Data modeling in UML and ORM: a comparison

Dr. Terry Halpin and Dr. Anthony Bloesch

Visio Corporation

This paper first appeared in the Journal of Database Management, vol. 10, no. 4 (Oct-Dec., 1999)*, Idea Group Publishing, Hershey PA, USA, pp. 4-13. (http://www.idea-group.com).*

The Unified Modeling Language (UML) is becoming widely used for software and database modeling, and has been accepted by the Object Management Group as a standard language for object-oriented analysis and design. For data modeling purposes, UML includes class diagrams, that may be annotated with expressions in a textual constraint language. Although facilitating the transition to object-oriented code, UML's implementation concerns render it less suitable for developing and validating a conceptual model with domain experts. This defect can be remedied by using a fact-oriented approach for the conceptual modeling, from which UML class diagrams may be derived. Object-Role Modeling (ORM) is currently the most popular fact-oriented approach to data modeling. This paper examines the relative strengths and weaknesses of ORM and UML for data modeling, and indicates how models in one notation can be translated into the other.

Introduction

The syntax The Unified Modeling Language (UML) is gaining popularity, and has been adopted by the Object Management Group as a standard for object-oriented (OO) modeling [24]. Much of UML has a programming flavor, with many constructs designed to assist developers of object-oriented code. The UML notation includes Use case diagrams, Static Structure diagrams (Class diagram, Object diagram), Behavior diagrams (Statechart diagram, Activity diagram), Interaction diagrams (Sequence diagram, Collaboration diagram), and Implementation diagrams (Component diagram, Deployment diagram). Since this paper focuses on conceptual data modeling, we restrict our discussion of UML to its class and object diagrams, as supplemented by textual annotations. Some empirical studies indicate that Entity Relationship (ER) schemas are often more correct and easier to develop than corresponding OO schemas [25]. There are many OO approaches however, and UML may be used for analysis by ignoring its implementation features. When used purely for analysis, UML class diagrams provide an extended ER notation.

UML's object-oriented approach facilitates the transition to object-oriented code. As shown later however, UML can make it awkward to capture and validate data concepts and business rules with domain experts, and to cater for structural changes in the application. These problems can be remedied by using a fact-oriented approach where communication takes place in simple sentences, each sentence type can easily be populated with multiple instances, and attributes are eschewed in the base model. Object Role Modeling (ORM) is a fact-oriented approach that harmonizes well with UML, since both approaches provide direct support for roles, n-ary associations and objectified associations. ORM pictures the world simply in terms of objects (entities or values) that play roles (parts in relationships). For example, you are now playing the role of reading, and this paper is playing the role of being read. Overviews of ORM may be found in [13, 14, 15] and a detailed treatment in [12].

The following section discusses criteria for evaluating the suitability of a conceptual modeling language. These design principles are then used to examine the relative strengths and weaknesses of UML and ORM for data modeling, focusing first on the data structures, and then moving on to constraints, outlining how models in one notation can be translated into the other. We then evaluate textual language support for constraints, derivation rules and queries. The conclusion summarizes the main points and identifies topics for future research.

Conceptual modeling language criteria

A modeling method comprises both a language and a procedure to guide modelers in using the language to construct models. A language has associated syntax (marks), semantics (meaning) and pragmatics (use). Written languages may be graphical (diagrams) and/or textual. The terms “abstract syntax” and “concrete syntax” are sometimes used to distinguish underlying concepts (e.g. class) from their representation (e.g. named rectangle). Conceptual modeling portrays the application domain at a high level, using terms and concepts familiar to the application users, ignoring logical and physical level aspects (e.g. the underlying database or programming structures used for implementation) and external level aspects (e.g. what screen forms will be used for data entry). The following criteria drawn from various sources [e.g. 4, 19, 20, 22] provide a basis for evaluating conceptual modeling languages.

- Expressibility
- Clarity
- Semantic stability
- Semantic relevance
- Validation mechanisms
- Abstraction mechanisms
- Formal foundation

The expressibility of a language is a measure of what it can be used to say. Ideally, a conceptual language should be able to completely model all details about the application domain that are conceptually relevant. This is called the 100% Principle [22]. ORM is a method for modeling and querying an information system at the conceptual level, and for mapping between conceptual and logical levels. Although various ORM extensions have been proposed for object-orientation and dynamic modeling [e.g. 1, 7, 21], the focus of ORM is on data modeling, since the data perspective is more stable and it provides a formal foundation on which operations can be defined. In this sense, UML is generally more expressive than standard ORM, since its use case, behavior and implementation diagrams model aspects beyond static structures. An evaluation of such additional modeling capabilities of UML and ORM extensions is beyond the scope of this paper. We show later that ORM diagrams are graphically more expressive than UML class diagrams.

The clarity of a language is a measure of how easy it is to understand and use. To begin with, the language should be unambiguous. Ideally, the meaning of diagrams or textual expressions in the language should be intuitively obvious. At a minimum, the language concepts and notations should be easily learnt and remembered. Semantic stability is a measure of how well models or queries expressed in the language retain their original intent in the face of changes to the application. The more changes one is forced to make to a model or query to cope with an application change, the less stable it is. Semantic relevance requires that only conceptually relevant details need be modeled. Any aspect irrelevant to the meaning (e.g. implementation choices, machine efficiency) should be avoided. This is called the conceptualization principle [22].

Validation mechanisms are ways in which domain experts can check whether the model matches the application. For example, static features of a model may be checked by verbalization and multiple instantiation, and dynamic features may be checked by simulation. Abstraction mechanisms allow unwanted details to be removed from immediate consideration. This is very important with large models (e.g. wall-size schema diagrams). ORM diagrams tend to be more detailed and larger than corresponding UML models, so abstraction mechanisms are often used. For example, a global schema may be modularized into various scopes or views based on span or perspective (e.g. a single page of a data model, or a single page of an activity model). Successive refinement may be used to decompose higher level views into more detailed views. Tools can provide additional support (e.g. feature toggles, layering, and object zoom). Such mechanisms can be used to hide and show just that part of the model relevant to a user's immediate needs [12, 6]. With minor variations, these techniques can be applied to both ORM and UML. ORM also includes an attribute abstraction procedure to generate an ER diagram as a view.

