Data modeling in UML and ORM revisited

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

This paper first appeared in *Proc. EMMSAD'98 4th IFIP WG8.1International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, Heidelberg, Germany in June 1999.

Although the traditional entity relationship approach is still the most widely applied technique for modeling database applications, object-oriented approaches and fact-oriented approaches are being increasingly used for data modeling in general. The most popular exemplars of the latter two approaches are respectively the Unified Modeling Language (UML) and Object-Role Modeling (ORM). An initial, comparative evaluation of these approaches indicated that UML has benefits for object-oriented code design (e.g. implementation detail, including behavior), while ORM has advantages for conceptual data modeling (e.g. semantic stability, graphical expressibility; clarity and validation mechanisms). This paper further examines the relative strengths and weaknesses of ORM and UML for data modeling, focusing on attribute multiplicity, association arity, advanced constraints and subtyping. This analysis is given wider generality by addressing various language design principles (e.g. parsimony, orthogonality, convenience, expressibility) and illustrating how metamodel extensibility can be used to capture some features of one approach within the other.

Introduction

Although most suited to the design phase of object-oriented (OO) programming, the Unified Modeling Language (UML) can be used for database design in general, since when stripped of OO implementation details, its class diagrams provide an extended Entity-Relationship (ER) notation that can be annotated with database constructs (e.g. key declarations). As an Object Management Group standard [20], the UML notation includes diagrams for use cases, static structures (class, object), behavior (state chart, activity), interaction (sequence, collaboration), and implementation (component, deployment). Detailed discussion of these diagram types can be found in [2, 21]. 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.

While UML's object-oriented approach facilitates the transition to object-oriented code, a fact-orientated approach as exemplified by Object-Role Modeling (ORM) arguably provides a better way to capture and validate data concepts and business rules with domain experts, and to cater for structural changes in the application. By omitting the

attribute concept as a base construct, ORM allows communication in simple sentences, where each sentence type is easily populated with multiple instances. ORM pictures the world simply in terms of objects (entities or values) that play *roles* (parts in relationships). Overviews of ORM may be found in [10, 11] and a detailed treatment in [9].

A previous, comparative evaluation of these approaches [13] indicated that UML has benefits for object-oriented code design (e.g. implementation detail, including behavior), while ORM has advantages for conceptual data modeling (e.g. semantic stability, graphical expressibility; clarity and validation mechanisms). This result is perhaps not surprising, given that UML is primarily intended to support object-oriented software design, while ORM is intended primarily for conceptual analysis of data. There is no question that UML provides more complete support than ORM does for developing object-oriented code. However, UML is also promoted for conceptual data analysis and designing database applications in general [3], and it is in this area that we believe ORM is superior. This paper further examines the relative strengths and weaknesses of ORM and UML for data modeling, focusing on attribute multiplicity (section 2), association arity (section 3), advanced constraints and subtyping (section 4). An earlier version of some of this work and related topics is accessible in online form [12].

In [13] the following language design criteria, drawn from several sources (e.g. [17]), were used to compare the data modeling capabilities of UML and ORM: expressibility; clarity; semantic stability; semantic relevance; validation mechanisms; abstraction mechanisms; formal foundation. The following alternative yardsticks for language design are discussed in [1]: orthogonality; generality; parsimony; completeness; similarity; extensibility; openness. Some of these criteria (e.g. completeness, generality, extensibility) may be subsumed under expressibility. Language orthogonality, and to a lesser extent, parsimony, may be viewed as sub-principles of clarity (a measure of how easy it is to understand and use). To this list, we may add the sub-principle of convenience (how convenient, suitable or appropriate a language feature is to the user). The analysis in sections 2-4 pays especial attention to design principles of parsimony, orthogonality, convenience, and expressibility. We also illustrate how metamodel extensibility can be used to capture some features of one approach within the other (section 5). The conclusion summarizes the main points and identifies topics for future research.

