Declarative Transformation for Object-Oriented Models

Keith Duddy, Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel

CRC for Enterprise Distributed Systems Technology (DSTC)

Level 7, General Purpose South

The University of Queensland

QLD 4072 Australia

Phone: +617 3365 4310

Fax: +617 3365 4311

Email: {dud,agerber,lawley,kerry,steel}@dstc.edu.au

Declarative Transformation for Object-Oriented Models

ABSTRACT

This chapter provides a context and motivation for a language to describe transformations of models within an object-oriented framework. The requirements for such a language are given, and then an object-oriented model of the language's abstract syntax is provided that meets these requirements. A concrete syntax is introduced along with some example transformations. Finally we discuss the tools required to use the language within a model-driven software engineering paradigm. The authors aim to demonstrate the principles of model transformation within an object-oriented framework, and show how this can be applied to the development of software systems.

INTRODUCTION

In Model-Driven Architecture - A Technical Perspective (*2001*) the Object Management Group (OMG) describes an approach to enterprise distributed system development that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. The MDA™ approach envisions mappings from Platform Independent Models (PIMs) to one or more Platform Specific Models (PSMs).

The potential benefits of such an approach are obvious: support for system evolution, high-level models that truly represent and document the implemented system, support for integration and interoperability, and the ability to migrate to new platforms and technologies as they become available.

While technologies such as the Meta Object Facility (*MOF v1.3.1, 2001*) and the Unified Modelling Language (*UML, 2001*) are well-established foundations on which to build PIMs and PSMs, there is as yet no well-established foundation suitable for describing how we take an instance of a PIM and transform it to produce an instance of a PSM.

In addressing this gap, our focus is on model-to-model transformations and not on model-to-text transformations. The latter come in to 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 (*Hearnden & Raymond, 2002*) and we deal briefly with such concrete syntax issues late in the chapter.

This chapter focuses on a particular program transformation language, designed specifically for use with object-oriented models and programming languages. We provide an overview of the general problem of software model transformation and survey some technologies that address this space. The technology we then describe is designed to satisfy a set of identified requirements and is illustrated with a variety of example transformation specifications. We pay particular attention to how its features easily handle complex transformations and enable modular, composable, and extendable transformation rules without imposing an undue burden on the writer of the rules.

TRANSFORMATION FOR ENTERPRISE DISTRIBUTED SYSTEMS DEVELOPMENT

In order to build and maintain the IT systems supporting large-scale enterprise distributed systems efficiently, descriptions of these systems at the domain level need to be

automatically transformed into components, code, and configurations. The use of a transformation language designed to support the features of modern object-oriented specification and implementation technologies leads to more flexible, maintainable, and robust transformations than present ad-hoc approaches.

The Meta Object Facility (MOF) is a technology specification standardised by the OMG in 1997. It provides an object-oriented framework for the specification of the abstract syntax of modelling languages. Space limitations do not permit a detailed description of MOF features, however one can think of MOF models as corresponding to a slightly simplified UML Class Diagram.

The benefits of using this facility for the specification of languages such as the Unified Modelling Language (UML™) are that there are standard mechanisms for automatically deriving:

- a set of interfaces in CORBA IDL or Java for programmatic access to object model repositories,
- a concrete syntax based on XML DTDs and/or schemas known as XML Model Interchange (XMI), and
- a customisable human-usable textual notation or HUTN (*Human-Usable Textual Notation, 2002*) for representing model instances.

However, to date, the common MOF foundation of OMG languages such as UML, the Common Warehouse Metamodel (CWM™) 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 MOF 2.0 Queries/Views/Transformations RFP (*2003*), known as QVT for short. 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 to the target models
- express all new languages as MOF models

In developing our response to the QVT RFP, the authors considered a number of alternative approaches (*Gerber, Lawley, Raymond, Steel & Wood, 2002*). The results, along with a review of other submissions to the QVT RFP, are summarised below.

Chapter 13 of the OMG's Common Warehouse Metamodel Specification (*2001*) defines a model for describing Transformations. It supports the concepts of both black-box and white-box transformations. Black-box transformations only associate source and target elements without describing how one is obtained from the other. White-box transformations, however, describe fine-grained links between source and target elements via the *Transformation* element's association to a *ProcedureExpression*. Unfortunately, because it is a generic model and re-uses concepts from UML, a *ProcedureExpression* can be expressed in any language capable of taking the source element and producing the target element. Thus CWM offers no actual mechanism for implementing transformations, merely a model for describing the existence of specific mappings for specific model instances.

