A comparison of UML and ORM for data modeling

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

and Dr. Anthony Bloesch

Director of Database Software Modeling, Visio Corporation

This paper first appeared in Proc. EMMSAD'98 3rd IFIP WG8.I International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, *Pisa, Italy, in June, 1998.*

The Unified Modeling Language (UML) is becoming widely used for both database and software modeling, and is being evaluated 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 *Unified Modeling Language* (*UML*) is gaining wide popularity, and has been adopted by the Object Management Group as a standard for object-oriented (OO) modeling. Much of UML has a programming flavor, with many constructs designed to assist developers of object-oriented code. However, this paper focuses on the suitability of methods for developing a conceptual model of the data perspective. Hence 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, understandable and easy to develop than a corresponding OO schema [24]. 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, but 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.

The following section discusses some basic criteria for evaluating the suitability of a conceptual modeling language. These design principles are used in later sections 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. Along the way, we outline how models in one notation can be translated into the other. The conclusion summarizes the main points and identifies topics for future research. Appendix 1 provides further background on UML and ORM, and Appendix 2 evaluates textual language support in UML and ORM for constraints, derivation rules and queries.

Conceptual modeling language criteria

A modeling method comprises both a language and a procedure to guide the modeler 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 at a fundamental but high level, using terms and concepts familiar to the application users. A conceptual model ignores logical and physical level aspects such as the underlying database structures to be used for implementation, and also ignores external level aspects such as what screen forms will be used for data entry. The following criteria provide a useful basis for evaluating conceptual modeling methods:

- 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 [20]. 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, 19], 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. In addition, ORM can be used in conjunction with the other UML diagrams, since ORM diagrams may be abstracted to attribute views or transformed into 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 [20].

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. ORM diagrams tend to be more detailed and take up more space than corresponding UML models, so abstraction mechanisms are often used. Various mechanisms such as modularization, refinement levels, feature toggles, layering, and object zoom can be used to hide and show just that part of the model relevant to a user's immediate needs [11, 6]. With minor variations, these techniques can be applied to both ORM and UML. ORM also includes an attribute abstraction procedure to automatically generate a UML or 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). One particular benefit is to allow formal proofs of equivalence and implication between alternative models for the same application [16]. Although ORM's richer graphic constraint notation provides a more complete diagrammatic treatment of schema transformations, use of textual constraint languages can partly offset this advantage. With respect to their data modeling constructs, both UML and ORM have an adequate formal foundation. Since the ORM and UML languages are roughly comparable with regard to abstraction mechanisms and formal foundations, our evaluation in the following sections will focus 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).

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	—
Value type	Data type	Internal frequency	Multiplicity §
Object type	Class or Data type	External frequency	—
— { use relationship type }	Attribute	Subset	Subset §
Unary relationship type	— {use Boolean attribute}	Exclusion	xor-constraint §
2⁺-ary relationship type	Association	Subtype link & definition	Subclass discriminator etc. §
2⁺-ary relationship instance	Link	Ring constraints	—
Nested object type	Association class	Join constraints	—
Co-reference	Qualified association §	— {use uniq. and mand.}	Aggregation / composition
		—	Initial / changeability
		Textual constraints	Textual constraints

§ = incomplete coverage of corresponding concept

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.

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 one of the fundamental differences 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. Firstly, attribute-free models and queries are *more stable*, because they are free of changes caused by attributes evolving into entities or relationships, or vice versa. 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, using traversal through object types to perform conceptual joins [5]. In addition, attribute-free models are easy to populate with multiple instances, facilitate verbalization, are simpler and more uniform, facilitate constraint specification and avoid arbitrary modeling decisions. Suppose we need to record the title and sex of each employee. A complete model should include a relationship type to indicate which titles are restricted to which sex (e.g. “Mrs”, “Miss”, “Ms” and “Lady” apply only to the female sex). In ORM this constraint can be captured graphically as a join-subset constraint between the relevant fact types, or textually as a constraint in a formal ORM language (e.g. **if** Person_1 has **a** Title **that** is restricted to Sex_1 **then** Person_1 is of Sex_1). If we instead model title and sex as attributes, it is unclear how to express relevant restriction association.

