of the additional requirements:

- Transformation rules should be able to match both collections of elements and single elements. That is, they 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.

- The application of a rule should establish associations between source and target model elements. These associations would then be used for maintaining traceability information.

- The language should allow for the definition of a stable total order over any unordered multi-valued attributes or unordered association links. The need for such stable orders typically arises when different mapping rules must be applied to the first and/or last element of some collections of values.

- It should handle recursive structure with arbitrary levels of nesting.

- Rules should be able to 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.

- Transformations should allow both multiple source extents and multiple target extents.

- There should be no dependency on the application order of rules, and all rules should be applied to all source elements.

- Creation of target objects should be 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 should be 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.

- Rules should be able to be grouped naturally for readability and modularity.

- Transformation patterns should be definable, thus supporting modular transformation definitions.

These requirements are set forth to allow transformations be written in a variety of styles (typically source-driven, target-driven or aspect-driven). See Section 5.

4 Concepts and Terminology

The transformation language has the following basic concepts: rules, patterns and tracking relationships.

Transformation rules are used to describe a correspondence between patterns of elements in a source model or models and the elements to be created in a target model or models. They will typically have variables declared which are used as parameters to a pattern in the source part of the rule which will bind them to values from the source model, and then these values will be used to populate another pattern in the target.

Patterns are expressed as reusable named definitions with parameters. A pattern will also have local variables used within the definition. When used in the source of a rule a pattern is a query which, when applied to a model element with the pattern parameters bound to external variables, will have a boolean result. When the result is true, the external variables will be populated with values from the model to which the pattern was successfully applied.

When a pattern is used in the target of a rule it acts as a template for model elements which will exist in the target model if the source of the rule matched. The parameters in this case are bound to external variables which contain values for the instantiation of the template.

Tracking relationships are used to associate the source model elements with the target model elements whose existence a rule implies. They are named relationships which typically occur in several different rules and allow each rule to address only part of the creation of a target object or structure of objects and relationships, without duplicates being created. An example shown in [5] is reproduced in Figure 1.

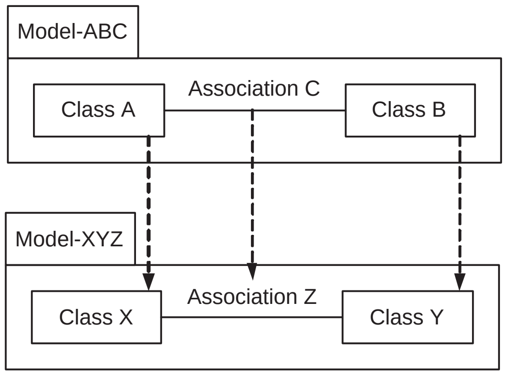

Figure 1. A common transformation kind.

This transformation can be described using three rules, one for Class A to Class X, another for Class B to Class Y, and a third for the association mapping. The first two rules are easily written, but the third will need to be able to refer to the instances of Classes X and Y that were created from instances of Classes A and B so that the newly created link of type Association Z has the correct objects at either end. This is done by using tracking relationships in the rules for transforming the classes and referring to the relationships in the rule for the association.

5 Styles of Transformation

The Model Driven Architecture (MDA^{TM}) development paradigm that is being championed by the OMG has associated terminology that can be misinterpreted to constrain the kinds of model transformations required. The terms Platform Independent Model (PIM), and Platform Specific Model (PSM) imply that a single style of transformation is required to map the more abstract PIM to a form in which enough concrete platform information is available in a PSM to proceed with an implementation.

The technical whitepaper produced by the OMG Architecture Board [19] clarifies this situation somewhat by making clear that PIM to PSM transformation is a pattern that can be applied at many levels of abstraction - and that the pattern must be reified in term of a particular platform. For example, a data model expressed in UML as a set of classes (the PSM) may be mapped to a set of SQL tables (the PIM), where the independence and specificity of the models are relative to the platform of relational databases. A layered example is a component-based design in the EDOC ECA modeling language [17] which can be mapped to a model of some CORBA Components. In this case the platform is CORBA 3.0. However, the component descriptions may be further mapped to a set of C++ classes and their associated stub code and libraries. This PIM/PSM pair is relative to the platform of, say, Solaris 2.5, Visibroker for C++ and gcc.

However, there are many other kinds of model to model transformations in which the level of abstraction remains the same, or even becomes greater. Transforming data models from Relational to Object Oriented, or generating a CORBA IDL wrapper interface from a legacy COBOL system are two examples.

5.1 Source-driven Transformations

Our transformation language is capable of expressing mappings between a single source model instance or even a single property thereof to a complex set of target elements with relationships between them. This style of transformation is most common when doing PIM to PSM transformations. It works well when the source instance is tree-like, but is less suited to graph-like sources.

5.2 Target-driven Transformations

Conversely the transformation language may have a single specific target model element as the subject of each rule, and match an arbitrarily complex set of model elements in the source model which entail its creation.