Varró and Gyapay (*2000*) and Varró, Varraó & Pataricza (*2002*) describe a system for model transformation based on Graph Transformations (*Andries et al, 1999*). In their approach, a transformation consists of a set of rules combined using a number of operators such as sequence, transitive closure, and repeated application. Each rule identifies before and after sub-graphs, where each sub-graph may refer to source and target model elements and associations between them (introduced by the transformation).

This style of approach to model transformation introduces non-determinism in the rule selection, and in the sub-graph selection when applying a rule.

Additionally, since rules are applied in a sequence, thus resulting in a series of state changes, one needs to be very careful about order of rule application and repeated rule application to ensure termination of the transformation process. A common technique is to delete elements from the source model as they are transformed, but this requires that an

element is transformed with a single rule rather than allowing multiple rules to address different aspects of the transformation.

Peltier, Ziserman & Bezevin (*2000*) and later, Peltier, Bezevin & Guillaume (*2001*), and Alcatel, Softeam, Thales & TNI-Valiosys (*2003*) in their QVT RFP response propose that transformation rules are best expressed at the model level, and that they should then be translated into a set of rules that operate on the concrete representations of model instances. As such, they propose MOF as the common meta-model for representing models, XMI as the concrete expression of model instances, and XSLT as the transformation tool to operate on these concrete instances.

Their rules have a mix of both procedural and declarative styles that is in part due to the fact that a given rule may only define a single target element per source element and that target element construction is explicit. They also require explicit ordering of the execution of rules.

XSLT (*W3C, 1999*) is explicitly designed for the transformation of XML documents and, through the XMI specification, all MOF models have an XML-based expression. However, being document-driven, XSLT is limited to source-driven transformations, and only provides for explicit object (or element) creation. Additionally, the XSLT syntax is both verbose and baroque, as it is based on XML. These reasons make it wholly unsuitable as an expressive model transformation language.

The UML 1.4 specification introduces a new Action Semantics language (ASL), which has also been proposed by both Kennedy Carter (*2003*) and Tata Consultancy Services (*2003*) as a model transformation language. The ASL provides a number of low-level constructs that can be composed to provide specifications of actions. However, in order to

be fully expressive for describing actions, ASL is defined at a very low level of abstraction, lower than appropriate for model transformation. Furthermore, it is an imperative language, so does not provide for implicit creation, or unordered rule evaluation.

Codagen Technologies (*2003*) propose an extension to XQuery and XPATH for selecting/querying MOF models, and a procedural templating language (MTDL) for constructing target model elements. In contrast, Interactive Objects Software GmbH & Project Technology (*2003*) propose the use of OCL 2.0 (*2003*) for querying/selecting source model elements and a declarative language for constructing target model elements. Because there can only be a single creation rule that gives rise to a target model element, this proposal also effectively uses explicit object creation which we believe is an unfavourable approach to building scalable and re-usable transformation definitions.

Finally, Compuware & Sun Microsystems (*2003*) propose a language that is based on OCL 2.0 for queries/selection, is declarative, uses implicit creation, and results in links between target model elements and the source model elements that lead to their creation. However, the language is defined as a minimal subtyping of the MOF meta-model and consequently provides little in the way of structuring mechanisms to guide the user in writing transformation definitions.

DESIGN REQUIREMENTS FOR OO-BASED TRANSFORMATION

Consider the problem of representing data described by an object-oriented class diagram in a relational database. Figure 1 shows possible models of the concepts involved in these two domains.