Multi-valued attributes

Language design often involves a number of trade-offs between competing criteria. One well known trade-off is that between expressibility and tractability [18]: the more expressive a language is, the harder it is to make it efficiently executable. Another trade-off is between parsimony and convenience: although *ceteris paribus*, the fewer concepts the better (cf. Occam's razor), restricting ourselves to the minimum possible number of concepts may sometimes be too inconvenient. For example, two-valued propositional calculus allows for 4 monadic and 16 dyadic logical operators. All 20 of these operators can be expressed in terms of a single logical operator (e.g. nand, nor), but while this might be useful in building electronic components, it is too inconvenient for direct human

communication. For example, "not *p*" is far more convenient than "*p* nand *p*". In practice, we use several operators (e.g. not, and, or, if-then) since their convenience far outweighs the parsimonious benefits of having to learn only one operator such as nand. When it comes to proving meta-theorems about a given logic, it is often convenient to adopt a somewhat parsimonious stance regarding the base constructs (e.g. treat "not" and "or" as the only primitive logical operators), while introducing other constructs as derived (e.g. define "if *p* then *q*" as "not *p* or *q*"). Similar considerations apply to modeling languages.

One basic question relevant to the parsimony-convenience trade-off is whether to use the attribute construct in base modeling. A detailed argument in [13] favors a negative answer to this question, and is not repeated here. ORM models attributes in terms of relationships in its base model (for capturing, validating and evolving the conceptual schema), while allowing attribute-views to be displayed in derived models (in this case, compact views used for summary or implementation purposes). Traditional ER supports single-valued attributes, while UML supports single-valued and multi-valued attributes. Are multi-valued attributes a good idea in modeling? Let's consider an example.

Suppose we wish to record the names of employees, as well as the sports they play (if any). In ORM, we might model this situation as shown in Figure 1(a). ORM depicts entity types as named, solid ellipses, value types as named broken ellipses, and associations as named sequences of role boxes. If an entity type has a simple reference scheme, this may be shown implicitly in parentheses below the entity type name, or shown explicitly as a reference association. The implicit notation does not denote an attribute; it is just shorthand for the reference association. For example, "Sport(name)" abbreviates the injective association Sport is identified by SportName. The black dot is a mandatory role constraint indicating that each employee has a name. The absence of a mandatory role dot on the first role of the Plays fact type indicates that this role is optional (it is possible that some employee plays no sport). The arrow-tipped bar spanning the first role of Employee has EmpName is a uniqueness constraint indicating that each employee has at most one name. The uniqueness constraint spanning both roles of the plays predicate indicates this association is m:n (an employee may play many sports, and vice versa).

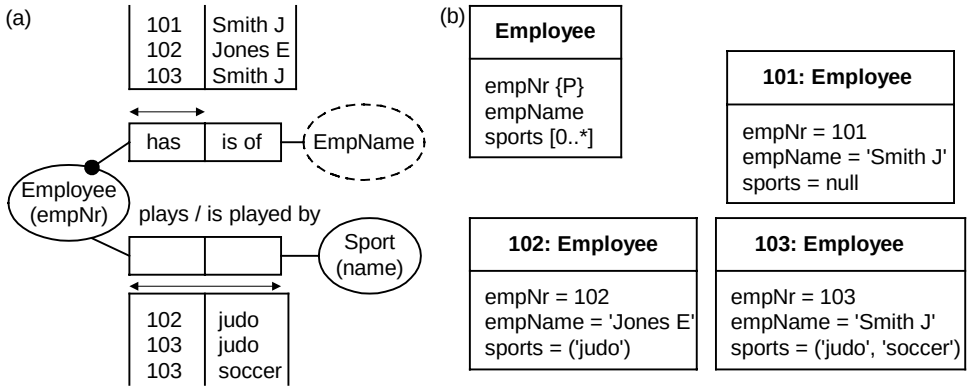

Figure 1: Employee plays Sport depicted as (a) ORM fact type and (b) UML multi-valued attribute.

One way of modeling the same situation in UML is shown in Figure 1(b). In the absence of any standard UML notation for primary reference, we use "{P}" for this purpose. The information about who plays what sport is modeled as the *multi-valued attribute* "sports". The "[0..*]" appended to this attribute is a *multiplicity constraint* indicating how many sports may be entered here for each employee. The "0" indicates that it is possible that no sports might be entered for some employee. Unfortunately, the UML standard uses a *null value* for this case, just like the relational model. The presence of nulls in the base UML model exposes users to implementation rather than conceptual issues, and adds considerable complexity to the semantics of updates and queries [7, 8]. By restricting its base structures to elementary fact types, and eschewing attributes, ORM avoids the notion of null values, enabling users to understand models and queries in terms of simple 2-valued logic. The "*" in "[0..*]" indicates there is no upper bound on the number of sports of a single employee. In other words, an employee may play many sports, and we don't care how many. The "0..*" constraint may be abbreviated as "*".