5.3 Aspect-driven Transformations

This approach, which is inspired by the viewpoint concept from RM-ODP [7], and more recently, by aspect-oriented programming [10, 11], structures rules around concepts rather than objects. Transformation descriptions written in this style will have some rules that imply the creation of only one part of a target object based on some complex pattern in the source model, and others that create a set of target objects and associations between them based on a simple property match in the source model.

A typical example of a PIM to PSM transformation in the aspect-driven style is the transformation of an architectural design for a component-based system into a set of interfaces suitable for a middleware platform. In this style we would have one set of rules which map, say EDOC ECA process components into Java interfaces to support EJBs. Another set of rules would map properties in the ECA model to transactional policies for the EJBs. Perhaps a third set of rules would match a configuration model external to the ECA model against the components in that model, and create additional target model elements, like deployment descriptor elements.

Aspect-driven transformations are the reason for many of the usability requirements given in Section 3. In particular the need for multiple rules per object and implicit object creation is motivated by the aspect-driven approach. When several rules may result in the creation of a target object, and it is unknown which of these will match the source model, it is impossible for the transformation architect to know which rule will be responsible for creation, and implicit creation is therefore the only useful semantics.

There are several motivations for modeling the relationships between the source and target models, which we call trackings. The primary reason is so that rules may cross-reference other rules, and refer to the objects in the source model which may have been matched by another rule, as well as the objects in the target model whose creation is implied by a match of another rule.

6 Transformation Model Semantics

The MOF model diagram in Figure 2 represents the Transformation model specified in [5]. The following subsections explain the important metaclasses.

6.1 Terms and Expressions

The lower part of Figure 2 is an expression language metamodel constructed specifically for identifying MOF model elements in patterns and rules. Its three main abstract metaclasses are *CompoundTerm* which deals with boolean algebra, and owns a number of *Term*s, which represent the operands to the boolean operators.

Expression is the superclass to *SimpleExpr* whose subtypes are all of the literal values in the language, such as strings and integers. Its other subtypes include *CollectionExpr* for literal set, bag and list expressions, *FunctionExpr* for calling MOF operations, and *NamedExpr* for access to library functions. Finally, *VarUse* is the place in which a variable name may be used in an expression.