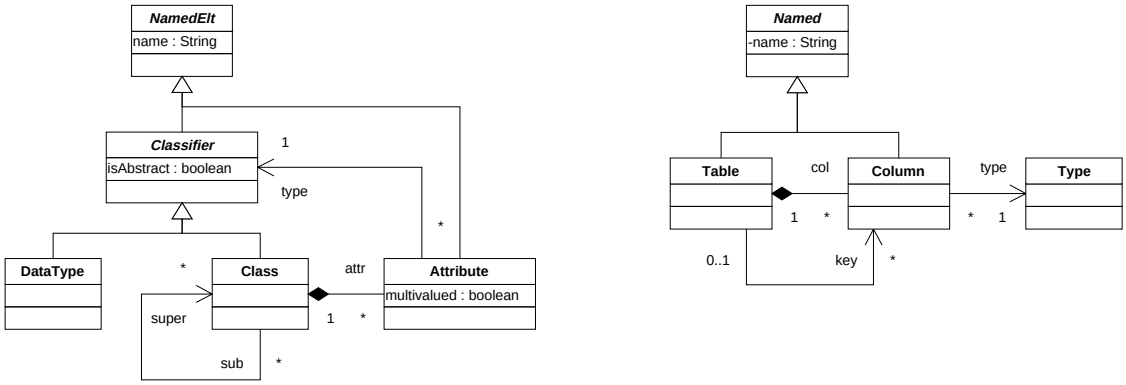

Figure 1 Simple source and target models

This is an interesting transformation problem because the two models are familiar to many people, and they are not so similar that a transformation is essentially one-to-one. Additionally, there are simple alternative transformations possible that result in different performance characteristics that we can also explore.

Let us consider what such a transformation might involve. Firstly, we might expect each Class to have a corresponding Table, each DataType to have a corresponding Type, and each Attribute of each Class to have a corresponding Column in the corresponding Table. Since our class model doesn't contain any information about the combination of attributes (if any) that constitute a key, we will also need a Column per Table to represent an *object-id*.

However, note that an Attribute may be multi-valued, so it would be rather inefficient to store all the other attributes multiple times in order to store the multi-valued attribute. Instead, we want a separate table for each Column corresponding to a multi-valued Attribute along with a Column for the object-id so we can perform the requisite joins.

If an Attribute's type is a DataType, then the corresponding Column's type would be the Type corresponding to the DataType, but if the Attribute's type is a Class, then the

corresponding Column would need to be a foreign key, so its type would be that of the (primary) key of the referenced table which is the object-id Column.

So far, describing the transformation has been relatively straightforward. Things get more interesting once we consider that one Class can subtype another Class. As we have described the mapping so far, an instance of a subclass ends up with its attributes split between several tables. An alternative transformation involves creating a Column for every Attribute owned by a Class and for every Attribute owned by a supertype of the Class.

Below we show, step by step, how one would express the transformations described above in our language. However, before we look at these details we reflect on the high-level natural-language description of the transformation we have just outlined in order to motivate the design of our language.

The first thing to notice is that the description above is naturally declarative; it describes the things we want to generate and the reasons why we want to generate them, but it does not say how or when to create them, not does it involve a description of traversing either the source or the target models. Additionally, while there are natural dependencies between parts of the description, there is no explicit statement of ‘‘do this then do that”; rule ordering is implicit, not explicit. Since the declarative style is a natural way to communicate a transformation description in a human language, we believe it is also suitable, indeed preferable, for a formal language for specifying transformations to be declarative.

A set of functional and usability requirements for a transformation language has been developed by Gerber, Lawley, Raymond, Steel & Wood *(2002*). A detailed list of these is presented in our response (*DSTC, IBM & CBOP, 2003*) to the OMG's QVT RFP.

The major functional requirements are as follows. A model-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);
- filter the set of matched elements or tuples based on associations, attribute values, and other context;
- match both collections of elements not just individual elements. For example, we may need to count the number of Attributes a Class has;
- establish named relationships between source and target model elements. These relationships can then be used for maintaining traceability information;
- specify ordering constraints (of ordered multi-valued attributes or ordered association links), either when matching source elements or producing target elements;
- handle recursive structure with arbitrary levels of nesting. For example, to deal with the subclassing association in our example Class model;
- match and create elements at different meta-levels;
- support both multiple source extents and multiple target extents.

In addition, the following non-functional requirements, identified for readability and expressiveness concerns, require that:

- there is no requirement to explicitly specify the application order of the rules, and all rules are matched against all relevant 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;
- a single target element can be defined by multiple rules. That is, different rules can provide values for different attributes of the same object;
- patterns can be defined and rules are able to be grouped naturally for readability and modularity;
- embedding of conditions and expressions in the language is explicit and seamless;
- transformation rules are composable.

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

- **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, or the various parts of the object-relational transformation described above.