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, 11]. 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 relationship type has at least one reading or predicate name. An *n*-ary relationship may have up to *n* readings (one starting at each role), to provide more natural verbalization of 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. Mixfix notation enables natural verbalization of sentences in any language (e.g. in Japanese, verbs come at the end of sentences). ORM includes various procedures to assist in the creation and transformation of models. A key step in its design procedure is the verbalization of information examples relevant to the application, such as sample reports expected from the system. This is in the spirit of UML’s 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, and this is optional. 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 the line ends, but may optionally be given names. 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 are almost useless for population checks (e.g. [28], p. 62). UML does provide *object diagrams* for instantiation, but these are convenient only for populating associations with a *single* instance. Adding multiple instances leads to a mess (e.g. [2], p. 31). Hence, "the use of object diagrams is fairly limited" ([28], p. 23).

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 original 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 [10]. However, ORM can be relaxed to downgrade this error to a warning, and mapping algorithms can add 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).

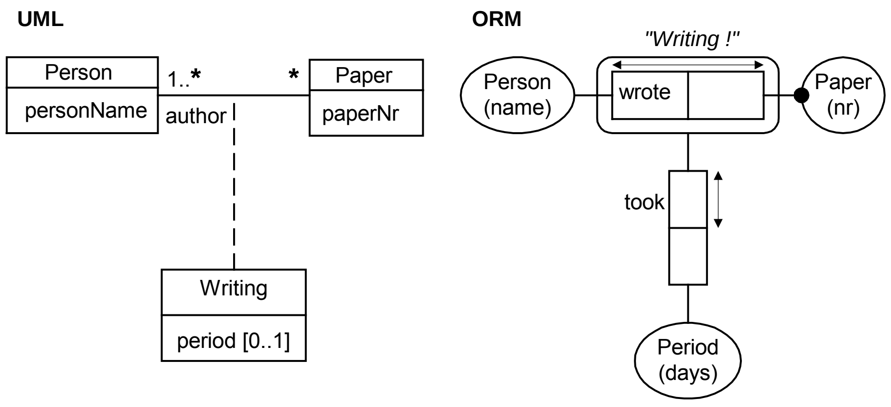

Figure 1: Writing is depicted as an objectified association in UML and ORM

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. InfoModeler 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), InfoModeler 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.

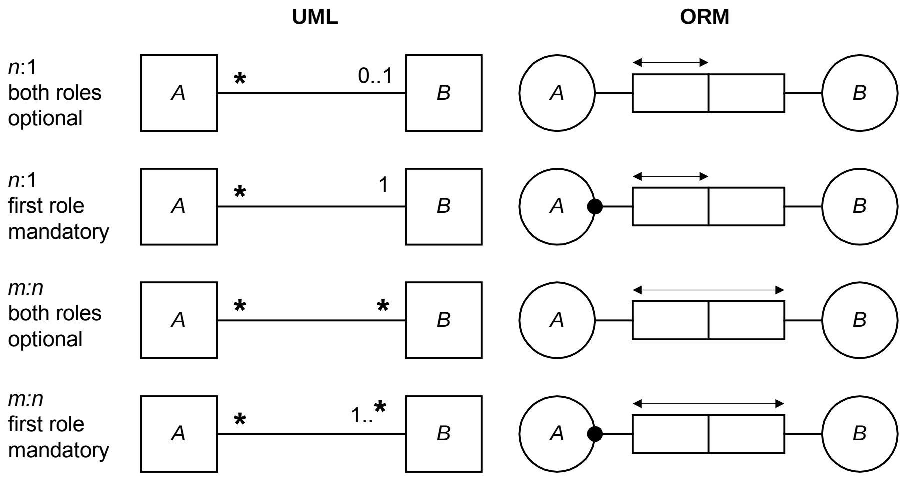

Figure 2: Some equivalent representations in UML and ORM

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 for a ternary association in which both Room-Time and Time-Activity pairs are unique.

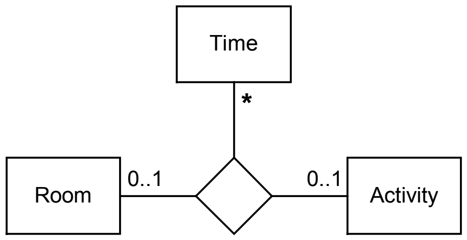

Figure 3: Multiplicity constraints on a ternary in UML

An ORM depiction of the same association is shown in Figure 4, along with two other associations and sample populations. Note how useful the population of the ternary is for checking the constraints. For example, if Time-Activity is not really unique, this can be tested by adding a counterexample.

