UML data models from an ORM perspective: Part 9

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

This paper first appeared in the June 1999 issue of the Journal of Conceptual Modeling*, published by InConcept.*

This paper is the ninth in a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 discussed historical background, language design criteria, object reference and single-valued attributes. Part 2 covered multi-valued attributes, basic constraints, and instantiation using UML object diagrams or ORM fact tables. Part 3 compared UML associations and related multiplicity constraints with ORM relationship types and related uniqueness, mandatory role and frequency constraints, as well as how associations may be instantiated. Part 4 contrasted ORM nesting, co-referencing and exclusion constraints with UML association classes, qualified associations, and xor-constraints respectively. Part 5 discussed subset and equality constraints. Part 6 discussed subtyping. Part 7 discussed value, ring and join constraints. Part 8 listed some recent updates to the UML standard, then discussed aggregation. Part 9 examines initial values and derived data in ORM and UML.

Initial values

The syntax of an attribute specification in UML includes six components as shown below. Square and curly brackets are used literally here as delimiters (not as BNF symbols to indicate optional components).

visibililty name [multiplicity] : type-expression = initial-value {property string}

If an attribute is displayed at all, its name is the only thing that must be shown. The visibility marker (+, #, – denote public, protected, and private respectively) is an implementation concern, and will be ignored in our discussion. Multiplicity has been discussed earlier and is specified for attributes in square brackets, e.g. [1..*]. For attributes, the default multiplicity is 1, i.e. [1..1]. The type expression indicates the domain on which the attribute is based (e.g. String, Date). Initial-value and property string declarations may

optionally be declared. Property strings may be used to specify changeability (see next article in this series). We now turn to a consideration of initial values.

An attribute may be assigned an *initial value* by including the value in the attribute's declaration after an equals sign (e.g. diskSize = 9; country = USA; priority = normal). The language in which the value is written is an implementation concern. In Figure 1, the nrColors attribute is based on a simple domain (e.g. PositiveInteger) and has been given an initial value of 1. The resolution attribute is based on a composite domain (e.g. PixelArea) and has been assigned an initial value of (640,480).

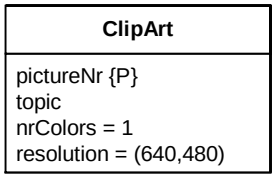


Figure 1: Attributes may be assigned initial values in UML

Unless over-ridden by another initialization procedure (e.g. a constructor), declared initial values are assigned when an object of that class is created. This is at least similar to the database notion of *default value*s, where during the insertion of a tuple an attribute may be assigned a predeclared default value if a value is not supplied by the user. However UML uses the term "default value" in other contexts only (e.g. template and operation parameters) [0], and some authors claim that default values are not part of UML models [0, p. 249]. The SQL standard treats **null** as a special instance of a default value, and this is supported in UML, since the standard notes that "a multiplicity of 0..1 provides for the possibility of null values: the absence of a value" [0, p. 3-41]. So an optional attribute in UML can be used to model a feature that will appear as a column with the default value of null, when mapped to a relational database. Presumably a multiplicity of [0..*] or [0..*n*] for any *n* > 1 also allows nulls for multi-valued attributes, even though an empty set could be used instead.

Currently, ORM does not provide explicit support for initial/default, values. However UML initial values and relational default values could be supported by allowing default values to be specified for ORM roles. At the meta-level, we add the fact type: Role has default- Value. At the external level, instances of this could be specified on a predicate properties sheet, or even entered on the schema diagram (e.g. by attaching an annotation such as d: *value* to the role, and preferably allowing this display to be toggled on/off). SQL default values are simple, so their source ORM roles need to be played by a simply identified object type. For example, the role played by NrColors in Figure 2 has been allocated a default value of 1. When mapped to SQL-92, this should add the declaration "default 1" to the column definition for ClipArt.nrColors.

To support the composite initial values allowed in UML, composite default values could be specified for ORM roles played by compositely identified object types (co-referenced or nested). When co-referencing involves at least two roles played by the same or compatible object types, an order is needed to disambiguate the meaning of the composite value. For example, in Figure 2 the role played by Resolution has been assigned

a default composite value of (640,480). To ensure that the 640 applies to the horizontal pixelcount and the 480 applies to the vertical pixelcount (rather than the other way round), this ordering needs to be applied to the defining roles of the external uniqueness constraint. In VisioModeler, this ordering is determined by the order in which the roles are selected when entering this constraint; although the display of this order is normally suppressed, the order can be displayed by right-clicking the constraint and choosing SelectRoleSequence from the pop-up menu.