Indeed, aspect-driven transformations are a major reason why we favour implicit (rather than explicit) creation of target objects, since 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.

A DECLARATIVE OBJECT-ORIENTED TRANSFORMATION LANGUAGE

We describe a declarative object-oriented transformation environment that satisfies the requirements described in the previous section. We present both a formal model for transformations and a concrete syntax and illustrate the transformation language through a series of simple examples.

This section presents the transformation language that we have designed to address the problems faced when realising the MDA, by illustrating how the language would be used to solve the object-relational mapping problem at hand.

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

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, other rules are able to match elements based on the tracking relationship independently of which rules were applied or how the target elements were 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.

Establishing and maintaining Tracking relationships is also essential for supporting round-trip development and the incremental propagation of source-model updates to through the transformation to the target model(s).

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.

The MOF model diagram in Figure 2 represents our current proposed Transformation model (*DSTC, IBM & CBOP, 2002*). The important classes are explained below.

The lower part of Figure 2 is an expression language metamodel constructed specifically for identifying MOF model elements in patterns and rules. Its two main abstract metaclasses are *Term* and *Expression.* Terms are evaluated to either true or false with respect to the models supplied for transformation. An Expression represents some kind of value referenced by a Term. VarUse is a particular kind of Expression that represents the binding of a value to a variable. Variables in the language are dynamically typed, and a *Var* slot may bind to any valid MOF type.

The top left part of Figure 2 shows the main structuring parts of the model. *TRule* represents a transformation rule, which is a statement that for all model elements that can be bound to the Vars of the TRule's *src* Term such that this Term evaluates to true the

