Typing Relationships in MDA

Jim Steel, Jean-Marc Jézéquel

IRISA, Campus universitaire Beaulieu
35042 Rennes cedex France
{jsteel, jezequel}@irisa.fr

Abstract

As the OMG's Model-Driven Architecture matures from a field of research and specification into one of system engineering, it faces all of the challenges endemic to the practice. Among the foremost of these is the need to support re-use of its artifacts as they evolve. As systems begin to be built upon the basic ideas of models interrelated by model transformations, it will become increasingly important to have appropriate definitions for the typing relationships that can exist between models and metamodels, since it is these definitions that will determine the substitutability characteristics of these artifacts in model transformations. This paper seeks to enumerate a number of these relationships, to provide initial characterisations of them in terms of their significance to the goal of re-use in MDA.

1 Introduction

The Model-Driven Architecture[14] uses model transformations to describe (and probably to enforce) the relationships between models, as described by metamodels.

However, the metamodels (and by consequence the models) that are used in model-driven systems are diverse, they evolve, and they are frequently overlapping in their domains, even when they are not in their definitions. For all of these reasons, when we write model transformations, we want them to apply over a wide range of models. This feature, much studied in software engineering, is known as re-use.

The languages currently used to write model transformations, including but not limited to those proposed in [5, 1, 8] are dramatically diverse, as illustrated in [7, 4, 6], a situation that is unlikely to be completely remedied by the eventual arrival of an adopted specification for their definition[11]. All these approaches build on the notion of model element, seen as an instance of a specific class of a given meta-model MM1. A model M is made of model elements linked between themselves to form an arbitrary complex graph, conforming to the meta-model MM1. Such a model M can be provided as an input parameter to a transformation T.

In this paper we explore various cases of relationships between model elements, models and meta-model, in order to discuss under which conditions a transformation T can be safely applied to a model M. We start by presenting three main motivations for this work before entering into the details of model typing and conformance.

Clearly, the study of type systems is not a new one. In particular, much research has been conducted and validated within the functional languages community, and particularly the ML languages. Closer in heritage to MDA is the field of object-oriented systems. Some of the earliest work was by Liskov [9], in the form of the much-referenced substitutability principle. This was built upon formally by, notably, Cardelli & Wegner in [2], and further by Castagna in [3], as extensions of lambda

calculus. Also relevant are the typing strategies that have been implemented in O-O languages such as Java, Eiffel, and dynamic languages such as Python and Ruby.

However, it is important to note that their important differences between the underlying data structures of object- and model-based systems. Most significant of these is the linking of fields/properties as opposites (represented in earlier versions of MOF as associations). This feature means that models form much more tightly-coupled graphs than objects, which could often be treated as atoms in isolation. In particular, this graph-ness has important implications for typing relationships, in that the the relationship between a model element and a class will generally involve the analysis of the types of the other model elements and classes in the respective graphs. It also introduces the need to deal with the inevitable circular dependencies that arise in evaluating type relationships across these graphs.

2 Motivation

We argue that the need for a flexible mechanism for re-use in model-driven engineering comes from the inevitable separation of the metamodels used to describe models. There are a number of reasons for this separation, a number of which are detailed here.

2.1 Physical vs Logical metamodel

For many reasons, it is not always possible to ensure that all models of the same notional metamodel are defined in terms of the same physical definition of that metamodel. For example, the physical metamodel may be in a serialized form, such as XMI, whereas a given transformation requires it in an object form, or vice versa. Ideally, these issues should not affect the ability of a transformation to apply equally to models whose metamodels are logically equivalent but physically distinct. This issue is also highlighted by the increasing application of models in the design and implementation of distributed systems, on such platforms as CORBA and web services.

2.2 Extension

There are number of mechanisms provided for extension of metamodels. In the 1.x versions of MOF[10], these included package import, extension and clustering. In MOF 2.0, there are additional mechanisms such as package merge and package combine. These relationships are established at the package level, and have varying implications for the corresponding relationships at the class level. While a full discussion of these relationships is not the domain of this paper, we will briefly analyse the package combine mechanism, since it is the most challenging form of metamodel extension in terms of re-use.

Package combine, as defined in the UML 2 Infrastructure submission, is a form of package merge, whereby all classes in the original package are copied into the new package. Any classes defined with the same names are "merged": the new set of features for the class is the union of the sets of each of the original classes. After application of the package merge, all relationships between the packages, and between the classes therein, are removed, so that the new package can be used independently of the original.