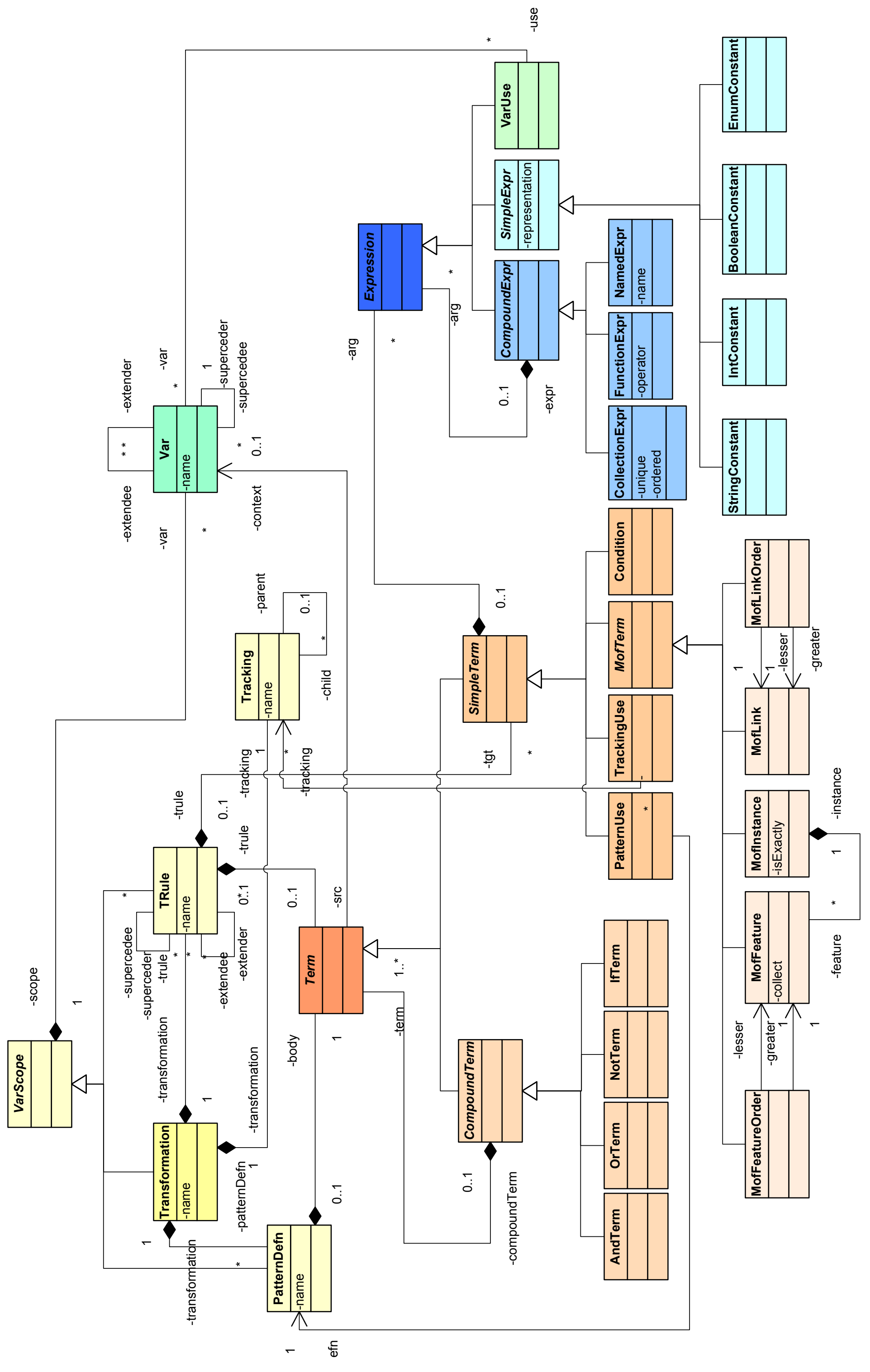

Figure 2. The Transformation Metamodel.

SimpleTerm is the superclass to all of the interesting MOF and Transformation related terms. It optionally owns a number of expressions, which are the "arguments" to its concrete subtypes.

PatternUse is a use of a pattern definition in which the expressions owned by the SimpleTerm represent the variables or values to be bound to the parameters of a pattern definition.

MofTerm is the place in the model where MOF class instances, attribute values and link values can be denoted.

Condition is a boolean valued expression.

TrackingUse is the place where a named relationship between source and target model elements is declared in a rule, or referenced by another rule. It is fully explained in Section 6.5

6.2 Variables

The *Var* metaclass is a variable declaration. Variables in this model are dynamically typed, and a Var slot may contain any valid MOF type. MOF reflection is used to determine the type of any variable, and this is not explicitly modelled here. The reader will note however, that in the examples given below, a number of type assertions are made on Variables. This is an assumed capability of a transformation tool, and is a shorthand for something that can be explicitly represented in the Transformation model by using the MOF metamodel itself.

Vars have *extended* and *superseded* relationships with other Vars. This is for the purpose of identifying which Vars in a rule bind to the same values as those in other rules related by rule extension and supersession. Section 6.4 explains these relationships between rules.

The other relationship, not visible to the Var, in which a Var may be involved is the *context* association. This is to allow a Term in a Rule to identify the MOF extent with respect to which it is to be evaluated. Transformations may have multiple source models and may produce multiple target models. They will declare Vars as parameters to allow users of the Transformation to pass in the MOF extents containing the source models, and the MOF extents in which the target model will be created. In concrete syntax representations of a Transformation description, the context is unambiguous, but we do have a notation to allow a specific variable name (representing an extent parameter) to be associated with a rule's source term.

6.3 Patterns

The *PatternDefn* model element is used for two distinct purposes in the model. In the source of a transformation rule it is used to match model elements in a source model, and in the target of a rule it is used as a template for new model elements.

Pattern Definitions are *VarScopes*, which means they own *Var*s, which are declarations of variables. These can be seen as formal by-reference parameters to the pattern. They are used in the source to bind values in the source model (when the pattern matches) to the the Variables passed in to the PatternDefn by a *PatternUse*. When used in the target, a PatternDefn is passed in Variables by a PatternUse which have values, and these are used to instantiate the model elements that the Pattern defines.

Here is an example pattern definition which matches UML Classes and their attributes, both those owned by the Class, and all of its superclasses.

```
PATTERN hasAttr(C, A)
FORALL Class C, Attribute A, Class C2
WHERE A.owner = C
   OR (C.super = C2 AND hasAttr(C2, A))
```

We will see a usage of this pattern in an example rule below.

6.4 Transformation Rules

The *TRule* model element in Figure 2 is a description of how a pattern in a source model is to be used to create some part of a target model element. A rule owns a "src" Term, which is a predicate stating under what conditions the "tgt" SimpleTerm (some MOF model elements) are to be created. In addition, the rule will typically declare a number of local variables, which are assigned values when the source model is matched, and those values are then used to populate the target model elements.

Rules are named for the convenience of concrete syntax representations, but the results of a rule being used to create model elements are stored in Trackings, as explained in Section 6.5. This is the mechanism by which one rule can refer to the results of the execution of another rule.

Transformation rules may be related to other rules in two ways

- A rule which *extends* another rule can add additional clauses to the source matching predicate, and it can also add additional Terms to the set of target elements and Trackings to be created.

- A rule which *supersedes* another can also refine its matching predicate, but its target Term replaces the Term for the superseded rule. This mechanism may be used to alter or turn off rules in particular circumstances.