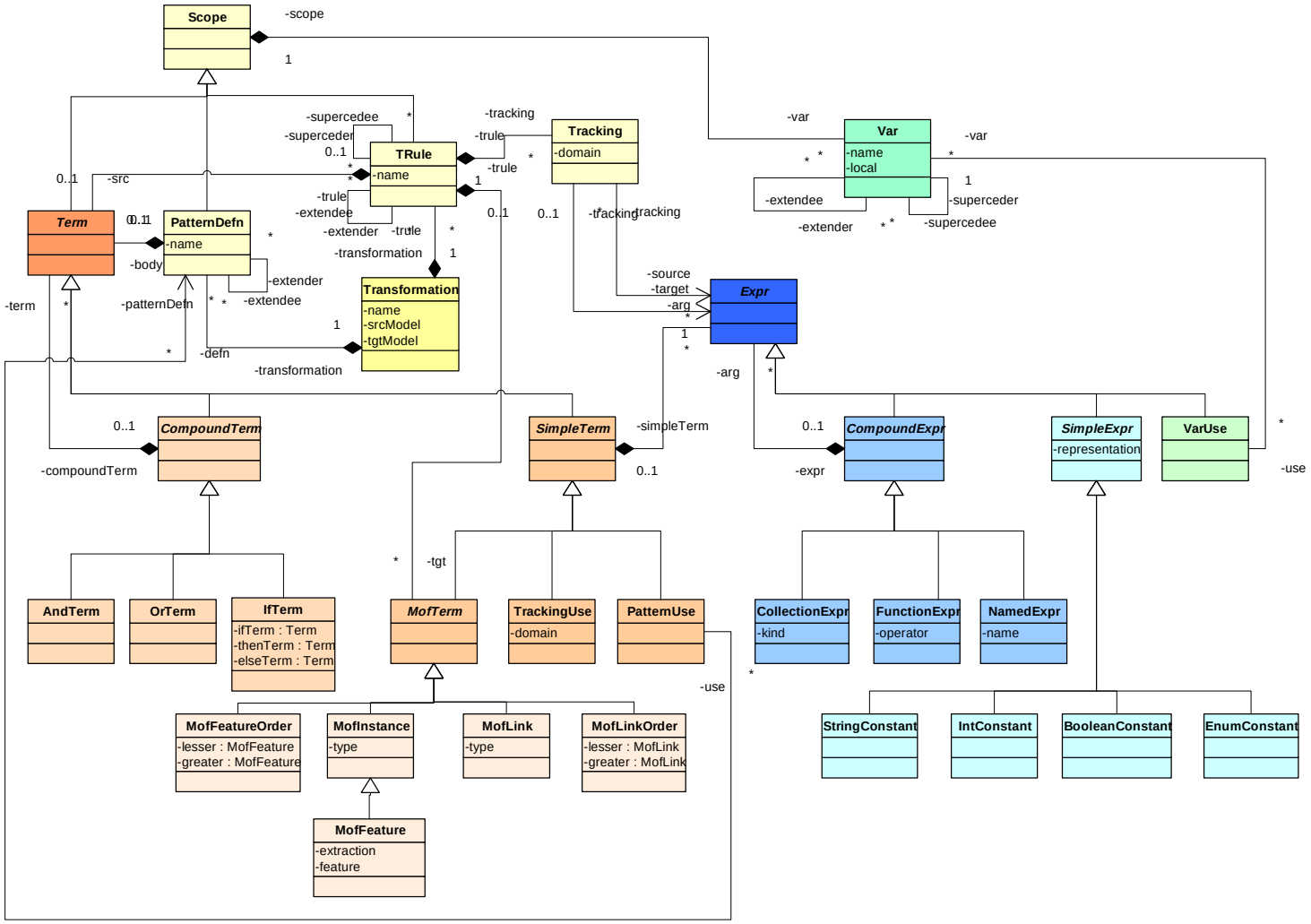

Figure 2 Transformation Model

TRule's *tgt* Terms must also evaluate to true. Generally, this involves creating new MOF model elements in the target model(s) and setting their attributes. In order to know when and how many model elements to create, *Trackings* are used. A Tracking is a statement of a functional dependency between a set of source model elements and a target model element. This allows several independent rules to require a target object to exist without needing to explicitly coordinate which rule is responsible for actually creating the instance.

Transformation rules may be related to other transformation rules either or both of the following two ways.

- A rule that *extends* another rule augments its source matching term with the source term of the extended rule.
- A rule that *supersedes* another rule restricts the source matching term of the superseded rule with the negation of its source term.

Rule extending and superseding allow for Transformations, which supply a namespace for rules and pattern definitions, to be reused and specialised. In particular, they allow rules to be written simply for the general case, and then superseded for some special or exceptional cases.

Concrete Syntax by Example

We now present the transformation described earlier using an SQL-like concrete syntax one rule at a time, based on one rule per concept-to-concept mapping. We then link these together and fill in details to provide the total transformation from our OO model to our Relational model.

The first key element of our transformation is that a Class will be transformed into a Table with an *object-id* Column, so this becomes our first rule. We also want to make sure that we preserve a tracking relationship between the table we create and the class from which we create it. The next major mapping, from an Attribute to a Column, is similar, as is the rule for DataTypes. As such, we start with the following simple rules:

```
RULE class2table
  FORALL Class Cls
  MAKE Table Tbl, Column idCol,
       idCol.name="id", Col.owner=Tbl
  LINKING Cls to Tbl by c2t;


RULE attr2col
  FORALL Attribute Att
  MAKE Column Col
  LINKING Att to Col by a2c;
```

Both Class and Attribute are subtypes of NamedElt, and we want their names to be mapped to the names of their corresponding Tables and Columns. We can make sure we have the right Class-Table or Attribute-Column pair by looking up the tracking relationships we established earlier. We can use then write a rule from an OO NamedElt to a Relational Named like this:

```
RULE named2named
  FORALL NamedElt n1
  WHERE c2t LINKS n1 to n2
     OR a2c LINKS n1 to n2
  MAKE Named n2,
       n2.name = n1.name;
```

We see here that trackings can be used to tie rules together, thus giving us the ability to express rules as fine-grained mappings rather than having to write complex, coarse-grained rules.

However, further inspection of our class diagram reveals that DataType names must also be mapped. Rather than adding another OR clause to our rule, we introduce generalization to our tracking relationships. So, we make another tracking relationship that stands as a superset of the two we have already used, and look up the parent tracking rather than alternating over the children, like so:

```
TRACKING c2t ISA named2named;
TRACKING a2c ISA named2named;

RULE named2named
  FORALL NamedElt n1
  WHERE named2named LINKS n1 to n2
  MAKE Named n2, n2.name=n1.name;
```

Next, we need to make sure that the column resulting from the transformation of an attribute will be contained by the appropriate table, i.e., the table resulting from the transformation of the attribute's containing class. We do this by again looking up the tracking relationships established in our earlier rules. This gives us the following rule:

```
RULE clsAttr2tblCol
  FORALL Attribute Att, Class Cls
  WHERE Att.owner = Cls
    AND c2t LINKS Cls to Tbl
    AND a2c LINKS Att to Col
  MAKE Table Tbl, Column Col,
       Col.owner = Tbl;
```

We already have a rule for transforming Attributes. However, we now find that we wish to transform multi-valued attributes differently. The values of a multi-valued attribute will be stored in a separate table, with one column for the values and one column for the Class's object-id.