As Figure 1 shows, ORM allows sample populations to be displayed as fact tables, while UML shows populations as a set of object diagrams. Notice how much easier it is to check the constraints on the ORM diagram than on the UML diagram.

UML gives us the choice of modeling a multi-valued attribute as an association (as in ORM). For conceptual analysis and querying, this choice helps us verbalize, visualize and populate the associations. It also enables us to express various constraints involving the "role played by the attribute" in standard notation, rather than resorting to some non-standard extension (e.g. consider modeling the 1:n association Person is the best player of Sport using a multi-valued attribute). Associations also offer more stability. For example, consider the association Employee plays Sport in Figure 1(a). If we now want to record a skill level for this play, we can simply objectify this association as Play, and attach the fact type: Play has SkillLevel. Using an association class, a similar move can be made in UML if the play feature has been modeled as an association. In Figure 1(b), however, this feature has been modeled as the sports attribute; so this attribute needs to be removed and replaced by the equivalent association before we can add the new details about skill level.

Another problem with multi-valued attributes is that queries (and updates and constraints) on them need some way of extracting the components, and hence complicate formulation for users. As a trivial example, compare queries Q1, Q2 expressed in ORM's ConQuer language [4, 4] with their counterparts in OQL (the Object Query language proposed by ODMG [6]):

(Q1) **List each** Employee **who** plays Sport 'judo'.

(Q2) **List each** Sport **that** is played by Employee 103.

(Q1a) **select** x.empNr **from** x **in** Employee **where** "judo" **in** x.sports

(Q2a) **select** x.sports **from** x **in** Employee **where** x.empNr = 103

Although this example is trivial, the use of multi-valued attributes in more complex structures can make it harder for users to express their requirements. If we choose to avoid multi-valued attributes in our conceptual model, we still have the option of using them in the actual implementation. Both ORM and UML allow schemas to be annotated with instructions to over-ride the default actions of whatever mapper is used to transform the schema to an actual implementation. Since multi-valued attributes add complexity without adding expressibility, we suggest they be avoided in the conceptual model that is being validated by the domain expert.

Association arity

Some early versions of ORM [19], as well as most current versions of ER, restrict associations to binaries (arity = 2). UML allows binary and longer associations (arity > 1). ORM allows unary, binary and longer associations (arity > 0). Associations of any arity may be transformed into equivalent binary associations (possibly nested), so no expressibility is added by permitting non-binaries. On parsimony grounds, should we then restrict ourselves to binaries? We think not, since the convenience of using non-binaries is well worth it.

Consider the ternary association Room at Time is used for Activity, and suppose that each room at a time is used for at most one activity, and that at most one room is used at a given time for a given activity. Diagrams of this in both UML and ORM may be found in [13]. We could binarize this ternary by objectifying the sub-association between room and time and attaching activity to it as an attribute or association. Alternatively we could objectify the sub-association between time and activity and attach room to it. We could also create an artificial Schedule object type, and model the room, time and activity details as binary associations or attributes. However these choices complicate validation by verbalization and population, and make it difficult to express the constraints. Hence being able to express the association directly as a ternary has obvious benefits at the conceptual modeling phase.

What about unaries? You can replace them by binary associations, enumerated attributes (e.g. Booleans) or subtypes, but this can make it harder to express constraints or validate the model. As a trivial example, consider Figure 2(a), where the rule that patients who smoke are cancer-prone is expressed directly in ORM using two unaries and a subset constraint (shown as a dashed arrow). Figure 2(b) shows the same rule expressed in UML, using boolean attributes and a note. Apart from diagrammatic simplicity, ORM verbalizes the constraint formally as each Patient who smokes is cancer prone, and facilitates checking the rule with same populations.