Both rule extension and supersession allow for Transformations to be reused, and specialised. They also allow rules to be written simply for a general case, and then superseded for some exception.

The syntax we use for rules is exemplified by the following mapping of UML Class to a Java Interface:

```
RULE Uclass2Jintf(Cls)
FORALL Class Cls
MAKE Interface Intf
```

6.5 Trackings

Trackings are declarations of the kinds of relationships between source and target models that need to be stored for traceability, as well as being a mechanism for cross-references between rules that address the same types of model element.

A *TrackingUse* is the expression of such a relationship, which either asserts that the relationship exists (when used in the target of a rule) or queries whether such a relationship exists (when used in the source of a rule). Its arguments (via the *arg* association of its supertype SimpleTerm), are usually variable names (VarUses) whose values are references to model elements for which a tracking relationship is asserted by a TrackingUse. To extend the example above, we add a TrackingUse to the target part of the rule.

```
RULE  Uclass2Jintf(Cls)
FORALL Class Cls
MAKE Interface Intf,
LINKING Cls to Intf by Uclass2Jint
```

Then this tracking name is used in the source of a (naive) rule to map each attribute of the UML Class to a Java Property. And we will create another tracking to note the creation of the Property.

```
RULE Uattr2Jprop(Attr, Cls)
FORALL Attribute Attr, Class Cls
WHERE hasAttr(Cls, Attr)
AND Uclass2Jint LINKS Cls to Intf
MAKE Property Pr,
     Pr.owner = Intf,
     Pr.name = Attr.name,
LINKING Attr to Pr by Uattr2Jprop
```

Trackings are related to other Trackings by a *parent/child* association. It allows for the definition of tracking hierarchies which act like object-oriented inheritance. A linkage created between objects in a child tracking will also be present and queryable in the parent tracking. This can be useful when writing rules that address features of base types in the source and target models without the need to know what the most derived type of the objects in question are. It may also be used in combination with rule extending and superseding.

6.6 Transformation

The *Transformation* metaclass is the container for all of the patterns, rules, tracking declarations variable declarations required to define some mapping from a MOF source extent to a MOF target extent. It defines a name space for rules, so that name clashes may be avoided, and explicit reuse of rules from other transformations can be facilitated in concrete syntaxes.

7 "Simple" Mappings

The core of the EDOC specification is called the Enterprise Collaboration Architecture (ECA). It is a structured framework for recursive definitions of computational objects and their interactions. It can represent designs for B2B interactions, container managed entities and components, synchronous and asynchronous messaging, and workflow-style processes. ECA is represented both as a Profile for UML Classes and their Collaborations, as well as a MOF meta-model. In this paper we will concentrate on the Entity model in ECA, which is an abstraction of the capabilities of distributed component middleware such as CORBA Components [15], Distributed COM [12], and Enterprise Java Beans [24] (EJB). This part of the language was chosen for its familiarity to readers, as space does not permit an in-depth introduction to the ECA meta-model and its semantics.

Figure 3 shows an excerpt of the ECA Entity meta-model. The concept of *Composite Data* should be familiar to readers as a data type model consisting of nested structures of named data *Attributes*, with some of those attributes identified as *Keys*. *Data Managers* are the computational interfaces that expose the data to the application via *Operations*. *Entities* are special Data Managers that can be identified via *Key* attributes.

As an example source model to illustrate the issues covered in this paper we use an ECA Entity model instance that describes simple aspects of customers and customer records. This model is shown in Figure 4 in familiar UML Class Diagram notation. It is also shown in Figure 5 using UML Object Diagram notation since it *is* an instance of the ECA Model, this notation shows all the attribute details, and it is this view that is appropriate for understanding the transformation rules.

The model shows several CompositeData types, *Address* and *Customer* where an attribute of Customer has the type Address. It also includes a *CustomerRecord* which is an identifiable Customer with an attribute *ID* that is the *Key* of the CustomerRecord. These CustomerRecords are then exposed to clients via the *CustomerManager* that is both *networkAccessible* and *sharable* (meaning that it is persistent and must participate in transactions).