Strictly speaking, package combine is not a relationship, but an operation, but this does not diminish its usefulness in modelling. Moreover, its use should not prohibit a transformation defined in terms of the original metamodel from working with the extended one.

2.3 Evolution

Metamodels evolve over time, as do the models that they describe. In general, the problem of model evolution and versioning is very complicated, and is still the subject of active research.

However, simple changes such as the addition of an extra attribute to a class should not impact on the ability of a transformation defined in terms of the original version of the metamodel to work with

the modified version. This should be possible regardless of whether the heritage of the metamodel, in terms of version history, has been properly preserved or not[12].

3 Relationships

We present here a number of relationships that can exist between classes and model elements, that might be used in typing models.

For the purposes of this section, we use definitions for model element, model, and metamodel as commonly understood in the OMG, and as mentioned briefly in the introduction. We characterise relationships using a number of properties, such as their normal modes of interaction (are they generally requested, or affirmed, and are they qualified with respect to a certain domain), and their relationships to one another in terms of supersets and subsets.

3.1 Instantiation

Model elements in model-driven systems are created from a class, be it directly or using a factory. In this way, the model element is given slots for each of the properties of the class and, typically, will delegate the semantics of an operation to a method attached to the method definition.

This relationship is the basic building block for the definition of other, more flexible typing relationships. Specifically, it defines the "provided type" that is used in comparison to the required type for the purposes of type-checking. Of itself, it offers little by way of flexibility, and addresses none of the issues raised in section 2.

This relationship is typically requested, rather than affirmed, and is always absolute, never qualified, since there can be only one correct response.

3.2 Reflection

Reflection is the process of asking a model element for a description of itself. More specifically, it involves learning what are the operations and properties provided/supported by the model element including, by extent, their types, and thus potentially extending over a large graph of types reachable from that of the original.

It should be noted that the type system in MDA, given by MOF, has no separation between between types and classes, and thus reflection provides a class. This includes details, such as the class name, that may not be relevant to the definition of reflection given here.

In theory, reflection is slightly more flexible than "instantiated by", since one may have multiple metamodels that equally describe the capabilities of the model element, through techniques such as collapsing subclasses. However, it is unable to handle substitutability problems such as instances of subclasses, or of structural subtypes.

This relationship, like "instantiated by", is typically requested, rather affirmed, and is usually absolute, although in theory could be qualified, such as by policies for collapsing subclasses. The "instantiated by" relationship is a subset of the reflection relationship.

3.3 Conformance By Inheritance

Inheritance, also known as generalization/specialization is an explicit relationship between classes dictating, among other things, that all features defined on the superclass will be available on instances of the subclass. In this relationship, a model element is conformant to a class iff its instantiated class is the same as the required class, or is an explicit subclass (either directly or transitively) of the required class.

This is the same relationship as is commonly seen in programming languages such as Java, and is the most common relationship presently used in model transformation languages. Moreover, it is the relationship used by OCL[13] and thus, by association, MOF and UML.

It has the advantage that it is more flexible than either instantiation or reflection, since it allows for instances of subclasses to be accepted, in addition to those of the specified class. Also, since it is explicitly defined, it is efficient to compute and well-suited to static evaluation.

Conformance relationships are affirmed, rather than queried, and as such are typically absolute rather than qualified. This relationship is a superset of instantiation, but not of reflection.

3.4 Structural Conformance

Structural conformance bears some similarity to reflection, in that it deals with the set of features (operations and properties) that are supported by the model element. In this way, a model element is structurally conformant to any class that is a subtype of its instantiating class, where the definition of subtype is based on that defined by Cardelli & Wegner in [2]. In fact, since Cardelli & Wegner's definition is based on objects, some small extensions need to be made to apply it to the realm of models, such as treatment of multiplicities. We present a summary of the definition.

```
A class A is a subtype of a class B iff:
    ∀ property P of B, ∃ property P' of A, such that

       P'.name == P.name
       the type of P' is a sub-
       type of the type of P (covariance)
       the multiplicity of P' conforms to the mul-
       tiplicity of P
       P.isReadOnly == false im-
       plies P'.isReadOnly == false
    ∀ operation O of A, ∃ operation O' of B, such that

       the return type of O' is a sub-
       type of the return type of O (covariance)
       ∀ parameter R of O, ∃ parame-
       ter R' of O', such that

          the type of R is a sub-
          type of the type of R' (contravari-
          ance)
          the multiplicity of R con-
          forms to the multiplicity of R'
```

Structural conformance of classes is characterized by covariance with the types of properties and the return types of operations, and contravariance with the types of parameters to operations.