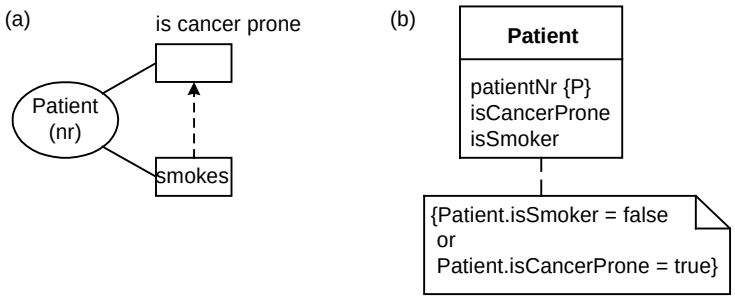

Figure 2: Patients who smoke are cancer prone, modeled in (a) ORM and (b) UML

Advanced constraints and subtyping

Figure 3 is the UML version of an OMT diagram used in [3, p. 68] to illustrate a subset constraint between associations. There are some obvious problems with the multiplicity constraints. For example, the "1" on the primary key association should be "0..1" (not all columns belong to primary keys), and the "*" on the define association should presumably be "1..*" (unless we allow tables with no columns). Assuming that tables and columns are identified by oids or artificial identifiers, the subset constraint makes sense, but the model is arguably sub-optimal since the primary key (PK) association and subset constraint can be replaced by a Boolean isaPKfield attribute on Column.

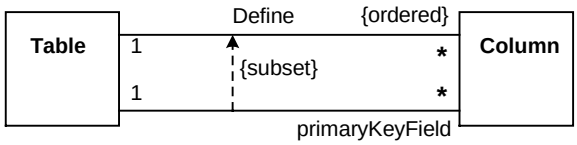

Figure 3: Spot anything wrong?

From an ORM perspective, heuristics lead us to initially model the situation using natural reference schemes as shown in Figure 4. Here ColName denotes the local name of the column in the table, and we have simplified reality by assuming tables may be identified just by their name. As seen by the external uniqueness constraints (circled "u"), two natural reference schemes for Column suggest themselves (name plus table, or position plus table). We can choose one of these as primary, or instead introduce an artificial identifier. The unary predicate indicates whether a column is, or is part of, a primary key. If desired, we could derive the association Column is a primary key field of Table from the path: Column is in Table **and** Column isaPKcol (the subset constraint from the previous model is then implied).

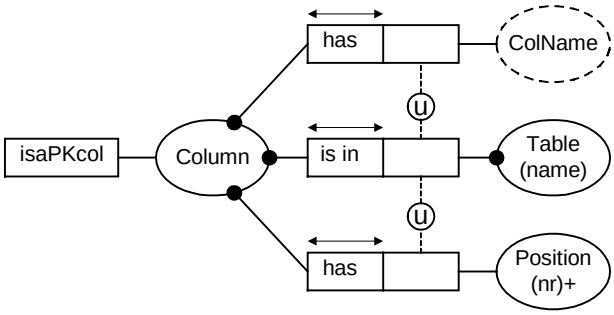

Figure 4: Alternative ORM model for schema shown in Figure 3

What is interesting about this example is not that the authors of the earlier model may have made some trivial errors with constraints, but rather the difference in modeling approaches. Most UML modelers seem to assume that oids will be used as identifiers in their initial modeling, whereas ORM modelers like to expose natural reference schemes right from the start, and populate their fact types accordingly. These different approaches often lead to different solutions. The main thing is to first come up with a solution that is natural and understandable to the domain expert, because here is where the most critical phase of model validation should take place. Once a correct model has been determined, optimization guidelines can be used to enhance it.

Another feature of the example is worth mentioning. The UML solution in Figure 3 uses the annotation "{ordered}" to indicate that a table is comprised of an *ordered set* (i.e. a sequence with no duplicates) of columns. In the ORM community, a long-standing debate has considered what is the best way to deal with collection type constructors (e.g. set, bag, sequence, unique sequence) at the conceptual level (e.g. [16]). Our view is that such constructors should not appear in the base conceptual model. Hence the use of Position in Figure 4 to convey column order (the uniqueness of the order is conveyed by the uniqueness constraint on Column has Position). Keeping fact types elementary has so many advantages (e.g. validation, constraint expression, flexibility and simplicity) that it seems best to relegate constructors to derived views. Constructors may also be added as an adornment to a pure conceptual model to specify implementation choices.