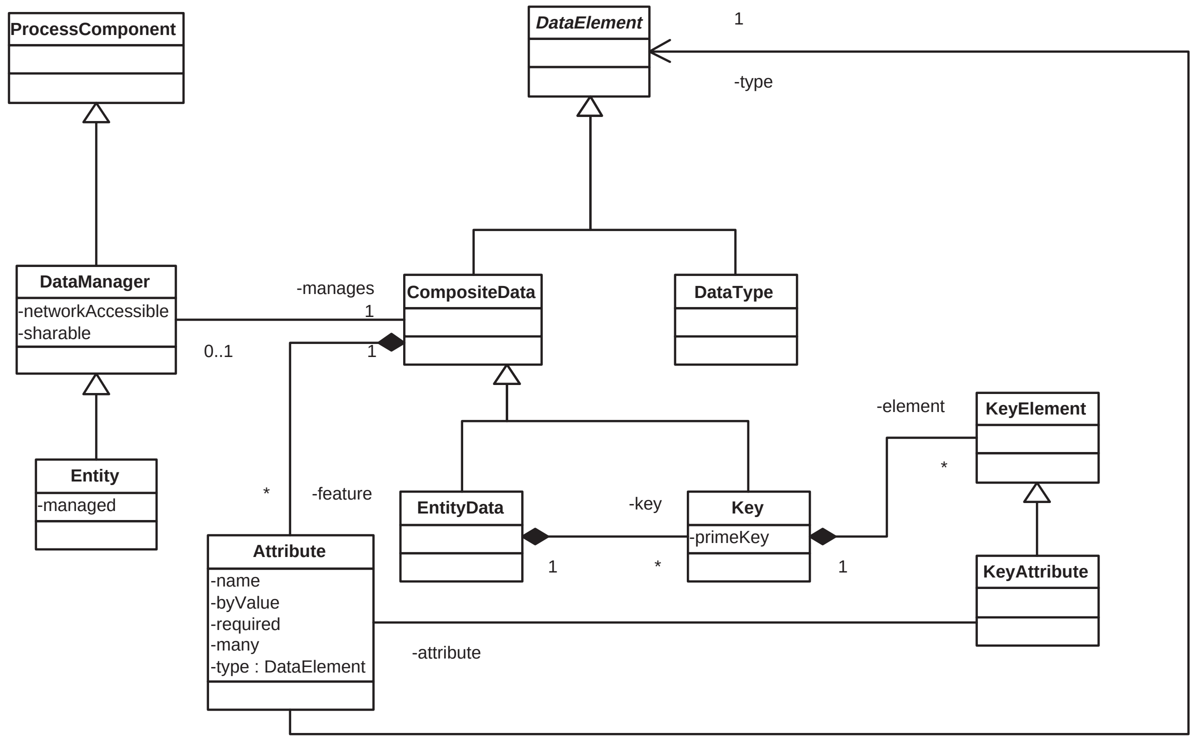

Figure 3. Part of the ECA Entity Model.

Part of a simple mapping from a CCA CompositeData to a Java Interface is given below. We omit appropriate getter and setter methods for the sake of brevity.

A mapping rule consists of a number of clauses. The RULE clause declares the rule's name and parameters, and any extended or superseded rules. The FORALL clause declares a set of source model variables and their types. These variables are further constrained by conditions in the WHERE clause, including references to previously populated tracking relationships. The MAKE clause declares target model variables and sets their properties. The LINKING clause establishes named tracking relationships between source and target model elements.

The rule below, `cd`, says that for every instance of a CCA CompositeData we must generate an instance of a Java Interface with the same name.

More specifically, for every binding of the variable `CD` such that `CD` is an instance of CompositeData in the source model, there must be an instance `JIFace` of Interface with `name` equal to `CD.name` in the target model, and that the tracking relationships `cdmap` and `dtmap` hold between the correlated pairs of CompositeData and Interface instances.

```
RULE cd(CD)
FORALL CCA::CompositeData CD
MAKE Java::Interface JIFace,
     JIFace.name = CD.name
LINKING CD to JIFace by cdmap,
        CD to JIFace by dtmap
```

The following rule, `cda`, *extends* the previous rule, `cd`, meaning that it is applied to exactly those instances of CCA CompositeData matched by the source expression of the rule `cd`. It generates a Java Attribute with the same name as each CCA Attribute, and with a type determined by looking up the `dtmap` tracking relationship. These Attribute instances are associated with the Interface instance generated in the rule `cd` that corresponds to the CompositeData instance by using the tracking relationship `cdmap`.

```
RULE cda(A, CD) extends cd(CD)
FORALL CCA::CompositeData CD,
       CCA::Attribute A,
       CCA::DataType Type
WHERE cdmap LINKS CD to JIFace
  AND dtmap LINKS Type to JType
  AND CD.feature = A
  AND A.type = Type
MAKE Java::Interface JIFace,
     Java::Attribute JAttr,
     Java::Type JType,
     JIFace.attributes = JAttr,
     JAttr.name = A.name,
     JAttr.type = JType
```

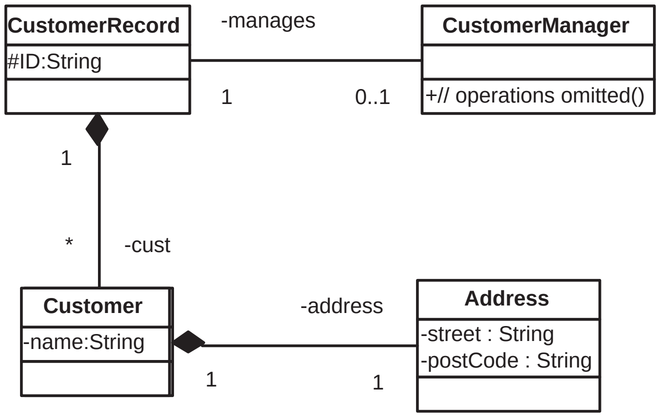

Figure 4. UML Class notation of example instance of the ECA Entity Model.

```
LINKING A to JAttr by cdattrmap
```