This new rule for attributes will need to match a subset of the cases that were true for the previous rule, and we can reuse the earlier Attribute rule's matching pattern by using rule extension. However, we also want to indicate that the earlier Attribute rule should not run when this new Attribute rule runs, and we can do this using rule supersession.

So now we have a rule for transforming Attributes to Columns, and another for linking the Column to a Table. However, we find that we want to map multi-valued attributes differently. The Column for a multi-valued Attribute should instead have its own Table, with another Column to link back to the key in the main Table for the Class. Therefore, we make a new rule that will supersede the rule that puts Columns in Tables, and link the Attribute's Column to a new Table with a new key Column.

```
RULE clsMultiAttr2tblCol extends
     and supersedes clsAttr2tblCol
  FORALL Attribute Att, Class Cls,
  WHERE Att.multivalued = TRUE
  MAKE Table AttTbl, Column KeyCol,
       Column AttCol,
       KeyCol.owner=Tbl, AttCol.owner=Tbl,
       KeyCol.name = Cls.name
  LINKING Att to AttTbl by mva2t;
```

Having created and placed these Columns, we need to give them an appropriate type. So we need rules for mapping DataTypes to Types, and for assigning the appropriate Type to a Column. The latter case requires two rules, since an Attribute with a Class type is typed

for a key value, but an Attribute with a DataType type is mapped for the corresponding Type.

```
TRACKING dt2t ISA named2named

RULE datatype2type
  FORALL DataType Dt
  MAKE Type T
  LINKING Dt to T by dt2t;


RULE atype2ctype
  FORALL Attribute Att, DataType Dt
  WHERE a2c LINKS Att to Col
    AND dt2t LINKS Dt to T
    AND Att.type = Dt
  MAKE Column Col, Type T,
       Col.type = T;


RULE actype2ctype
  FORALL Attribute Att, Class C
  WHERE Att.type = C
    AND a2c LINKS Att to Col
  MAKE Column Col, Type T,
       Col.type = T, T.name = "String";
```

The other approach to mapping attributes, as described above, is to include inherited attributes as columns in the tables of subclasses. To do this, we need to define a recursive Pattern for finding the inherited attributes of a class.

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

Having defined this pattern, we can make a rule for creating a column for each inherited attribute. To handle the linking of these columns to their tables, we need to change the Attribute to Column tracking to include Class as a source, by modifying the earlier rules, `attr2col` and `clsAttr2tblCol`. The new rule, as well as these modified rules, is below:

```
RULE superattr2col
  FORALL Attribute Att, Class Cls
  WHERE hasAttr(Cls, Att)
    AND c2t LINKS Cls to Tbl
  MAKE Table Tbl, Column Col
  LINKING Att, Cls to Col by a2c;


RULE attr2col
  FORALL Attribute Att, Class C
  WHERE Att.owner = C
  MAKE Column Col
  LINKING Att, Cls to Col by a2c;


RULE clsAttr2tblCol
  FORALL Attribute Att, Class Cls
  WHERE c2t LINKS Cls to Tbl
  AND a2c LINKS Att, Cls to Col
  MAKE Table Tbl, Column Col,
       Col.owner = Tbl;
```

While some people may be daunted by the idea of specifying transformations using a declarative language, we believe that the use of an appropriate concrete syntax such as the SQL-like one introduced above will allay their fears and allow them to reap the benefits of not needing to worry about rule order and explicit object creation that a declarative language affords. Additionally, an equivalent graphical syntax would be a useful addition for simple transformations, although our experiences indicate that more complicated transformations are better handled textually.

ADVANCED TRANSFORMATIONS

As with expert systems, a substantial transformation embodies a significant investment in capturing domain knowledge and therefore the careful organisation and structuring of the transformation will aid its long-term maintenance and evolution.