In ORM, an *equality constraint* between two compatible role sequences is shorthand for two subset constraints (one in either direction), and is shown as a double-headed arrow. Such a constraint indicates that the populations of the role-sequences must always be equal. If two roles played by an object type are mandatory, then an equality constraint between them is implied (and hence not shown). As a simple example of an equality constraint, consider Figure 5(a). Here the equality constraint indicates that if a patient's systolic blood pressure is measured, so is his/her diastolic blood pressure (and vice versa). In other words, either both measurements are taken, or neither. This kind of constraint is fairly common. Less common are equality constraints between sequences of two or more roles.

UML has no graphic notation for equality constraints. For whole associations we could use two separate subset constraints, but this would be messy. We could add a new notation, using "{equality}" besides a dashed arrow between the associations, but this notation would be unintuitive, since the direction of the arrow would have no significance (unlike the subset case). In general, equality constraints in UML would normally be specified as textual constraints (in braced comments). For our current example, the two blood pressure readings would typically be modeled as attributes of patient, and hence a textual constraint is attached to the Patient class as shown in Figure 5(b). This is awkward compared to the corresponding ORM constraint (graphic or verbalized). The situation could also be modeled in UML using a subtype for patients with blood pressure tested, or by introducing a blood pressure class with the pressures shown as attributes; however these approaches are rather artificial, and hinder validation.

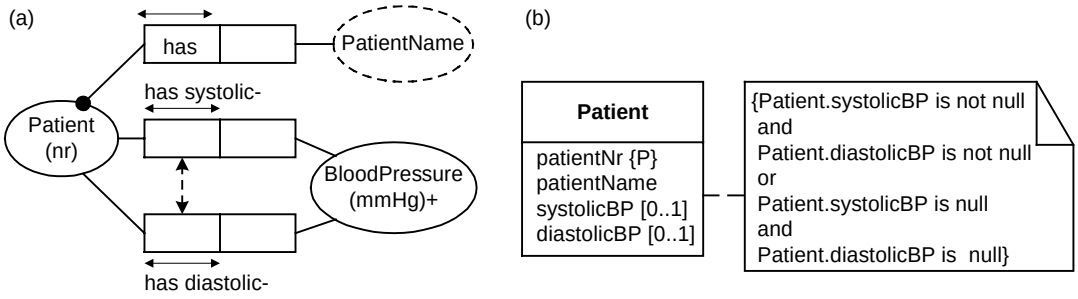

Figure 5: A simple equality constraint modeled in (a) ORM and (b) UML

In [13] it was shown that, apart from validation benefits, ORM's graphical constraint language is more expressive for conceptual data modeling than UML's graphical constraint notation (excluding notes). ORM's notation is also more orthogonal and general than UML's. To begin with, ORM mandatory constraints may be applied to a *set* of one or more roles (each object of that type must play at least of the indicated roles). UML allows mandatory constraints to be applied only to single association roles or single attributes, by declaring a minimum multiplicity above 0: for attributes, 1 is the default minimum multiplicity, but for association roles 0 is the default minimum. ORM's exclusion constraint (shown as ⊗) may be applied to any *set* of compatible *role sequences*, indicating at most one of these can be instantiated at a time, and its subset and equality constraints may be applied between any pair of compatible role sequences. UML allows subset constraints only between whole associations, and the only form of an exclusion constraint that it does provide is an exclusive-or constraint between single roles (shown as {xor}, with the meaning that exactly one is played). In ORM, an xor constraint is declared by orthogonally combining an exclusion constraint with a disjunctive mandatory role constraint.

In principle, an inclusive "{or}" constraint could be added to UML to express a mandatory disjunction between association roles; but this would not enable us to express a mandatory disjunction between attributes (or between association roles and attributes). A similar comment applies if we wish to extend UML with a mutual exclusion constraint.

And so on. In contrast, ORM's parsimonious decision to exclude attributes from its base constructs enables it to achieve great expressibility without undue complexity.