A formal foundation is needed to ensure unambiguity and executability (e.g. to automate the storage, verification, transformation and simulation of models), and allow formal proofs of equivalence and implication between alternative models [17]. Although ORM's richer, graphical constraint notation provides a more complete diagrammatic treatment of schema transformations, use of textual constraint languages can partly offset this advantage. For their data modeling constructs, both UML and ORM have an adequate formal foundation. Since ORM and UML are roughly comparable with regard to abstraction mechanisms and formal foundations, our following evaluation focuses on the criteria of expressibility, clarity, stability, relevance and validation.

Data structures

Table 1 summarizes the main correspondences between conceptual data modeling concepts in ORM and UML. In this section we consider the left half of the table. In UML and ORM, objects and data values are both instances. Each object is a member of at least one type, known as class in UML and an object type in ORM. ORM classifies objects into entities (UML objects) and values (UML data values—constants such as character strings or numbers).

In UML, entities are identified by oids, but in ORM they must have a reference scheme for human communication (e.g. employees might be referenced by social security numbers). UML classes must have a name, and may also have attributes, operations (implemented as methods) and play roles. ORM object types must have a name and play roles. Since our focus is on the data perspective, we avoid any detailed discussion of operations, except to note that some of these may be handled in ORM as derived relationship types. A relationship instance in ORM is called a link in UML (e.g. Employee 101 works for Company 'Visio'). A relationship type in ORM is called an association in UML (e.g. Employee works for Company). Object types in ORM are depicted as named ellipses, and simple reference schemes may be abbreviated in parentheses below the type name. Classes in UML are depicted as named rectangles to which attributes and operations may be added.

Table 1 Basic correspondence between ORM and UML conceptual concepts for data models

Data instances/structures		***Constraints***	
ORM	*UML*	*ORM*	*UML*
Entity	Object	Internal uniqueness	Multiplicity of ..1 §
Value	Data value	External uniqueness	— { use qualified assoc. § }
Object	Object or Data value	Simple mandatory role	Multiplicity of 1..
Entity type	Class	Disjunctive Mandatory role	—
Value type	Data type	Frequency: internal; external	Multiplicity §; —
Object type	Class or Data type	Value	Enumeration, and textual
— { use relationship type }	Attribute	Subset and Equality	Subset §
Unary relationship type	— { use Boolean attribute }	Exclusion	Or-constraint §
2+-ary relationship type	Association	Subtype link and definition	Subclass discriminator etc. §
2+-ary relationship instance	Link	Ring constraints	—
Nested object type	Association class	Join constraints	—
Co-reference	Qualified association §	Object cardinality	Class multiplicity
		— { use unique and mand. §}	Aggregation/composition
		—	Defaults/changeability
		Textual constraints	Textual constraints

§ = incomplete coverage of corresponding concept

Apart from object types, the only data structure in ORM is the relationship type. In particular, attributes are not used at all in base ORM. This is a fundamental difference between ORM and UML (and ER for that matter). Wherever an attribute is used in UML, ORM uses a relationship instead. The advantages of this are not fully recognized, despite debates in the past over the issue [e.g. 9]. Firstly, attribute-free models and queries are more stable, because they are free of changes caused by attributes evolving into other constructs (e.g. associations), or vice versa. For example, suppose we model car as an attribute of Employee. If we later decide that employees may drive many cars, we need to replace this attribute by an association (Employee drives Car) or a multi-valued attribute (cars). If we decide to record data about cars (e.g. carmodel) a Car class must be introduced. If attributes are replaced, queries and constraints based on them also need replacing. Since we can't be sure about our future modeling needs, use of attributes in the base model decreases semantic stability.

An ORM model is essentially a connected network of object types and relationship types. The object types are the semantic domains that glue things together, and are always visible. This connectedness reveals relevant detail and enables ORM models to be queried directly, traversing through object types to perform conceptual joins [5]. In addition, attribute-free models are easy to populate with multiple instances, facilitate verbalization (in sentences), are simpler and more uniform, avoid arbitrary modeling decisions, and facilitate constraint specification (see later).

Attributes however have two advantages: they often lead to a more compact diagram, and they can simplify arithmetic derivation rules (see later). For this reason, ORM includes algorithms for dynamically generating attribute-based diagrams as views [6, 12]. These algorithms assign different levels of importance to object types depending on their current roles and constraints, redisplaying minor fact types as attributes of the major object types. Elementary facts are the fundamental conceptual units of information, are uniformly represented as relationships, and how they are grouped into structures is not a conceptual issue. Apart from standard ORM, the OSM modeling method also rejects the use of attributes because of their inherent instability [8].

ORM allows relationships of any arity (number of roles), each with at least one reading or predicate name. An n-ary relationship may have up to n readings (one starting at each role), to more naturally verbalize constraints and navigation paths in any direction. ORM also allows role names to be added. A predicate is an elementary sentence with holes in it for object terms. These object holes may appear at any position in the predicate (mixfix notation), and are denoted by an ellipsis "…" if the predicate is not infix-binary. In support of our clarity criterion, mixfix notation enables natural verbalization of sentences in any language (e.g. in Japanese, verbs come at the end of sentences). ORM includes procedures to assist in the creation and transformation of models. A key step in its design procedure is the verbalization of relevant information examples, such as sample reports expected from the system. These "data use cases" are in the spirit of UML use cases, except the focus is on the underlying data.