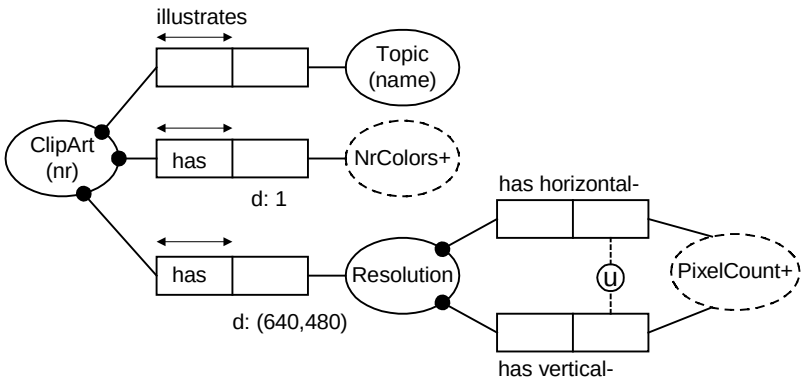


Figure 2: A possible extension to ORM to capture simple and composite default values

If all or most roles played by an object type have the same default, it may be useful to allow a default value to be specified for the object type itself. This could be supported in ORM by adding the meta-fact-type ObjectType has default- Value, and proving some notation for instantiating it (e.g. by an entry in the Object Type Properties sheet, or by annotating the object type ellipse with d: value). This corresponds to the default clause permitted in a create-domain statement in SQL-92. Note that an object-type default can always be expressed instead by role-based defaults, but not conversely (since the default may vary with the role).

Specification of default values does not cover all the cases that can arise with regard to default information in general. A detailed proposal for providing greater support for default information in ORM is discussed in [0], but this goes beyond the built-in support for defaults in either UML or SQL. Default information can be modeled informally by using a predicate name to convey this intention to a human. For example, we might specify default medium (e.g. 'CD', 'DVD', 'T') preferences for delivery of soft products (e.g. music, video, software) using the 1:n fact type: Medium is default preference for SoftProduct. In cases like this where default values overlap with actual values, we may also wish to classify instances of relevant fact types as actual or default (e.g. Shipment used Medium). For the typical case where the uniqueness constraint on the fact type spans *n*-1 roles, this can be achieved by including fact types to indicate the default status (e.g. Shipment was based on Choice {actual, default}), resulting in extra columns in the database to record the status. While this approach is generic, it requires the modeler and user to take full responsibility for distinguishing between actual and default values.

Derived data

In UML, derived elements (e.g. attributes, associations or association-roles) are indicated by prefixing their names with "/". Optionally, a *derivation rule* may be specified as well. The derivation rule can be expressed as a constraint or note, connected to the derived element by a dashed line. This line is actually shorthand for a dependency arrow, optionally annotated with the stereotype name «derive». Since a constraint or note is involved, the arrow-tip may be omitted (the constraint or note is assumed to be the source). For example, Figure 3 includes area as a derived attribute.

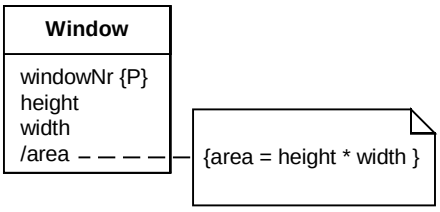


Figure 3: Area depicted as a derived attribute in UML, with derivation rule declared in a note

The dependency line may also be omitted entirely, with the constraint shown in braces beside the derived element (in this case, it is the modeling tool's responsibility to maintain the graphical linkage implicitly). A club-membership example of this was included in Part 8 of this series. As another example, Figure 4 expresses uncle information as a derived association. For illustration purposes, rolenames have been included for all association ends. Although precise rolenames are not always elegant, the use of rolenames in derivation rules corresponding to a path projection can facilitate concise expression of rules, as shown here. More complex derivation rules can be stated informally in English or formally in a language such as the Object Constraint language (OCL) [0].