ORM's generic approach to constraints enables various classes of schema transformations to be stated and visualized in their most general form. For example, Figure 6 depicts a basic ORM equivalence [9, p. 331]. As an illustration of this theorem, consider the fact types Driver has Status {main, backup} and Driver drives Car, where each driver has exactly one status and drives exactly one car, and each car has two drivers, one main and one backup. Now transform this schema into the 1:1 fact types Driver is main driver of Car and Driver is backup driver of Car, where each driver plays exactly one role and each car plays two roles [9, p. 330]. For a formal discussion of ORM schema equivalence and optimization, see [15].

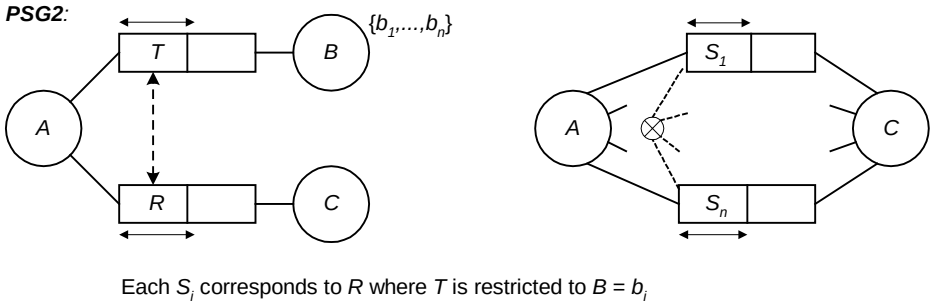

Figure 6: A basic schema equivalence in ORM

Both UML and ORM support *subtyping*, including multiple inheritance, using substitutability ("is-a") semantics. Both show subtypes outside, connected by arrows to their supertype(s), and allow declaration of constraints between subtypes such as exclusion and totality. In ORM, a subtype inherits all the roles of its supertypes. In UML, a subclass inherits all the attributes, associations and operations / methods of its supertype(s). Since our focus is on data modeling, not behavior modeling, we restrict our attention to inheritance of static properties (attributes and associations), ignoring operations or methods.

Subtypes are used in data modeling to assert typing constraints, encourage reuse of model components and show a classification scheme (taxonomy). In this context, typing constraints ensure that subtype-specific roles are played only by the relevant subtype. Using subtypes to show taxonomy is of little use, since taxonomy is often more efficiently captured by predicates. For example, the fact type Person is of Sex {male, female} implicitly provides the taxonomy for the subtypes MalePerson and FemalePerson.

Like other ER notations, UML provides only weak support for defining subtypes. A *discriminator* label may be placed near a subtype arrow to indicate the basis for the classification. For example, Figure 7 includes a "sex" discriminator to specialize Person into MalePerson and FemalePerson. This attribute is based on the enumerated type Sexcode, which has been defined using the stereotype «enumeration», and listing its values as attributes.

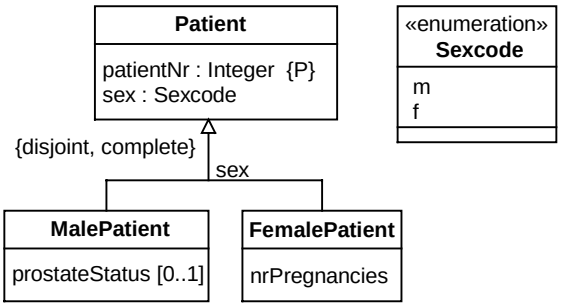

Figure 7: Subtyping in UML

By itself, this model fails to ensure that instances populating these subtypes have the correct sex. For example, there is nothing to stop us populating MalePatient with some patients that have the value 'f' for their sexcode. ORM overcomes this problem by requiring that *formal subtype definitions* be declared for all subtypes. These definitions must refer to roles played by the supertype(s). An ORM version of the correct schema is shown in Figure 8, together with a satisfying population. Note that an ORM partition (exclusion and totality) constraint is implied by the combination of the subtype definitions and the three constraints on the fact type Patient is of Sex.