The previous rule is incomplete because it does not deal with the case where the CCA Attribute is multi-valued (i.e., its `many` attribute is true). We can deal with this by writing another rule, `cdaMul`, which *supersedes* the previous rule, `cda` as follows.

```
RULE cdaMult(A, CD) supersedes cda(A, CD)
FORALL CCA::CompositeData CD,
       CCA::Attribute A
WHERE A.many = true
  AND cdmap LINKS CD to JIFace
MAKE Java::Interface JIFace,
     Java::Attribute JAttr,
     Java::Interface ListFace,
     JAttr.name = A.name,
     JIFace.attributes = JAttr,
     JAttr.type = ListFace,
     ListFace.name = "java.util.List"
LINKING A to JAttr by cdattrmap
```

This rule actually changes the way the superseded rule `cda` is evaluated so that it will no longer be applied to those source elements that match the `src` pattern of the superseding rule.

8 Capturing Best Practice in the Mapping

In this section we discuss the drawbacks of using the primitives of the middleware platform as the basis for an application design, and the benefits of having a PIM and mapping rules. We then go on to show how the mapping in the previous section may be improved to encapsulate some known optimisations for EJB.

8.1 Design to Platform vs Model the Business

When designing an implementation of a middleware-based application, there is a body of knowledge in the developer community that provides advice and guidelines for getting the best performance and scalability from the application by altering the design to avoid known pitfalls, and to exploit known optimisations [3, 23, 8].

The best practice for design of EJB applications often avoids using the very features that were designed to provide a close correspondence between the domain concepts being supported by the application, and the application server platform itself. EJB Entities were originally designed to provide a Smalltalk-style class extent (Home) for a number of business entities (Entity Beans) whose state could automatically be managed by the container supporting the Entity Beans (Container Managed Persistence). It is now recognised that for most business entities, this model is far too fine-grained. For example, our Customer Records, even if they were real examples with many more attributes, would not be best implemented using a Home to manage a set of Entity Beans which represent the Customers.

The implication, as we will see below, is that the "natural" model implied by the platform provides a poor implementation. However, given that we have used a platform independent language to model our Customer Records, we have cleanly represented the business concept of a Customer as data and a data manager entity, without concern for performance. Now we can exploit the mapping process to produce a more performant implementation by encoding this expert knowledge in the mapping rules rather than altering, and consequently obfuscating, the original platform independent design.

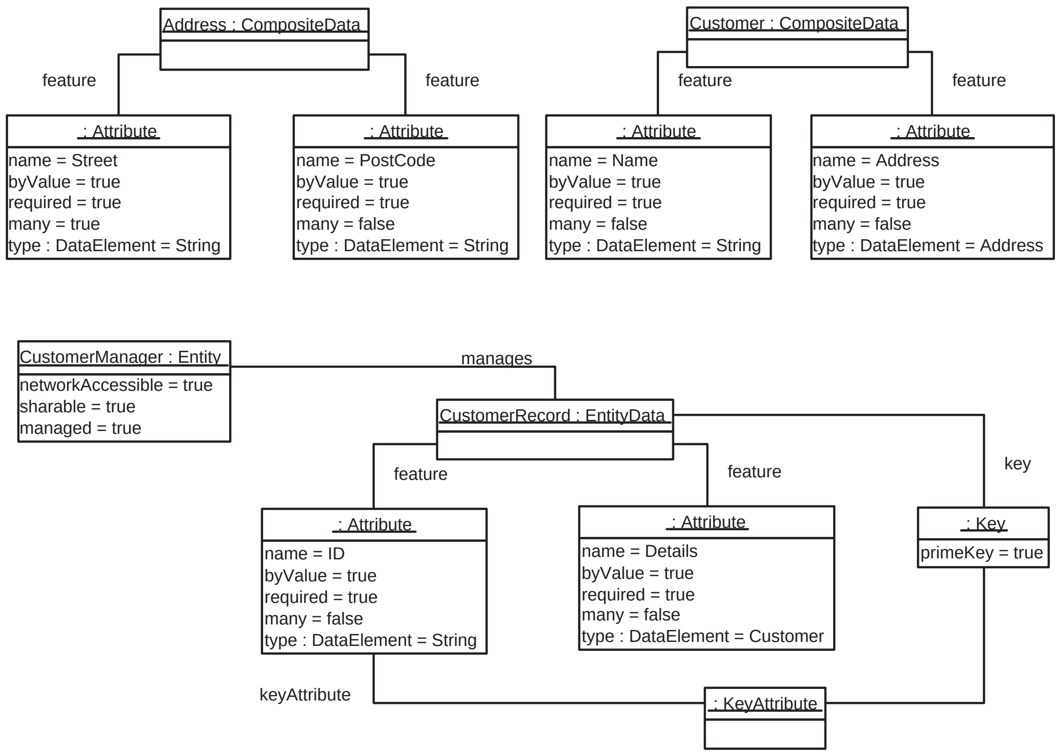

Figure 5. UML Object notation of example instance of the ECA Entity Model.