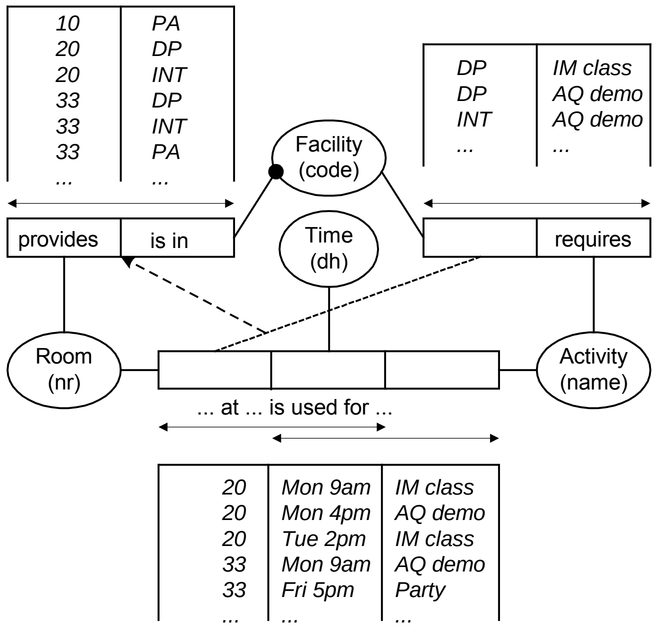

Figure 4: An ORM diagram with sample populations

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 two orthogonal notions of mandatory role constraints and frequency constraints. This separation is useful in localizing 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 need to be careful not to specify this constraint unless it is really needed. 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.

Uniqueness constraints are just frequency constraints with a frequency of 1, but are given a special notation because of their importance and ubiquity.

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. In Figure 5, for example, each academic is either tenured or contracted till some date. UML cannot express disjunctive mandatory role constraints graphically. Perhaps influenced by oids, UML does not specify any standard notation to mark *attribute uniqueness constraints* (candidate keys). It suggests that boldface might be used for this (or other purposes) as a tool extension ([28], p. 25). Another alternative is to annotate unique attributes with comments (e.g. {CK1}). It seems strange to have a standard notation for uniqueness when the feature is modeled as an association but not when it is modeled as an attribute.

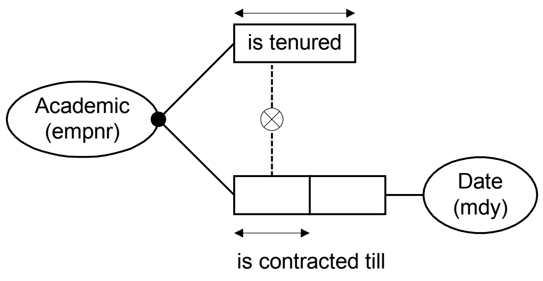

Figure 5: Disjunctive mandatory role constraint and exclusion constraint.

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 graphic expressibility of UML. ORM constraints that span different predicates are called *external constraints*. Only a few of these can be graphical expressed or emulated 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. For example, 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. UML is capable of expressing only basic subset constraints between binary associations, and its inability to project on the relevant roles invites modeling errors (e.g. [2], p. 68).

ORM allows *exclusion constraints* over a set of compatible role-sequences, by connecting "⊗" by dotted lines to the relevant role-sequences. A trivial example is given in Figure 5: no academic is both tenured and contracted. UML supports exclusion constraints only between roles played by the same object type, by connecting "OR" to the relevant associations by dashed lines ([28], p. 52). This notation is confusing (e.g. "or" here means "xor")[1]. Also consider the difference between the following two constraints: no person both wrote and reviewed; no person wrote and reviewed the same paper (ORM

[1] Subsequent to the original publication of this paper, version 1.3 of UML renamed the constraint "xor"

clearly distinguishes these by noting the precise arguments of the constraint). UML has no graphic notation for exclusion between attributes, or between attributes and associations (e.g. in Figure 5, the unary predicate must be modeled in UML as a boolean attribute, and the contract predicate would probably be modeled as a date attribute).

UML uses *qualified associations* in many cases where ORM uses an *external uniqueness* constraint for *co-referencing*. Figure 6 is based on an example from the standard document ([28], p. 59), 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).