ORM sentence types (and constraints) may be specified either textually or graphically. Both are formal, and can be automatically transformed into the other. In an ORM diagram, roles appear as boxes, connected by a line to their object type. A predicate appears as a named, contiguous sequence of role boxes. Since these boxes are set out in a line, fact types may be conveniently populated with tables holding multiple fact instances, one column for each role. This allows all fact types and constraints to be validated by verbalization as well as sample populations. Communication between modeler and domain expert takes place in a familiar language, backed up by population checks.

UML uses Boolean attributes instead of unary relationships, but allows relationships of all other arities. Each association may be given at most one name. Binary associations are depicted as lines between classes, with higher arity associations depicted as a diamond connected by lines to the classes. Roles are simply association ends, but may optionally be named. Verbalization into sentences is possible only for infix binaries, and then only by naming the association with a predicate name (e.g. "employs") and using an optional marker "▸" to denote the direction. Since roles for ternaries and higher arity associations are not on the same line, directional verbalization is ruled out. This non-linear layout also makes it impractical to conveniently populate associations with multiple instances. Add to this the impracticality of displaying multiple populations of attributes, and it is clear that class diagrams do not facilitate population checks (e.g. see later discussion of Figure 3 and Figure 4). UML does provide object diagrams for instantiation, but these are convenient only for populating with a single instance. Hence, "the use of object diagrams is fairly limited" ([24]). We conclude that ORM surpasses UML on the validation mechanism criterion.

Both UML and ORM allow associations to be objectified as first class object types, called association classes in UML and nested object types in ORM. UML requires the same name to be used for the association and the association class, impeding natural verbalization, in contrast to ORM nesting based on linguistic nominalization (a verb phrase is objectified by a noun phrase). UML allows objectification of n:1 associations. Currently ORM forbids this except for 1:1 cases, since attached roles are typically best viewed as played by the object type on the "many" end of the association [11]. However, ORM could be relaxed to allow this, with its mapping algorithms adding a pre-processing step to re-attach roles and adjust constraints internally. In spite of identifying association classes with their underlying association, UML displays them separately, making the connection by a dashed line. In contrast, ORM intuitively envelops the association with an object type frame (see Figure 1).

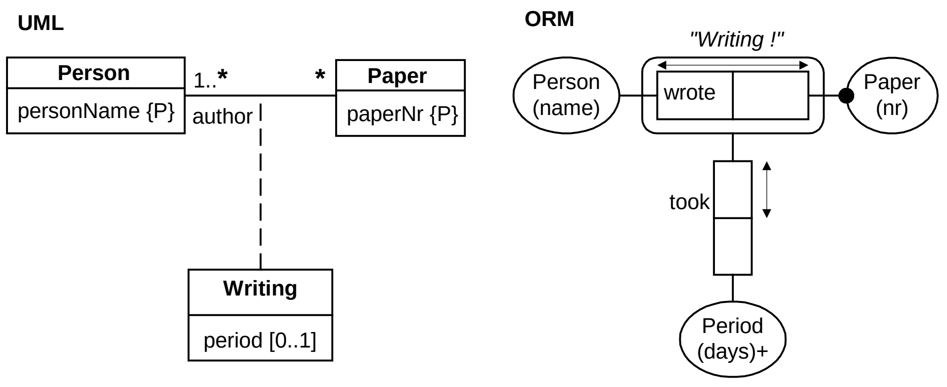

Figure 1: Writing is depicted as an objectified association in UML and ORM

Constraints

In Figure 1, the UML diagram includes multiplicity constraints on the association roles. The "1..*" indicates that each paper is written by one or more persons. In ORM this is captured as a mandatory role constraint, represented graphically by a black dot. VisioModeler, a popular ORM tool, allows this constraint to be entered graphically, or by answering a multiplicity question, or by induction from a sample population, and can automatically verbalize the constraint. If the inverse predicate "is written by" has been entered (its display may be suppressed for tidiness, as in Figure 1), VisioModeler verbalizes the constraint as "each Paper is written by at least one Person".

In UML the "*" on the right hand role indicates that each person wrote zero or more papers. In ORM the lack of a mandatory role constraint on the left role indicates it is optional (a person might write no papers), and the arrow-tipped line spanning the predicate is a uniqueness constraint indicating the association is many:many (when the fact table is populated, each whole row is unique). A uniqueness constraint on a single role means that entries in that column of the associated fact table must be unique. Figure 2 summarizes the equivalent constraint notations for binary associations, read from left to right. The third case (m:n optional) is the weakest constraint pattern. Though not shown here, 1:n cases are the reverse of the n:1 cases, and 1:1 cases combine the n:1 and 1:n cases.

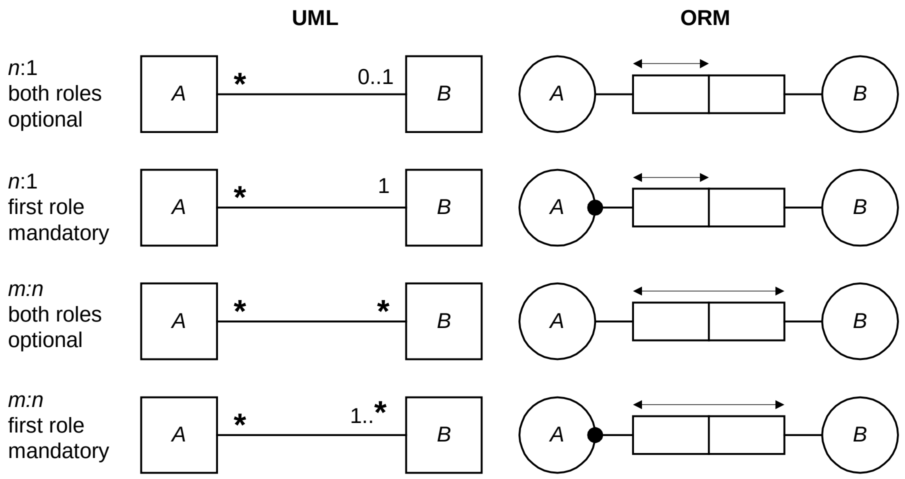

Figure 2: Some equivalent representations in UML and ORM

An internal constraint applies to roles in a single association. For an n-ary association, each internal uniqueness constraint must span at least n-1 roles. Unlike many ER notations, UML and ORM can express all possible internal uniqueness constraints. For example, Figure 3 is a UML diagram that includes a ternary association (Usage) in which both Room-Time and Time-Activity pairs are unique. This schema also includes a textual constraint written informally as note attached by dashed lines to the three associations involved in the constraint. A textual constraint is needed here since UML provides no graphical way of capturing the constraint. UML does allow the constraint to be captured formally in OCL, but the syntax of OCL is too mathematical for it to be used to validate rules with subject matter experts who typically have little technical background.