8.2 A Mapping Capturing Best-Practice Design Patterns

In the following mappings we show only those parts relevant to illustrating the main issues. We omit the details of Home interfaces and finder methods, implementation classes for the interfaces, etc.

The first mapping rule `dm` produces a Java Interface instance for every ECA DataManager instance in a similar manner to rule `cd` from Section 7. The second rule `dm2` extends `dm` for the case where the generated Interface must be network accessible. That is, the generated Interface instance inherits from the RMI `java.rmi.Remote` interface.

```
RULE dm (DM)
FORALL ECA::DataManager DM
MAKE Java::Interface JIFace,
     JIFace.name = DM.name
LINKING DM to JIFace by dmmap
```

```
RULE dm2 (DM) extends dm (DM)
FORALL ECA::DataManager DM
WHERE dm LINKS DM to JIFace
  AND DM.networkAccessible = true
MAKE Java::Interface JIFace,
     Java::Interface SuperFace,
     SuperFace.name = "java.rmi.Remote",
     JIFace.extends = SuperFace
```

The following rule both extends and supersedes the previous rule. This means that `dm3` only applies for instances of ECA Entity where the networkAccessible attribute is true while `dm2` only applies for instances of ECA DataManager that are not also instances of the subtype ECA Entity.

```
RULE dm3 (DM) extends and supersedes dm2 (DM)
FORALL ECA::Entity DM
WHERE dmmap LINKS DM to JIFace
MAKE Java::Interface JIFace,
     Java::Interface SuperFace,
     Java::Interface HomeSuperFace,
     Java::Interface JIFaceHome,
     SuperFace.name =
         "javax.ejb.EntityBean",
     JIFace.extends = SuperFace,
     JIFaceHome.name = DM.Name + "Home",
     HomeSuperFace = "javax.ejb.EJBHome",
     JIFaceHome.extends = HomeSuperFace
LINKING DM to JIFaceHome by dmhome
```

The astute reader may observe that the above mapping directly to an EntityBean is somewhat naive and may result in consequent performance penalties. A more sophisticated mapping would employ the Session Facade Pattern as described very lucidly by Kyle Brown in “Rules and Patterns for Session Facades” [3]. Other examples of expert knowledge regarding EJBs can be found on the Java Performance Tuning website [23].

The goal of employing the Facade pattern here is to abstract from the actual system objects (the objects representing ECA Entity) making them simpler to use and concealing unnecessary information such as Entity relationships.

A full mapping employing the Session Facade Pattern is beyond the scope and size limitations of this paper, but the essence can be captured in the following mapping rule. Remote access will now interact with a (Stateless) SessionBean which will then delegate methods to a local EntityBean.

```
RULE dm4 (DM) supersedes dm3 (DM)
FORALL ECA::Entity DM
WHERE dmmap LINKS DM to JIFace
MAKE Java::Interface JIFace,
     Java::Interface SessionBean,
     Java::Interface JIFaceLocal,
     Java::Interface EJBLocalObject,
     Java::Interface JIFaceHome,
     Java::Interface EJBLocalHome,
     JIFace.extends = SessionBean,
     SessionBean.name =
         "javax.ejb.SessionBean",
     JIFaceLocal.name = DM.name + "Local",
     JIFaceLocal.extends = EJBLocalObject,
     EJBLocalObject.name =
         "javax.ejb.EJBLocalObject",
     JIFaceHome.name =
          DM.name + "LocalHome",
     JIFaceHome.extends = EJBLocalHome,
     EJBLocalHome.name =
          "javax.ejb.EJBLocalHome"
LINKING DM to JIFaceLocal by dmlocal,
        DM to JIFaceHome by dmhome
```

The rule `dm4` effectively replaces the rule `dm3` since the source patterns are the same. Any other rules that involve referencing the generated Home object will do so via a lookup of the tracking relationship `dmhome` and will get the EJBLocalHome version rather than the superseded EJBHome version.

9 Conclusion

The benefits of an MDA approach to middleware application development have been demonstrated in the small by the example mapping of some instances of an ECA application model to an EJB implementation.

Firstly, the application architecture can be expressed in a “pure” form, without regard for the optimisation concerns of a particular middleware platform. This results in a consistent, easy to understand, domain-focussed model which can be easily explained to domain experts in diagrams.

Although platforms such as EJB attempt to offer applications a number of simplifying services to aid implementation, performance considerations often dictate that these services are worked around, or used only in certain constrained ways [23].