The multiplicities, consisting of cardinality ranges, orderedness and uniqueness, of the properties, operations and parameters of the classes must also be considered. The simplest, but most restrictive, approach is to consider only exactly equal multiplicities as conformant. Alternatively, one can impose a hierarchy, whereby ordered collections are a subtype of unordered ones (but not vice versa), and cardinality ranges are given a priority order such as [0..*, 0..1, 1..1], where each range is conformant of any range that follows it. Such an approach would cover 90% of cases, although for full coverage, a more sophisticated heuristic such as partial orders would be needed to handle other ranges such as 1..*.

In a general purpose programming language, the failure to consider the behaviour of the operations would mean that structural conformance falls short of true substitutability. However, it is important to remember that MOF is not a general purpose programming language. In fact, it bears more resemblance to signature languages such as java interfaces, C++ templates, or CORBA interfaces. As such, any consideration of operation behaviour, such as would be required in terms of Liskov's substitutability principle[9], is out of scope.

Like inheritance-based conformance, this relationship is affirmed rather than qualified, and is generally not qualified. Structural conformance is a superset of direct instantiation, reflection, and conformance by subclassing. (That is, conformance by subclassing implies structural conformance; a useful axiom for implementation.)

Structural conformance offers considerable advantages over conformance by subclassing in terms of flexibility. In particular, with respect to the motivations presented in Section 2, it is much better able to handle the issues of evolution and extension, including package merge. Its disadvantage is that it is significantly more intensive to evaluate, and is less amenable to static evaluation.

4 Future Work And Conclusions

The contrast between inheritance-based and structural conformance for models is clearly one of efficiency versus flexibility. To evaluate these criterion in more detail, we now propose to prototype the different approaches and apply them to various examples of model transformation.

To this end, we have developed a MOF repository using the Ruby programming language[15]. Ruby is notable for its flexible approach to typing, which is often described as "duck-typing" (if it walks like a duck, and it quacks like a duck, then it must be a duck). The lack of any real type-checking in Ruby makes it well-suited for the evaluation of different typing strategies.

A possible subsequent avenue for exploration is that of typing strategies within models themselves. The relationship between the type of a property and the type of a model element that might fill it, for example, is very similar to the relationship between a transformation specification and those models that might be permitted as input. Moreover, a number of transformation languages, such as [5] and [1] are based largely on the population of functional or relational models (sometimes called traceability models) as the determining factor for creating, deleting, or modifying model elements. As such, any approach to structural type conformance in these languages would need support for structural conformance within these models.

References

[1] David H. Akehurst and Stuart Kent. A relational approach to defining transformations in a metamodel. In *UML 2002 - The Unified Modeling Language, 5th International Conference, Proceedings*, pages 243–258, 2002.

[2] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):211–221, 1985.

[3] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Birkhäuser, 1997.

[4] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Anaheim, USA, October 2003.

[5] K. Duddy, A. Gerber, M.J. Lawley, K. Raymond, and J. Steel. Model transformation: A declarative, reusable patterns approach. In *Proc. 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2003*, pages 174–195, September 2003.

[6] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 query / views / transformations submissions and recommendations towards the final standard, August 2003. OMG Document: ad/03/08/02.

[7] A. Gerber, M.J. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In *Proc. 1st International Conference on Graph Transformation, ICGT'02*, volume 2505 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[8] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *Proc. Automated Software Engineering, ASE'99, Florida*, October 1999.

[9] B. H. Liskov and S. N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, April 1974.

[10] Object Management Group (OMG). Meta Object Facility (MOF) specification. OMG Document ad/97-08-14, September 1997.

[11] Object Management Group (OMG). MOF 2.0 Query/Views/Transformations RFP. OMG Document ad/2002-04-10, October 2002.

[12] Object Management Group (OMG). MOF 2.0 Versioning RFP. OMG Document ad/2002-06-23, June 2002.

[13] Object Management Group (OMG). The object constraint language (OCL), 2003. http://www.omg.org/docs/ptc/03-08-08.pdf.

[14] R. Soley and the OMG Staff. Model-Driven Architecture. OMG Document, November 2000.

[15] David Thomas and Andrew Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley Professional, 2000.