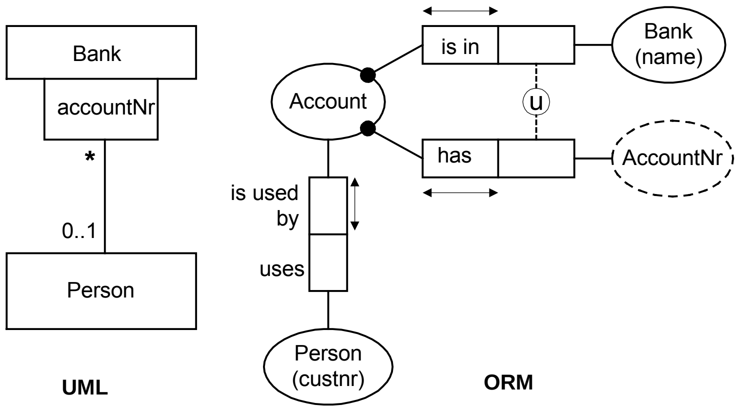

Figure 6: Qualified association in UML, and co-referenced object type in ORM

The UML notation is not only less clear, but 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. As a complicated example of this deficiency, see [2] (p. 51, Fig. 3.14) where the semantic connection between Node and nodeName is lost. The problem can be solved in UML by using an association class instead, though this is not always natural.

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. However 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 be to subtype Man and Woman from Person). This is not enough to formally guarantee that instances that populate these subtypes have the correct values for a sex attribute that might apply to Person. Moreover, much 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 formally captured in ORM but are ignored in UML, leading to the possibility of inconsistent UML models. For further discussion on these issues see [11, 15].

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:m 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). However UML makes the rather strange demand that both aggregation and composition be transitive and antisymmetric. ORM's modeling guidelines favor *direct* containment for base predicates (marked as intransitive and acyclic), defining full containment recursively as a derived predicate in the usual fashion to compute the transitive closure.

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 a sequence with unique members. In ORM there are two approaches to handle this. One way is to keep base predicates elementary, and annotate them with the appropriate constructor as an implementation instruction to the mapper. In this case 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 is the method we recommend, partly because elementarity allows individual instantiation and simplifies the semantics. The other way is to allow complex object types in the base model by applying constructors directly to them (e.g. [19]).

UML allows *default and initial values* to be declared for attributes, as well as allowing some attributes to be specified as *immutable*. Though not part of standard ORM, proposals to extend ORM to handle default information have been made [17], and it would be a trivial extension to cater for specification of initial values and immutability.

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 much vaster set of such annotations for class diagrams, that go into 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. Both UML and ORM include formal textual languages for expression of constraints, derivation rules and queries (see Appendix 2 for a comparative evaluation).

Conclusion

This paper identified a set of principles for evaluating modeling languages and applied them in evaluating UML and ORM for conceptual data modeling. ORM was generally found to be more expressive graphically, simpler, easier to validate (through verbalization and multiple instantiation) and more stable for both modeling and queries. However UML can offer a more compact notation, is gaining wide support in industry, especially for the design of object-oriented software, and includes several mechanisms for modeling behavior. Hence 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. Tool support is already available to transform between ORM, ER, Relational and Object-Relational models, and this is currently being extended to provide extensive support for UML, including transformations to and from ORM. Once this support is widely available, empirical studies are planned to study why and how practitioners choose and / or integrate modeling methods in practice.

Appendix 1: Background on UML and ORM

The Unified Modeling Language (UML) is largely derivative of earlier object-oriented modeling techniques. In 1996, a team at Rational Corporation led by Grady Booch, Jim Rumbaugh and Ivar Jacobson released an initial UML proposal that synthesized the Booch, OMT (Object Modeling Technique) and OOSE (Object-Oriented Software Engineering) methods. In September 1997, version 1.1 of UML was published by a consortium of several companies, who collaborated to refine and extend UML for evaluation by the Object Management Group (OMG) as a standard language for object-oriented analysis and design [26, 27, 28, 29]. Version 1.1 was added to the list of OMG Adopted Technologies in November 1997. UML is currently undergoing minor revisions by the OMG Revision Task Force, with versions 1.2 through 1.4 expected to be completed by April 1999 [23].

Any complete information modeling method must address the data, process and behavioral perspectives [22], and cover both analysis and design. To this end, UML provides a large suite of concepts and notations, including the following diagram types: Use case diagram; 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). As an extension, UML diagrams may be annotated with constraints in a textual language. UML provides the textual language OCL (Object Constraint Language) to enable constraints to be formally expressed, but does not mandate its use. Users may choose other formal languages or even informal, natural languages for this purpose. UML does not mandate a modeling process, but generally encourages a use-case driven, architecture-centric, iterative and incremental process.