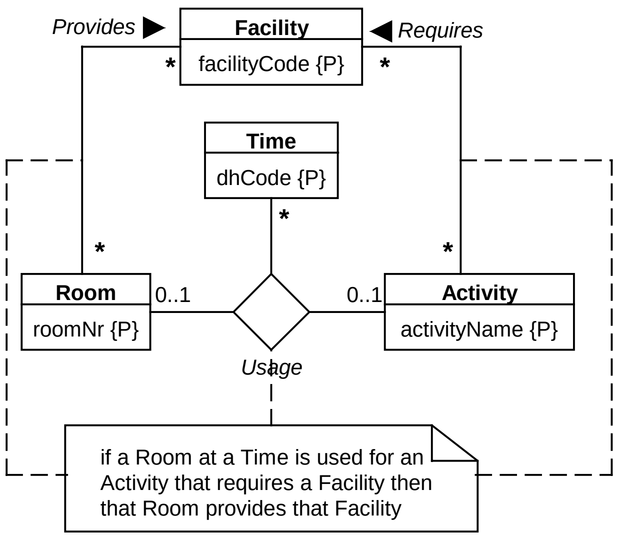

Figure 3: Multiplicity constraints on a ternary in UML, and a textual constraint declared in a note

An ORM depiction of the same situation is shown in Figure 4, along with sample populations. The dotted arrow is a join-subset constraint (see later) that formally captures the textual constraint shown in the UML version. Note how useful populations are for checking the constraints. For example, the data clearly illustrate the uniqueness constraints. If Time-Activity is not really unique, this can be shown by adding a counterexample. If UML object diagrams were used to show these populations, the constraint patterns would be far harder to see, since each instance of each class and association would be shown separately (checking even the simple 1:1 constraint between facility codes and names requires inspecting three facility objects). Although in principle, ORM-like fact tables could be used with UML associations, relating association-roles to the relevant fact columns would often be awkward.

Multiplicity constraints in UML may specify any range of occurrence frequencies (e.g. 1, 3..7) but each is applied to a single role (for n-aries, the range indicates what is possible when the other n-1 classes have a fixed value). ORM allows the same ranges, but partitions the multiplicity concept into the orthogonal notions of mandatory role and frequency constraints. This useful separation localizes global impact to just the mandatory role constraint (e.g. every population instance of an object type A must play every mandatory role of A). Because of its non-local impact, modelers should omit this constraint unless it is really needed (as in Figure 1). ORM frequency constraints apply only to populations of the constrained roles (e.g. if an instance plays that role, it does so the specified number of times) and hence have only local impact. Frequency constraints in ORM are depicted as number ranges next to the relevant roles. For example, to add the constraint that papers must be reviewed by at least 2 people, we add the mark “≥2” beside the first role of Paper is reviewed by Person. Uniqueness constraints are just frequency constraints with a frequency of 1, but have a special notation because of their importance.

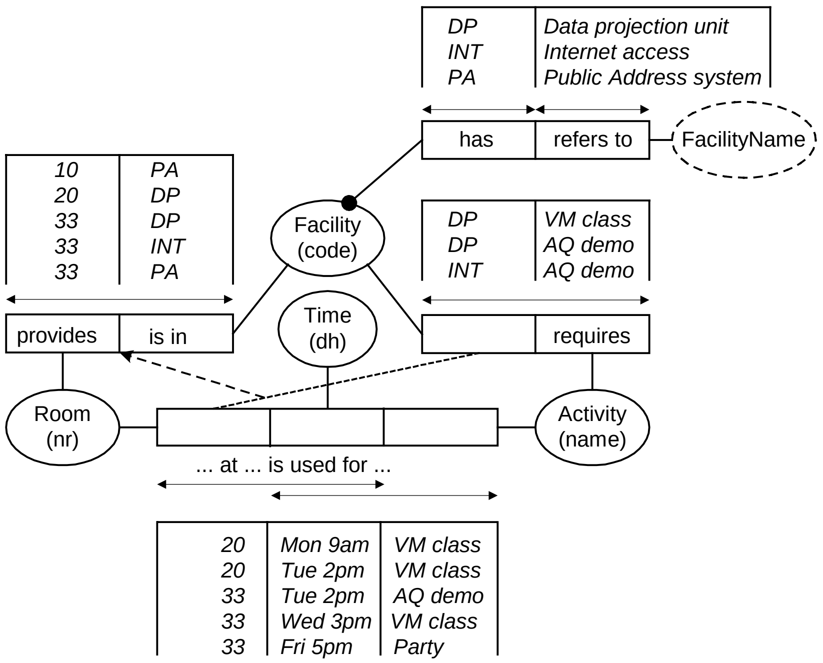

Figure 4: An ORM diagram with sample populations

Attribute multiplicity constraints in UML are placed in square brackets after the attribute name (e.g. Figure 1). If no such constraint is specified, the attribute is assumed to be single-valued and mandatory. Multi-valued attributes are arguably an implementation concern. Mandatory role constraints in ORM may apply to a disjunction of roles, e.g. each academic is either tenured or contracted till some date. UML cannot express disjunctive mandatory role constraints graphically. Perhaps influenced by oids, UML omits a standard notation for attribute uniqueness constraints (candidate keys). It suggests that boldface might be used for this (or other purposes) as a tool extension. Another alternative is to annotate unique attributes with comments (e.g. {P} for primary reference, {U1} etc.).