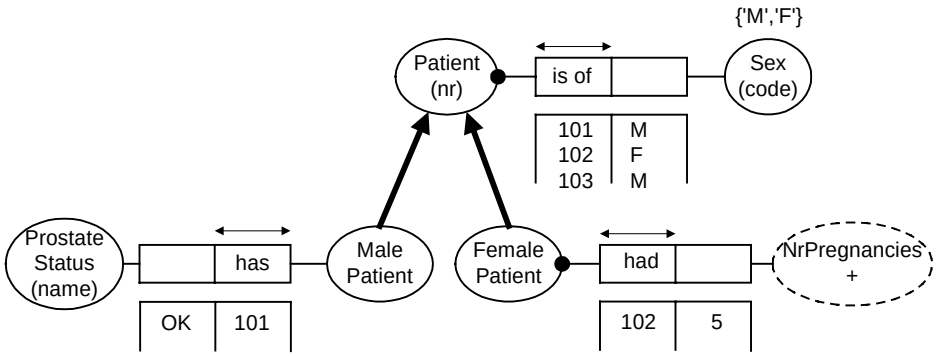

Figure 8: Formal subtype definitions are needed, and subtype partition constraints are implied

While the subtype definitions in Figure 8 are trivial, in practice more complicated subtype definitions are sometimes required. As a basic example, consider a schema with the fact types City is in Country, City has Population, where certain facts are to be recorded only for US cities with over a million people. The required subtype, LargeUScity, may be formally defined in ORM using the following ConQuer expression:

each LargeUScity **is a** City **that** is in Country 'US' **and** has Population > 1000000

There does not seem to be any convenient way of doing this in UML, at least not with discriminators. One could perhaps add a derived Boolean isLarge attribute, with an associated derivation rule in OCL, and then add a final subtype definition in OCL, but this would be less readable than the ORM definition above. For a detailed ORM perspective on these and other subtyping issues see [9, 14].

Meta-modeling

Because of ORM's greater expressive power, it is reasonably straightforward to capture UML models with an ORM framework. Though less convenient, it is possible to work in the other direction as well. To begin with, UML's graphic constraint notation can be supplemented by textual constraints in a language of choice (e.g. OCL). Moreover, the UML metamodel itself has built-in extensibility that allows many constraints specific to ORM to be captured within a UML based repository. As an example, the ORM model in Figure 9(a) contains four constraints numbered C1..C4, and four roles numbered r1..r4. Constraint C1 allows that a person wrote many books, and that a book was written by many persons. Constraint C2 asserts that each book was proofed by at most one person. Constraint C3 declares that if a book was proofed by somebody, it was also written by somebody (in this example, recorded authorship is optional, e.g. a book might be planned before assigning writers). The UML metamodel fragment in Figure 9(b) extends the standard UML metamodel by adding constraintNr, constraintKind and elementNr attributes, and adding ArgConstraint as a subtype along with the nrArguments attribute.

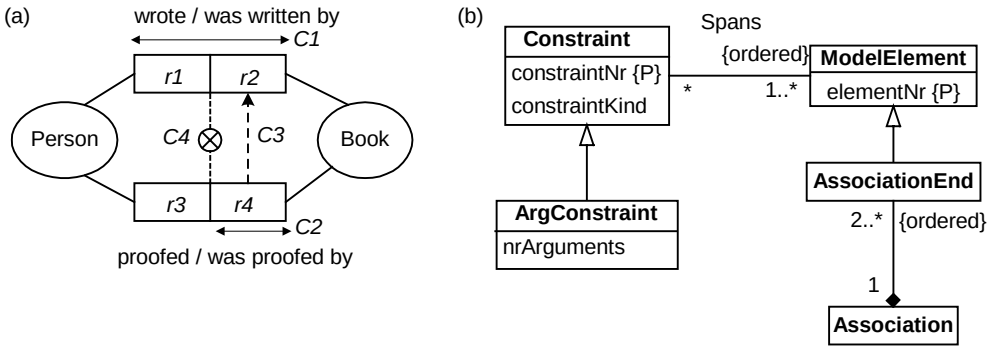

Figure 9: These ORM constraints (a) may be stored in an extended UML metamodel fragment (b)

The full UML metamodel [20] is very large, and we have included only that fragment relevant to our example. The attribute constraintKind stores the kind of constraint (subset, exclusion, mandatory etc.) and nrArguments is the number of arguments governed by the constraint. In this example, each argument is a sequence of one or more roles (in UML, a role is called an AssociationEnd). The four ORM constraints may now be stored as in the following object-relation:

Constraint:

constraintNr	*constraintKind*	*nrArguments*	*argumentsSpanned*
C1	uniquenessInternal	1	(r1, r2)
C2	uniquenessInternal	1	(r4)
C3	subset	2	(r4, r2)
C4	exclusion	2	(r1, r2, r3, r4)

Although nrArguments is partly determined by constraintKind, it is not fully determined (e.g. exclusion constraints may have two or more arguments). The argument list is divided by the number of arguments to determine the individual arguments, and constraintKind is used to determine the appropriate semantics. Though this simple example illustrates the basic idea, transforming the complete ORM metamodel into UML is complex. For example, as the UML metamodel fragment indicates, UML associations must have at least two roles (association ends), so rather artificial constructs must be introduced for dealing with unaries.

Conclusion

This paper extended a prior comparative evaluation of the data modeling facilities within UML and ORM, by examining multi-valued attributes, association arities, advanced constraints and subtyping, with particular reference to the language design principles of parsimony, expressibility, orthogonality and convenience. The following parsimonious approach to multi-valued attributes seems judicious: multi-valued attributes should be avoided in conceptual analysis, but may be used at the implementation level. A similar view was reached with regard to collection types (sets, bags etc.). Convenience dictates that associations of any arity (1 or above) should be allowed in conceptual modeling. ORM's constraint notation was found to be more orthogonal, partly because its notion of role unifies a concept treated as two separate notions in UML (within attributes and associations) and partly because its constraint primitives were chosen to apply orthogonally over sets of sequences or one or more roles. In spite of ORM's graphical advantages, UML can be used to capture specific ORM constraints either by use of a supplementary textual language, or by adapting its underlying metamodel using its built-in extensibility mechanisms.

For data modeling, ORM offers several advantages at the conceptual analysis phase, while UML provides greater functionality for specifying a data model at an implementation level suitable for the detailed design of object-oriented code. Hence both methods have value, and a complete development cycle may well profit by using ORM as a front end to UML. Automatic transformations between the two notations seems desirable, and research is currently under way to provide this. Once this support becomes available, empirical studies are planned to study why and how practitioners choose and / or integrate these modeling methods in practice.

References

1. Bentley, J. 1988, 'Little languages', *More Programming Pearls*, Addison-Wesley, Reading MA, USA.

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

3. Blaha, M. & Premerlani, W. 1998, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, New Jersey.

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. Cattell, R. & Barry, D. 1997, The Object Database Standard: ODMG 2.0, Morgan Kaufmann, San Francisco, CA.

7. Date, C. 1995, *Relational Database Writings 1991-1994*, Addison-Wesley, Reading MA, USA (see chapter 9).

8. Date, C. & Darwen, H. 1998, *Foundation for Object/Relational Databases: the Third Manifesto*, Addison-Wesley, Reading, MA, USA .

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

10. Halpin, T. 1998, 'Object Role Modeling: an overview', white paper, www.orm.net.

11. Halpin, T. 1998, 'Object Role Modeling (ORM / NIAM)', *Handbook on Architectures of Information Systems*, P. Bernus, K. Mertins & G. Schmidt eds, Springer-Verlag, Berlin, 81-101.

12. Halpin, T. 1998-9, 'UML data models from an ORM perspective', *Journal of Conceptual Modeling*, article series published online at www.inconcept.com.

13. Halpin, T. & Bloesch, A. 1998, 'A comparison of UML and ORM for data modeling', *Proc. EMMSAD-98: 3rd IFIP8.1 Int. Workshop on evaluation of modeling methods in systems analysis and design*, K. Siau, Y. Wand eds, Pisa, Italy.

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

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

16. 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.

17. 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.

18. Levesque, H. 1984, 'A fundamental trade-off in knowledge representation and reasoning', *Proc. CSCSI-84*, London, Ontario, 141-52.

19. Mark, L. 1987, 'The binary relationship model – 10th anniversary', *Tech. Report CS-TR-1933*, Univ. of Maryland.

20. OMG UML Revision Task Force, *OMG Unified Modeling Language Specification*, http://uml.systemhouse.mci.com/.

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