Several features of the transformation language described in this chapter are key to supporting both re-use and maintenance of transformation definitions. These features are the *supersedes* and *extends* relationships, and dynamic typing of variables.

Duddy, Gerber, Lawley, Raymond & Steel (*2003*) describe in detail how superseding and extending can be used in the context of a transformation to an Entity Java Bean (EJB) model. Specifically, they show how a mapping that results in remote access to EntityBeans can be modified to instead employ the Session Façade pattern (*Brown, 2001*) using a SessionBean that delegates methods to local EntityBeans. One could also use an extra source model as a parameterisation, or *marking model*, to provide finer grain control over which rules are applied to which source model elements.

Dynamic typing both simplifies rule writing; if an object bound to a variable does not have the attribute or reference mentioned in a term, then the term simply evaluates to

false rather than requiring explicit runtime type introspection and narrowing (down-casting). Dynamic typing also allows meta-transformations and libraries of transformations matching common structural patterns to be easily developed without the use of cumbersome reflective methods. Such transformations may, for example, deal with mapping names in nested namespaces to structured names in a single flat namespace.

THE TRANSFORMATION TOOL-SET

The transformation language described previously is just one piece of the automated development environment. Other tools that deal with the input and output of concrete textual representations of models, and the graphical visualisation of models and transformations are vital parts of an MDA tool-set.

The OMG has developed the XML Metadata Interchange (XMI) Format (*2002; XMI Production of XML Schema, 2001*) which defines rules for producing both an XML DTD and an XML Schema from a model, and a set of rules for transferring data between conformant XMI documents and MOF-compliant repositories. The main use of XMI is for interchange of models and model instances between tools, and for storage.

The result of the OMG's RFP for a Human Usable Textual Notation standard (*1999*) was the adoption of a more generic approach for automatically creating a human-friendly language for an arbitrary MOF model. By exploiting the inherent structure in a MOF model (containment, association cardinalities, etc), and allowing for some on-the-fly customisation (e.g., for default values) it is possible to fully-automate the generation of both parsers and printers of the HUTN language for a given MOF model.

Most commonly, the ultimate goal of a transformation process is to generate code or some other textual representation of a system. While it would be reasonable to realise this

goal by transforming an abstract model to a model corresponding to the concrete syntax of the target language (Java, Cobol, HTML, etc), this task is particular enough to warrant and benefit from a specialised tool.

The motivation of the AntiYacc tool is to provide a simple means to render the content of a MOF-based repository in a textual form corresponding to some specified syntax. It is capable of producing text that conforms to an arbitrary user-supplied EBNF grammar. Just as a parser generator such as Yacc (*Johnson, 1974*) takes a set of grammar rules decorated with code fragments to, typically, construct a parse tree from a stream of lexical tokens, AntiYacc takes a set of grammar rules decorated with code fragments to construct a stream of lexical tokens from a traversal of a graph of MOF objects. An AntiYacc specification also includes lexical rules that convert the token stream into text, typically addressing issues such as horizontal and vertical whitespace, and delimiters.

Finally, it would often be very useful to be able to visualise or edit a graphical representation of the models being transformed. However, since much of the time their metamodels may be purpose-designed and therefore have no standard graphical representation let alone a tool to display/edit the model, it would be extremely useful to be able to generate such a tool in a manner analogous to the HUTN approach. That is, to employ a set of standard, simple visual concepts (box, line, label, containment, proximity, etc) to render a given model. Such a tool is currently under development by the authors.

CONCLUSION

In this chapter we have introduced the problem of model-to-model transformation for the purpose of building distributed systems from high-level models describing the system to be built in platform independent terms then generating the system implementation for a

particular, technology specific, platform. This is the vision embodied in the OMG's Model Driven Architecture (MDA).

We have described the functional and non-functional design requirements identified for a language suitable for writing transformation definitions and presented a language satisfying these requirements along with examples of a usable, familiar concrete syntax for the language. We have also briefly touched on issues relating to advanced transformations and mentioned a number of additional technologies required for dealing with textual and graphical forms of the models.