Frequency and uniqueness constraints in ORM may apply to a sequence of any number of roles from any number of predicates. This goes far beyond the graphical expressibility of UML. For example, consider the m:n fact type Account(nr) is used by Client(nr) and the n:1 fact type Account(nr) has AccountType(code), and add the uniqueness constraint that for any given account type, each client has at most one account [12, p. 407]. ORM constraints that span different predicates are called external constraints. Only a few of these can be graphically expressed in UML. For example, subset and equality constraints in ORM may be expressed between two compatible role-sequences, where each sequence is formed by projection from possibly many connected predicates. Figure 5 includes two simple examples: students have second names only if they have first names, and may pass tests in a course only if they enrolled in that course. ORM visually distinguishes value types from entity types by using dashed-ellipses (e.g. Surname, FirstName and SecondName).

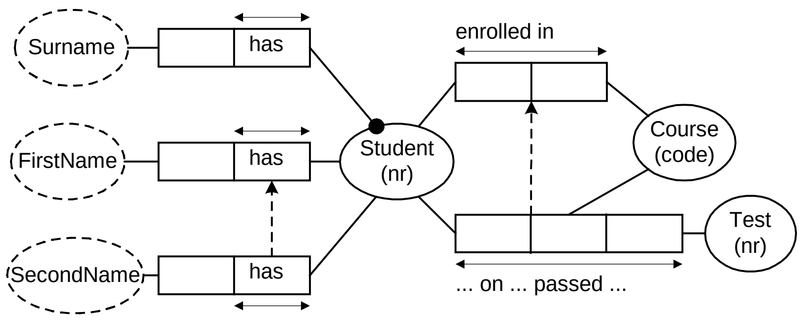

Figure 5 Subset constraints in ORM

UML is capable of diagramming only basic subset constraints between binary associations, e.g. in Figure 6, a person who chairs a committee must be a member of it. UML omits a notation for diagramming subset constraints between parts of associations (as in Figure 5), and this inability to project on the relevant roles invites modeling errors (e.g. [0], p. 68). However, as the right half of Figure 6 illustrates, it is possible to capture ORM subset constraints in UML by adding a textual constraint or sometimes by applying a model transformation (e.g. remodel a ternary using an association class)

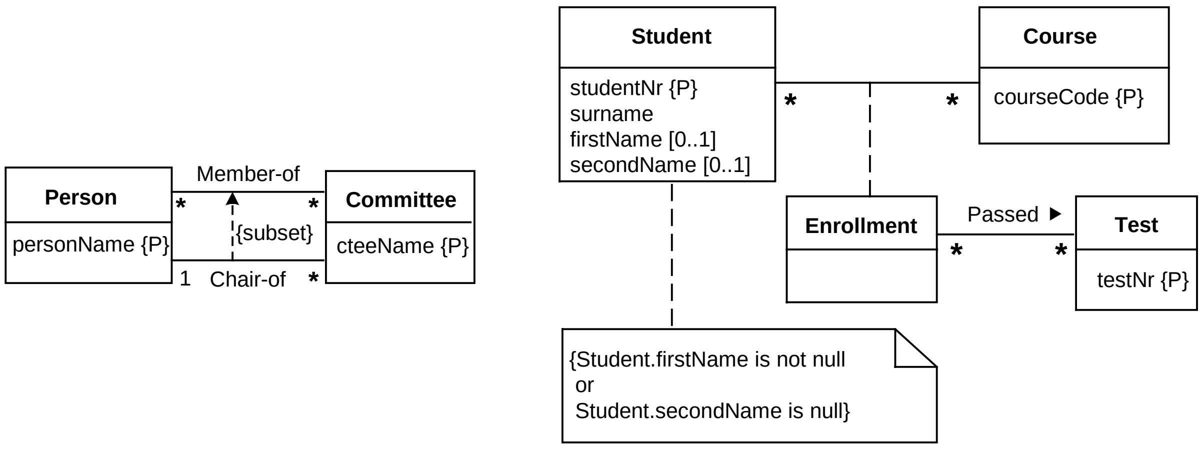

Figure 6 Specifying subset constraints in UML directly or by other means

The dotted arrow in Figure 4 expressed the following join-subset constraint: if a Room at a Time is used for an Activity that requires a Facility then that Room provides that Facility. If we need to record the title and sex of each employee, we should also include a populated relationship type indicating which titles determine which sex (e.g. “Mrs”, “Miss”, “Ms” and “Lady” apply only to the female sex). In ORM this is easily visualized as a join-subset constraint (see Figure 7) and verbalized (if Person1 has a Title that determines Sex1 then Person1 is of Sex1). If we instead model title and sex as attributes, this rule cannot be diagrammed. In ORM a value constraint restricts the population of a value type, and is indicated in braces. In UML, such constraints may be declared as enumerations or as textual constraints (e.g. see the Sexcode constraint in Figure 7).