Object Role Modeling (ORM) originated in the mid-1970s as a semantic modeling method, one of the early versions being NIAM [30], and has since been extensively revised and extended by many researchers. Overviews of ORM may be found in [12, 13, 14] and a detailed treatment in [11]. To better exploit the benefits of UML, or ER for that matter, ORM can be used for the conceptual analysis of business rules, and if desired, the resulting ORM model can be easily transformed into a UML class diagram or ER diagram. Although all versions of ORM are based on the same framework, minor variations do exist. For the purposes of this paper we focus on the most popular version of ORM as supported in modeling and query tools such as InfoModeler and ActiveQuery[2].

Appendix 2: Textual languages for constraints, derivation rules 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 joint submission, by IBM and ObjecTime Limited, to the OMG in January 1997. 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)
```

By contrast, in an algebraic language we would use axioms like:

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

Any practical constraint language must deal with undefined values. For example, the following specification should have the obvious meaning.

```
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 ($\mathbf{K}_3$). In $\mathbf{K}_3$, `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 $\mathbf{K}_3$

[2] InfoModeler and ActiveQuery are trademarks of Visio Corporation.

instead of classical logic means that theorems from standard mathematics do not apply—a high price to pay.

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 [21], PSM [18] 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:

$$\{\ p{:}\text{Product};\ m{:}\text{MoneyAmount} \mid \exists\ c{:}\ \text{MoneyAmount};\ w{:}\ \text{MoneyAmount} \bullet$$
$$p \text{ has cost of } c \wedge p \text{ has wholesale price of } w \wedge m = w - c\ \}$$

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:

$$\neg\ \exists\ p{:}\text{Product} \bullet \exists\ c{:}\ \text{MoneyAmount};\ w{:}\ \text{MoneyAmount} \bullet$$
$$p \text{ has cost of } c \wedge p \text{ has wholesale price of } w \wedge w / c < 1.3$$

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.

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. Fowler, M. with Scott, K. 1997, *UML Distilled*, Addison-Wesley.

10. Halpin, T. 1993, 'What is an elementary fact?', *Proc. First NIAM-ISDM Conf.*, G.Nijssen, J. Sharp eds, Utrecht.

11. Halpin, T. 1995, *Conceptual Schema and Relational Database Design, 2nd edn*, Prentice Hall Australia.

12. Halpin, T. 1996, 'Business rules and object-role modeling', *Database Prog. & Design*, 9, 10, Miller Freeman, 66-72.

13. Halpin, T. 1998, 'Object Role Modeling: an overview', white paper, www.visio.com/infomodeler.

14. Halpin, T. 1998, 'Object Role Modeling (ORM / NIAM)', *Handbook on Architectures of Information Systems*, Springer (to appear).

15. Halpin, T. & Proper, H. 1995, 'Subtyping and polymorphism in object-role modelling', *Data & Knowledge Engineering* 15, 3 (June), 251-281.

16. Halpin, T. & Proper, H. 1995, 'Database schema transformation and optimization', *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, 1021 (Dec.) 191-203.

17. Halpin, T. & Vermeir, D. 1997, 'Default reasoning in information systems', *Database Application Semantics*, R. Meersman & L. Mark eds, Chapman & Hall, London, 423-442.

18. 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.

19. 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.

20. 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.

21. Meersman, R. 1982, 'The RIDL conceptual language', Research report, Int. Centre for Information Analysis Services, Control Data Belgium Inc., Brussels, Belgium.

22. Olle, T.W., Hagelstein, J., Macdonald, I.G., Rolland, C., Sol, H.G., Van Assche, F.J.M., & Verrijn-Stuart, A.A. 1991, *Information Systems Methodologies: a framework for understanding*, 2nd edn, Addison-Wesley.

23. OMG UML Revision Task Force website, http: / / uml.systemhouse.mci.com / .

24. 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.

25. Silberschatz, A., Korth, F. & Sudarshan, S. 1996, 'Data models', *ACM Computing Surveys*, 28, 1 (Mar.), 105-8.

26. UML Partners 1997, *UML Summary*, version 1.1, OMG document ad / 97-08-03, www.omg.org.

27. UML Partners 1997, *UML Semantics*, version 1.1, OMG document ad / 97-08-04, www.omg.org.

28. UML Partners 1997, *UML Notation Guide*, version 1.1, OMG document ad / 97-08-05, www.omg.org.

29. UML Partners 1997, *Object Constraint Language Specification*, version 1.1, OMG document ad/97-08-08, www.omg.org.

30. Wintraeken, J. 1990. *The NIAM Information Analysis Method: Theory and Practice*, Kluwer, The Netherlands.