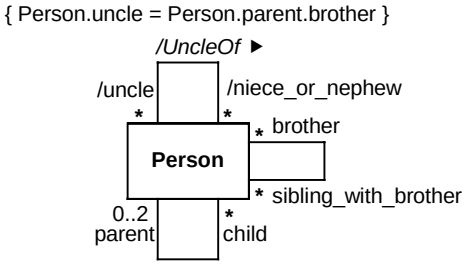


Figure 4: Derived uncle association (and roles) in UML, with derivation rule declared as a constraint

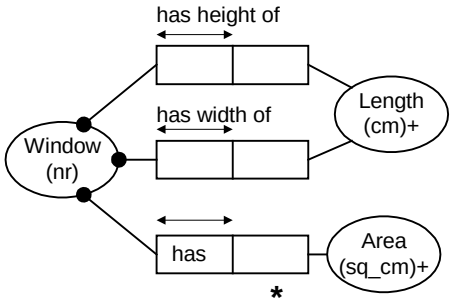
In ORM, a fact type that is primitive (i.e. not defined in terms of others) is said to be a *base* fact type. *Derived* fact types are defined in terms of other fact types (base or derived). If displayed on a diagram, derived fact types are marked with an asterisk. Constraints on derived fact types are typically implied. Whether or not a fact type is displayed on a diagram, a rule for deriving it should be declared. For example, Figure 5 includes a derivation rule to define the fact type Window has Area. The rule is specified here using ConQuer, an ORM conceptual query language supported in Visio's ActiveQuery tool. A

comment in braces has been prepended to the formal definition. Although automatic translation from ConQuer to SQL is provided in ActiveQuery, VisioModeler does not currently support this, so it is the developer's responsibility to implement any derivation rules entered in predicate property sheets.

An alternative ORM syntax for derivation rules uses "… **iff** …" (if and only if) instead of "**define** … **as** …". This syntax is useful if we wish to declare the underlying constraint before deciding which fact type is to be the *definiendum* (what is required to be defined). For example, the following logical constraint involves three fact types with one degree of freedom:

Window has Area **iff** Window has height of $Length_1$ **and**
Window has width of $Length_2$ **and**
Area = $Length_1$ * $Length_2$.

Any one of the fact types could be chosen to be derived from the other two. Given height and width, we can compute area; given area and height, we can compute width; and given area and width, we can compute height. Listing the area fact type before the "iff" doesn't conceptually require us to make that the derivable one. However, once the definiendum has been selected, it should be written as the head of the definition. In cases like this, where there really is a choice as to which is the definiendum, the decision is often based more on performance than on conceptual issues. In many cases however, there simply is no choice. For example, facts about sums and averages are derivable from facts about individual cases, but except for trivial cases we cannot derive the individual facts from such summaries.



* { area = height x width }
define Window has Area(sq_cm)
as
Window has height of $Length_1$ **and**
Window has width of $Length_2$ **and**
Area = $Length_1$ * $Length_2$

Figure 5 Window area depicted in ORM using a derived fact type with its derivation rule

It is an implementation issue whether a derived fact type is derived-on-query (lazy evaluation) or derived-on-update (eager evaluation). In the former case, the derived information is not stored, but computed only when the information is requested. For example, if our Window schema is mapped to a relational database, no column for area is included in the base table for Window (see Figure 6(a)). The rule for computing area may be included in a view definition or stored query, and is invoked only when the view is

queried or the stored query is executed. In most cases, lazy evaluation is preferred (e.g. computing a person's age from their birthdate and current date).

Sometimes eager evaluation is chosen because it offers significantly better performance (e.g. computing account balances). In this case, the information is stored as soon as the defining facts are entered, and updated whenever they are updated. In VisioModeler this option is chosen by selecting "Derived and Stored" from the Derived pane of the predicate properties sheet. As a sub-conceptual annotation, VisioModeler uses a double-asterisk "**" to indicate this choice. When the schema is mapped to a relational database, a column is created for the derived fact type (e.g. see Figure 6(b)), and the computation rule should be included in a trigger that is fired whenever the defining columns are updated (including inserts or deletes).