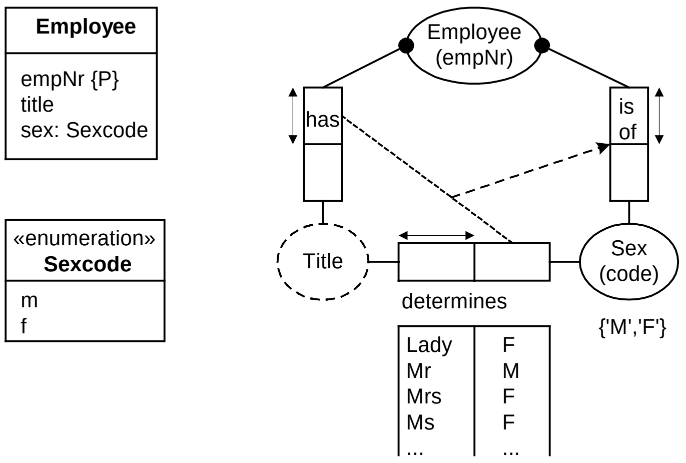

Figure 7 ORM makes it easy to capture the constraint between title and sex

ORM allows exclusion constraints over a set of compatible role-sequences, by connecting "⊗" by dotted lines to the relevant role-sequences. For example, given the associations Person wrote Paper and Person reviewed Paper, consider the two exclusion constraints: no person wrote and reviewed; no person wrote and reviewed the same paper. ORM distinguishes these cases by noting the precise arguments of the constraint. If a set of roles is both exclusive and disjunctively mandatory, ORM specifies this by combining an exclusion and disjunctive-mandatory constraint. This "exclusive or" case is captured in UML by connecting "{xor}" to the relevant associations by dashed lines to indicate exactly one is played. UML has no graphic notation for simple exclusion between roles, role-sequences, attributes, or between attributes and associations. As a convenience, ORM also includes *equality constraints* as an abbreviation for subset constraints in both directions.

UML uses qualified associations in many cases where ORM uses an external uniqueness constraint for co-referencing. Figure 8 is based on an example from the UML standard [24], along with the ORM counterpart. Qualified associations are shown as named, smaller rectangles attached to a class. ORM uses a circled "u" to denote an external uniqueness constraint (the bank name and account number uniquely define the account). The UML notation is less clear, and less adaptable. For example, if we now want to record something about the account (e.g. its balance) we need to introduce an Account class, and the connection to accountNr is unclear. The problem can be solved in UML by using an association class instead, though this is not always natural.

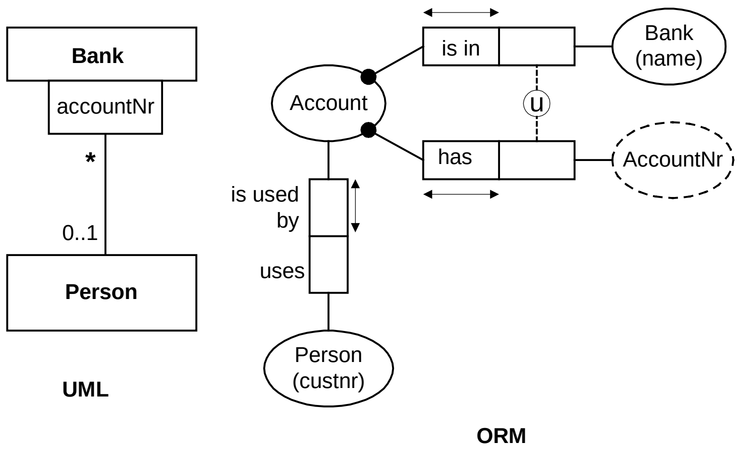

Figure 8 Qualified association in UML, and co-referenced object type in ORM

Both UML and ORM provide support for subtyping, including multiple inheritance. Both show subtypes outside, connected by arrows to their supertype(s), and both allow declaration of constraints between subtypes such as exclusion and totality. UML provides only weak support for defining subtypes: a discriminator label may be attached to subtype arrows to indicate the basis for the classification (e.g. a "sex" discriminator might specialize Man and Woman from Person). This does not guarantee that instances populating these subtypes have the correct values for a sex attribute that might apply to Person. Moreover, more complicated subtype definitions are sometimes required. Finally, subtype constraints such as exclusion and totality are typically implied by subtype definitions in conjunction with existing constraints on the supertypes; these implications are captured in ORM but are ignored in UML, leading to the possibility of inconsistent UML models. For further discussion on these issues see [12, 16].

ORM includes a number of other graphic constraints with no counterpart in UML. For example, ring constraints such as irreflexivity, asymmetry, intransitivity and acyclicity, may be specified over a pair of roles played by the same object type (e.g. Person is parent of Person is acyclic and deontically intransitive). Such constraints can be specified as comments in UML. UML treats aggregation as a special kind of whole/part association, attaching a small diamond to the role at the "whole" end of the association. In ORM this is shown as an m:n association Whole contains Part. UML treats composition as a stronger form of aggregation in which each part belongs to at most one whole (in ORM the "contains" predicate becomes 1:n). Whole and Part are not necessarily disjoint types, so ring constraints may apply (e.g. Part contains Part). UML aggregation also has behavioral semantics concerned with implementation at the code level (e.g. copy and access semantics), but these are not conceptual issues and have no counterpart in ORM.

UML allows collection types to be specified as annotations. For example, if we wish to record the order in which authors are listed for any given paper, the UML diagram in Figure 1 can have its author role annotated by "{ordered}". This denotes an ordered set (sequence with unique members). ORM has two approaches to handle this. One way keeps base predicates elementary, annotating them with the appropriate constructor as an implementation instruction. In this example, we use the ternary fact type Person wrote Paper in Position, place uniqueness constraints over Person-Paper and Paper-Position, and annotate the predicate with "{seq}" to indicate mapping the positional information as a unique sequence. Sets, sequences and bags may be treated similarly. This method is recommended, partly because elementarity allows individual instantiation and simplifies the semantics. The other way allows complex object types in the base model by applying constructors directly to them (e.g. [21]).

Both ORM and UML include object cardinality constraints for limiting the cardinality of a type's population. For example, the number of senators may be capped at 50 in ORM by writing "#≤50" beside the object type Senator, and in UML by writing "50" at the top right hand corner of the Senator class. In UML, attributes may be assigned default values, and restrictions placed on their changeability (e.g. "frozen"). Although not supported in ORM, such features could easily be added to ORM as role properties. More sophisticated proposals to extend ORM to handle default information have also been made [18].