Thanks to the power of the proposed DSTC MOF Transformation specification the experience of multiple architects over multiple projects can be expressed as a concrete, declarative mapping from the concepts available in the PIM language to the concepts of the implementation platform. This mapping specification can capture the best practice for implementation of domain concepts on a particular platform.

In addition to mappings based only on the PIM concepts, configuration models can be exploited to add back the details of a platform that are abstracted out in languages like ECA. This effectively allows tools to query architects about which of the available implementation styles to choose for each PIM artifact based on, among other things, knowledge of the domain, and knowledge of probable usage patterns, performance, and robustness requirements.

In the future we envisage that modeling languages like ECA will be become more widely used. We expect that the MOF Transformation standard will form *the* key enabling technology for MDA. Transformation descriptions from PIM languages to PSM languages will become increasingly sophisticated, and will capture best practice in middleware design from highly legible domain-centric architectures.

10 Acknowledgements

The work reported in this paper has been funded in part by the Co-operative Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government’s CRC Programme (Department of Education, Science and Training).

References

[1] Alcatel, Softeam, Thales, TNI-Valiosys. Response to the MOF 2.0 Queries/Views/Transformations RFP. OMG Document: ad/03-03-25, Mar. 2003.

[2] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Pump, A. Schürr, and G. Taentzer. Graph transformation for specification

and programming. *Science of Computer Programming*, 34(1):1–54, Apr. 1999.

[3] K. Brown. Rules and Patterns for Session Facades. IBM's WebSphere Developer Domain: http://www7b.boulder.ibm.com/wsdd/library/techarticles/0106_brown/sessionfacades.html, June 2001.

[4] CWM Partners. Common Warehouse Metamodel (CWM) Specification. OMG Documents: ad/01-02-{01,02,03}, Feb. 2001.

[5] DSTC, IBM, CBOP. MOF 2.0 Query/Views/Transformations RFP. OMG Document: ad/03-02-03, Mar. 2003.

[6] S. Gyapay and D. Varró. Automatic Algorithm Generation for Visual Control Structures. Technical report, Dept. of Measure,ment and Information Systems, Budapest University of Technology and Economics, Dec. 2000.

[7] ISO. ISO/IEC 10746:1996 information technology – open distributed processing – reference model, 1996.

[8] R. Kalidindi and R. Datla. Best Practices to improve performance in EJB. PreciseJava: http://www.precisejava.com/javaperf/j2ee/EJB.htm, Nov. 2001.

[9] Kennedy Carter. Initial Submission to MOF Query, Views and Transformations. OMG Document: ad/03-03-11, Mar. 2003.

[10] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es):154, 1996.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[12] Microsoft. DCOM technical overview. Technical report, Microsoft Corporation, 1996.

[13] OMG. Meta Object Facility (MOF) v1.3.1. OMG Document: formal/01-11-02, Nov. 2001.

[14] OMG. Unified Modeling Language v1.4. OMG Document: formal/01-09-67, Sept. 2001.

[15] OMG. Corba Component Model. OMG Document: formal/02-06-65, June 2002.

[16] OMG. Human-Usable Textual Notation. OMG Document: ad/02-03-02, Apr. 2002.

[17] OMG. UML Profile for Enterprise Distributed Object Computing (EDOC). OMG Document: ptc/02-02-05, Feb. 2002.

[18] OMG. XML Metadata Interchange V 1.2. OMG Document: formal/2002-01-01, Jan. 2002.

[19] OMG Architecture Board MDA Drafting Team. Model Driven Architecture – A Technical Perspective. OMG Document: ormsc/01-07-01, July 2001.

[20] M. Peltier, J. Bézivin, and G. Guillaume. MTRANS: A general framework, based on XSLT, for model transformations. In *WTUML'01, Proceedsings of the Workshop on Transformations in UML*, Genova, Italy, Apr. 2001.

[21] M. Peltier, F. Ziserman, and J. Bézivin. On levels of model transformation. In *XML Europe 2000*, pages 1–17, Paris, France, June 2000. Graphic Communications Association.

[22] Request for Proposal: MOF 2.0 Query/Views/Transformations RFP. OMG Document: ad/02-04-10, Apr. 2002.

[23] J. Shirazi. EJB Performance Tips. http://www.javaperformancetuning.com/tips/j2ee_ejb.shtml, Nov. 2002.

[24] Sun Microsystems. Enterprise JavaBeans^{TM}Specification, Version 2.0 . Sun Microsystems: http://java.sun.com/produc ts/ejb/index.html, Aug. 2001.

[25] Tata Consultancy Services. Initial Submission to MOF Query/Views/Transformations RFP. OMG Document: ad/03-03-27, Mar. 2003.

[26] D. Varró, G. Varraó, and A. Pataricza. Designing the Automatic Transformation of Visual Languages. Accepted for Science of Computer Programming.

[27] W3C. XSL Transformations (XSLT) v1.0. W3C Recommendation: http://www.w3.org/TR/xslt, Nov. 1999.