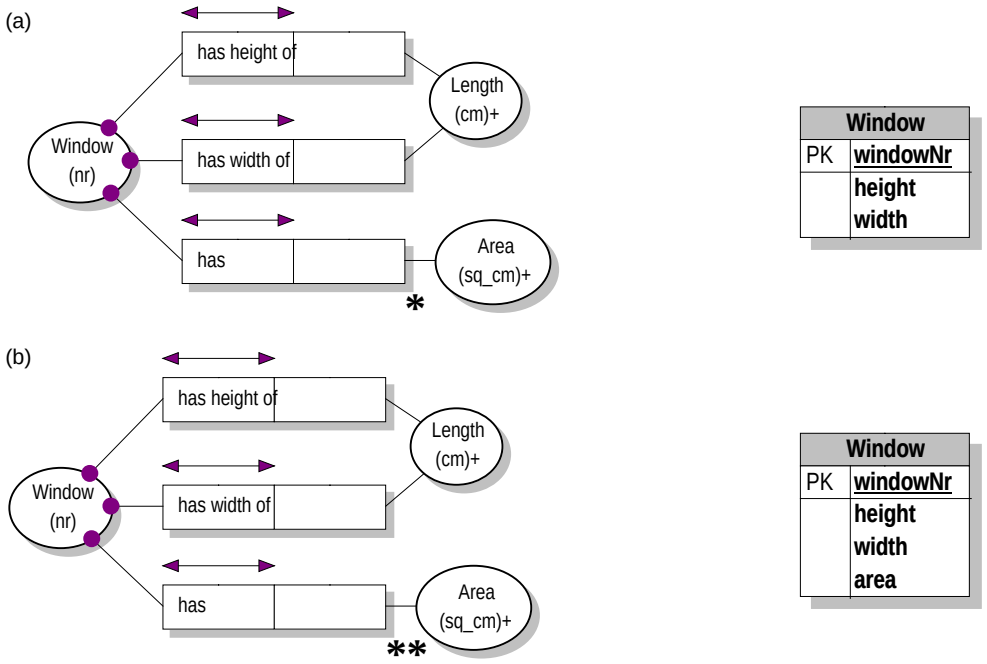


Figure 6 As an implementation issue, derived fact types may be evaluated lazily (a) or eagerly (b)

Some but not all derivations can be modeled graphically in ORM using equality constraints. In other cases, a fact type may be partly base and partly derived. These are sometimes called *hybrid* fact types. Although a notation has been suggested for them [0, p. 56], this is not yet included in UML. Some hybrid fact types may be handled in ORM using a subset constraint, e.g. see [0, p. 239]. As an example of a hybrid fact type, suppose that we know somebody's uncles but not his/her parents, and we wish to record this information about uncles. In this case, some uncle facts may be derived (as discussed earlier) while others must be entered directly. One way of dealing with this is to stored the entered facts in a base uncle fact type, separate from the derived fact type discussed earlier, which might be renamed, and specify the disjunction of these two fact types as another derived fact type.

We have seen that UML and ORM both provide support for derived information. As the examples illustrate, the use of attributes and association role names in UML often enables derivation rules to be expressed concisely using a functional notation. In contrast, the predicate-based derivation rules of ORM may appear somewhat verbose, especially for derivations of a mathematical rather than logical nature. While it is easy to come up with ORM derivation rules that are neater than the corresponding UML rules, the functional style of UML is definitely more convenient in many cases. To address this reality, ORM now allows rolenames as well as predicate names, and ConQuer has been enhanced to support this alternative notation. The main advantage of ORM's predicate-based notation is that it is more stable than an attribute-based notation, since it is not impacted by schema changes such as attributes being remodeled as associations. So the choice of a functional or relational style for derivation rules can involve a trade-off between convenience and stability.

Next issue

The next article in this series will discuss changeability and collection types in UML and ORM.

References

Booch, G., Rumbaugh, J. & Jacobson, I. 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading MA, USA.

Halpin, T. 1995, *Conceptual Schema and Relational Database Design, 2nd edn (revised 1999)*, WytLytPub, Bellevue WA, USA.

Halpin, T.A. & Vermeir, D. 1997, 'Default reasoning in information systems', *Database Applications Semantics*, Chapman & Hall, London, pp. 423-41.

Martin, J. & Odell, J. 1998, *Object-Oriented Methods: a Foundation, UML edn*, Prentice Hall, Upper Saddle River, New Jersey.

OMG-UML Specification v. 1.3 beta R6 draft, OMG UML Revision Task Force website, http://uml.systemhouse.mci.com/artifacts.htm.

Rumbaugh, J., Jacobson, I. & Booch, G. 1999, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading MA, USA.

Warmer, J. & Kleppe, A. 1999, *The Object Constraint Language: precise modeling with UML*, Addison-Wesley, Reading MA, USA.