ORM includes various sub-conceptual notations that allow a pure conceptual model to be annotated with implementation detail (e.g. indexes, subtype mapping choices, constructors). UML includes a vast set of such annotations for class diagrams, providing intricate detail for implementation in object-oriented code (e.g. navigation directions across associations, attribute visibility (public, protected, private), etc.). These are irrelevant to conceptual modeling and are hence ignored in this paper.

Textual languages for constraints, derivation and queries

Graphical languages are convenient for expressing common constraints. However, their simplicity comes at the cost of expressive power. The common solution is to add textual constraint annotations to the notation. UML allows informal, semi-formal, and formal constraints. As an extension, UML includes OCL (Object Constraint Language) as a formal textual constraint language. OCL was part of a submission by IBM and ObjecTime Limited to the OMG. OCL was developed by Jos Warmer and is based on the Syntropy method of Steve Cook and John Daniels. Constraint languages tend to be either algebraic (e.g. OBJ) or model based (e.g. Z and VDM). OCL is a model based constraint language. Constraints define the set of legal models. For example, a stack pop operation could be specified as:

```
Stack::pop() : Element
            pre: elements->notEmpty
            post: elements@pre = elements->append(result)
```

This means that before a pop operation, the list of elements must be nonempty, and after a pop operation the list of elements before the pop equals the list after the pop with the result appended. By contrast, in an algebraic language we would use axioms like the following (meaning that if we push an element e onto a stack, then a pop operation will return e):

```
∀ s:Stack; e:Element •
        s.push(e).pop() = e
```

Any practical constraint language must deal with undefined values. For example, the following specification declares an operation to calculate the result of dividing a real number by denom. Note: if denom is zero (and hence the result undefined) then the result is zero.

```
Real::safeDiv(denom:Real) : Real
post:  self@pre = self and
       denom = 0.0 implies result = 0.0 and
       denom <> 0.0 implies result = value@pre / denom
```

In OCL, expressions containing undefined expressions are themselves undefined. To stop the entire expression above becoming undefined, logical operators follow Kripke's strong three valued logic (K3). In K3, a implies b is true exactly when a is false or a and b are true; thus the above specification is defined for denom = 0.0. In practice, much more careful handling of undefined expressions is required [3]. For example, using K3 instead of classical logic means that theorems from standard mathematics cannot be used for proofs. Having to redevelop standard results is error prone and greatly increases the effort required to prove results.

UML attributes and associations may be derived (e.g. /count). OCL can be used to express the derivation rules through constraints. For example, the following invariant expresses a derivation rule for /count.

```
Stack
    count = elements->size
```

Various textual languages have been defined to express constraints, derivation rules and queries in ORM (e.g. RIDL [23], PSM [20] and ConQuer [4, 5]). Of these, only ConQuer has been implemented in a conceptual query tool. ConQuer is essentially classical logic with set theory. Unlike OCL, ConQuer is based on standard mathematics and thus can use all the theorems of standard mathematics. Also unlike OCL, ConQuer is designed to take advantage of modern user interfaces. Derivation rules are expressible in ConQuer using set comprehension, since an ORM fact table is essentially a set of tuples. In ConQuer, the derived fact: 'Product has gross margin of MoneyAmount.' is expressible as:

```
Product has cost of MoneyAmount as Cost
  └─ has wholesale price of MoneyAmount as Price
  └─ ✓ Price - Cost
```

Or mathematically, as: { p:Product; m:MoneyAmount | ∃ c: MoneyAmount; w: MoneyAmount • p has cost of c ∧ p has wholesale price of w ∧ m = w – c }

Similarly, the constraint: ‘No product may have a gross margin under 30%.’ is expressible as:

```
for no Product
Product has cost of MoneyAmount as Cost
   └─ has wholesale price of MoneyAmount as Price
   └─ Price / Cost < 1.3
```

Or mathematically, as: $\neg \exists$ p:Product $\bullet$ $\exists$ c: MoneyAmount; w: MoneyAmount $\bullet$ p has cost of c $\wedge$ p has wholesale price of w $\wedge$ w / c < 1.3

By using a ‘.’ notation, OCL is able to express mathematical expressions more succinctly than ConQuer. However, since ORM already supports named roles, ConQuer could be extended to support expressions like:

```
✓ Product
    └─ ✓ Product.Price - Product.Cost
```

A disadvantage of the dot notation is its reliance on functional attributes. Constraint changes and schema additions might require attributes to be remodeled, making the expression obsolete. ConQuer’s predicate-based notation is immune to such changes. Nevertheless, some features may reliably remain functional (e.g. birthdate), and for mathematical operations functional notation is certainly convenient.

For common data modeling constraints, such as subset constraints, ORM’s graphical notation is more clear and less error prone than UML’s textual notation. As well, ORM’s graphical constraints can be automatically mapped to efficient SQL. By contrast, code generated from UML’s OCL constraints is unlikely to be as efficient as hand crafted SQL. UML constraints could be expressed in SQL but at the cost of both being error prone and less clear. However, OCL’s rich constraint language does allow a wide variety of constraints to be expressed succinctly. For example, OCL has a rich language for dealing with collections and associations. The constraint “Every student must take at least two foreign language courses.”, applied to Figure 6, can be expressed in OCL as:

```
Student
    courses->select(c | courseCode.substring(1, 2) = ‘FL’)->size >= 2
```

Debate rages over how desirable it is for textual constraint languages to be efficiently translatable into code. The prospect of automatically generating code from constraints is attractive, but this must be weighed against reduced expressive power. OCL strikes a good balance since OCL constraints are easily mapped into procedural code yet the language has many powerful constructs.