It should be noted that the transformation language presented here is evolving as we gain further experience and as a result of the OMG's RFP process. In particular, influenced by the Compuware/Sun submission, we are extending the concept of Trackings to more closely approximate a class model. Also, composition of transformations is essential for the use and extension of existing transformations. While there is no explicit mention of this in the language presented here, the ability to reference elements in one MOF model from another MOF model should be sufficient for simple composition of transformations. However, more sophisticated forms of composition such as producing a transformation that maps A to C from one that maps A to B and one that maps B to C or producing a transformation that merges A and B to produce C from the A to B and B to C transformations is the subject of future research.

Additionally, the transformations discussed in this chapter have generally dealt with transformation in a single direction, from model A to model B. Another use of a declarative transformation language, and one in which the declarative nature offers an advantage over imperative alternatives, is in the application of a single transformation to

perform transformations both from A to B, and back again. This is another aspect that is currently under investigation by both the authors and the larger OMG community.

REFERENCES

A Human-Usable Textual Notation for the UML Profile for EDOC: Request for Proposal (1999). OMG Document ad/99-03-12.

Alcatel, Softeam, Thales & TNI-Valiosys (2003). Response to the MOF 2.0 Queries/Views/Transformations RFP. OMG document ad/03-03-35.

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H., Kuske, S., Pump, D., Schurr, A., & Taentzer, G. (1999). Graph transformation for specification and programming. Science of Computer Programming, 34(1), 1-54.

Boldsoft, Rational Software Corporation, IONA & Adaptive Ltd (2003). Response to the UML 2.0 OCL RFP. OMG Document ad/2003-01-02

Brown, K. (2001). Rules and Patterns for Session Facades. IBM's WebSphere Developer Domain. Retrieved from http://www.boulder.ibm.com/wsdd/library/techarticles/0106_brown/sessionfacades.html

Codagen Technologies Corporation (2003). MOF 2.0 Query/Views/Transformations. OMG Document ad/2003-03-23.

Common Warehouse Metamodel (CWM) Specification (2001). OMG Documents ad/01-02-01, ad/01-02-02, ad/01-02-03.

Compuware Corporation & Sun Microsystems (2003). XMOF Queries Views and Transformations on Models using MOF, OCL and Patterns. OMG Document ad/2003-03-24

DSTC, IBM, & CBOP (2003). MOF Query/Views/Transformations Initial Submission. OMG Document ad/2003-02-03.

Gerber, A., Lawley, M., Raymond, K., Steel, J., & Wood, A (2002). Transformation: The Missing Link of MDA. In Proceedings of the First International Conference on Graph Transformation (ICGT'02), Barcelona, Spain. LNCS 2505, 90-105.

Human-Usable Textual Notation (HUTN) Specification (2002). OMG Document ptc/02-12-01.

Johnson, S. (1974). YACC - Yet Another Compiler Compiler. CSTR 32, Bell Laboratories.

Model Driven Architecture - A Technical Perspective (2001). OMG Document ormsc/01-07-01.

MOF 2.0 Queries/Views/Transformations: Request for Proposal (2002). OMG Document ad/02-04-10.

Meta Object Facility (MOF) v1.3.1 (2001). OMG Document: formal/01-11-02.

Peltier, M., Bezevin, J., & Guillaume, G., (2001). MTRANS: A general framework, based on XSLT, for model transformations. In Proceedings of the Workshop on Transformations in UML, Genova, Italy.

Peltier, M., Ziserman, F., & Bezevin, J., (2000). On levels of model transformation. In XML Europe 2000, Paris, France.

Unified Modelling Language v1.4 (2001). OMG Document: formal/01-09-67.

Varró, D., Varraó, G., & Pataricza, A. (2002). Designing the Automatic Transformation of Visual Languages. Science of Computer Programming 44(2), 205-227.

Varró, D. & Gyapay, S. (2000). Automatic Algorithm Generation for Visual Control Structures. Retrieved Feb 8, 2001 from http://www.inf.mit.bme.hu/FTSRG/Publications/TR-12-2000.pdf

XMI Production of XML Schema (2001). OMG Document ptc/2001-12-03.

XML Metadata Interchange (XMI) Version 1.2 (2002). OMG Document formal/2002-01-02.

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

Hearnden, D. & Raymond, K. (2002). Anti-Yacc: MOF-to-text. In Proceedings of the Sixth IEEE International Enterprise Distributed Object Computing Conference, Lausanne, Switzerland. IEEE.