Conclusion

This paper identified the following principles for evaluating modeling languages and applied them in evaluating UML and ORM for conceptual data modeling: expressibility; clarity; semantic stability; semantic relevance; validation mechanisms; abstraction mechanisms; formal foundations. Although ORM's richer constraint notation makes it more expressive graphically, both methods extend expressibility through the use of textual languages. ORM scores higher on clarity, because its structures may be directly verbalized as sentences, it is based on fewer, more orthogonal constructs, and it reveals semantic connections across domains. Being attribute-free, ORM is more stable for both modeling and queries. ORM is easier to validate (through verbalization and multiple instantiation). Both methods are amenable to similar abstraction mechanisms, and have adequate formal foundations. UML class diagrams are often more compact, and can be adorned with a vast array of implementation detail for engineering to and from object-oriented programming code. Moreover, UML includes mechanisms for modeling behavior, and its acceptance as an OMG standard is helping it gain wide support in industry, especially for the design of object-oriented software.

Thus both methods have their own advantages. For data modeling purposes, it seems worthwhile to provide tool support that would allow users to gain the advantages of performing conceptual modeling in ORM, while still allowing them to work with UML. Visio Enterprise already supports transformations between ORM, ER, Relational and Object-Relational models, as well as fully supporting UML. Research is currently under way to add transformations between ORM and UML. Once this support is widely available, empirical studies are planned to study why and how practitioners choose and/or integrate modeling methods in practice.

References

1. Barros, A., ter Hofstede, A. & Proper, H. 1997. 'Towards real-scale business transaction workflow modelling', *Proc. CAiSE'97* (Barcelona, Spain, June), A. Olive, J. Pastor eds, Springer Verlag, Berlin, 437-450.
2. Blaha, M. & Premerlani, W. 1998, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, New Jersey.
3. Bloesch, A. 1995, 'The Standard Ergo Theories', *Technical Report 95-43*, Software Verification Research Centre, The University of Queensland, Brisbane, Australia (Oct.).
4. Bloesch, A. & Halpin, T. 1996, 'ConQuer: a conceptual query language', *Proc. 15th International Conference on Conceptual Modeling ER'96* (Cottbus, Germany), B. Thalheim ed., Springer LNCS 1157 (Oct.) 121-133.
5. Bloesch, A. & Halpin, T. 1997, 'Conceptual queries using ConQuer-II', *Proc. 16th Int. Conf. on Conceptual Modeling ER'97* (Los Angeles), D. Embley, R. Goldstein eds, Springer LNCS 1331 (Nov.) 113-126.
6. Campbell, L., Halpin, T. & Proper, H. 1996, 'Conceptual schemas with abstractions: making flat conceptual schemas more comprehensible', *Data & Knowledge Engineering*, 20, 1, 39-85.
7. De Troyer, O. & Meersman, R. 1995, 'A logic framework for a semantics of object oriented data modeling', *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS. 1021, (Dec.) 238-49.
8. Embley, D. 1998, *Object Database Management*, Addison-Wesley.

9. Falkenberg, E. 1976, ‘Concepts for modelling information’, *Modelling in Data Base Management Systems*, G. Nijssen ed., North-Holland, Amsterdam, pp. 95-109 (see esp. p. 104, where “properties” means “attributes”).
10. Fowler, M. with Scott, K. 1997, *UML Distilled*, Addison-Wesley.
11. Halpin, T. 1993, ‘What is an elementary fact?’, *Proc. First NIAM-ISDM Conf.*, G.Nijssen, J. Sharp eds, Utrecht.
12. Halpin, T. 1995, *Conceptual Schema and Relational Database Design, 2nd edn* (revised 1999), WytLytPub, Bellevue WA, USA.
13. Halpin, T. 1996, ‘Business rules and object-role modeling’, *Database Prog. & Design*, 9, 10, Miller Freeman, 66-72.
14. Halpin, T. 1998, ‘Object Role Modeling: an overview’, white paper, www.orm.net.
15. Halpin, T. 1998, ‘Object Role Modeling (ORM / NIAM)’, *Handbook on Architectures of Information Systems*, P. Bernus, K. Mertins & G. Schmidt eds, Springer-Verlag, Berlin, pp. 81-101.
16. Halpin, T. & Proper, H. 1995, ‘Subtyping and polymorphism in object-role modelling’, *Data & Knowledge Engineering* 15, 3 (June), 251-281.
17. Halpin, T. & Proper, H. 1995, ‘Database schema transformation and optimization’, *OOER’95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, 1021 (Dec.) 191-203.
18. Halpin, T. & Vermeir, D. 1997, ‘Default reasoning in information systems’, *Database Application Semantics*, R. Meersman & L. Mark eds, Chapman & Hall, London, 423-442.
19. ter Hofstede, A.1993, *Information Modelling in Data Intensive Domains*, PhD thesis, University of Nijmegen.
20. ter Hofstede, A., Proper, H. & van der Weide, T. 1993, ‘Formal definition of a conceptual language for the description and manipulation of information models’, *Information Systems* 18, 7 (Oct.), 489-523.
21. ter Hofstede, A. & van der Weide, T. 1994, ‘Fact orientation in complex object role modelling techniques’, *Proc. First Int. Conf. on Object-Role Modelling* (Magnetic Island, Australia, July), T. Halpin, R. Meersman eds, 45-59.
22. ISO 1982, *Concepts and Terminology for the Conceptual Schema and the Information Base*, J. van Griethuysen ed., ISO / TC97 / SC5 / WG3-N695 Report, ANSI, New York.
23. Meersman, R. 1982, ‘The RIDL conceptual language’, Research report, Int. Centre for Information Analysis Services, Control Data Belgium Inc., Brussels, Belgium.
24. OMG UML Revision Task Force, *OMG Unified Modeling Language Specification*, http: / / uml.systemhouse.mci.com / .
25. Shoval, P. & Shiran, S. 1997, ‘Entity-relationship and object-oriented data modeling—an experimental comparison of design quality’, *Data & Knowledge Engineering*, 21, 3 (Feb.) 297-315.