UML Data Models From An ORM Perspective: Part 1

by Dr. Terry Halpin, BSc, DipEd, BA, MLitStud, PhD
Director of Database Strategy, Visio Corporation

This paper appeared in the April 1998 issue of the Journal of Conceptual Modeling *published by Information Conceptual Modeling, Inc. and is reproduced here by permission.*

Although the Unified Modeling Language (UML) facilitates software modeling, its object-oriented approach is arguably less than ideal for developing and validating conceptual data models with domain experts. Object Role Modeling (ORM) is a fact-oriented approach specifically designed to facilitate conceptual analysis and to minimize the impact on change. Since ORM models can be used to derive UML class diagrams, ORM offers benefits even to UML data modelers. This multi-part article provides a comparative overview of both approaches.

Introduction

In our competitive and dynamic world, businesses require quality software systems that meet current needs and are easily adapted. These requirements are best met by modeling business rules at a very high level, where they can be easily validated with clients, and then automatically transformed to the implementation level. The *Unified Modeling Language* (UML) is becoming widely used for both database and software modeling, and version 1.1 was adopted in November 1997 by the Object Management Group (OMG) as a standard language for object-oriented analysis and design [11, 12, 13]. Initially based on a combination of the Booch, OMT (Object Modeling Technique) and OOSE (Object-Oriented Software Engineering) methods, UML was refined and extended by a consortium of several companies, and is undergoing minor revisions by the OMG Revision Task Force [10]. A simple introduction to UML is contained in [4], and a thorough discussion of OMT for database applications is given in [1], although its notation for multiplicity constraints differs from the UML standard.

UML includes diagrams for use cases, static structures (class and object diagrams), behavior (state-chart, activity, sequence and collaboration diagrams) and implementation (component and deployment diagrams). For data modeling purposes UML uses class diagrams, to which constraints in a textual language may be added. Although class diagrams may include implementation detail (e.g. navigation and visibility indicators), it is possible to use them for analysis by omitting such detail. When used in this way, class diagrams essentially provide an extended Entity Relationship (ER) notation.

UML's object-oriented approach facilitates the transition to object-oriented code, but can make it awkward to capture and validate business rules with domain experts. This problem can be remedied by using a fact-oriented approach where communication takes place in simple sentences, and each sentence type can easily be populated with multiple instances. *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 article is playing the role of being read.

ORM originated in the mid-1970s as a semantic modeling method, one of the early versions being NIAM (Natural language Information Analysis Method), and has since been extensively revised by many researchers. Overviews of ORM may be found in [6, 7] and a detailed treatment in [5]. Although all versions of ORM are based on the same framework, minor variations do exist. This article focuses on the most popular version of ORM as supported in modeling and query tools such as Visio's InfoModeler and ActiveQuery.

Since business requirements are subject to ongoing change, it is critical that the underlying data model be crafted in a way that minimizes the impact of these changes. The ORM framework is more stable under business changes than either OO or ER models, and facilitates the remaining changes that need to be made. This stability applies not only to the model itself, but also to conceptual queries based on the model.

Although ORM can be used independently of other methods, it may also be used in conjunction with them. To better exploit the benefits of UML, or ER for that matter, ORM can be used for the conceptual analysis of business rules, and the resulting ORM model can be easily transformed into a UML class diagram or ER diagram.

This article summarizes the main data modeling constructs in both ORM and UML, and discusses how they relate to one another. It aims to provide a basic understanding of both approaches and to illustrate translation between their notations. Along the way, some comparative advantages of ORM are noted. However this is not to disparage UML, which does have some nice features. Overall, UML provides a useful suite of notations for behavior and software modeling, and its class diagram notation is better than most other ER notations for data modeling. Visio Professional already provides basic support for several data and process modeling notations, and the integration of InfoModeler technology will enable very powerful support for both ORM and UML. So it will be possible to work in one or more of your preferred notations (ORM, UML, ER) with automatic mapping to an implementation in a variety of DBMSs. You could even do part of the model in ORM and part in UML, and have these merged to a single model.

This article is divided into parts, only the first of which appears in this issue. Part 1 focuses on the basic fundamentals. To provide an evaluation framework, some design criteria for modeling languages are first identified. We then discuss simple cases of how objects are referenced, and how single-valued “attributes” and can be captured in ORM and UML. From an ORM perspective, we confine our discussion of constraints to simple uniqueness and mandatory role constraints. From a UML perspective, we consider only attribute multiplicity and related textual constraints. Later parts will discuss UML associations and more advanced features such as other constraint types, aggregation, subtyping, derivation rules and queries.

Conceptual modeling language criteria

A modeling method comprises a language and also a procedure for using the language to construct models. Written languages may be graphical (diagrams) and/or textual. Conceptual models portray applications at a fundamental level, using terms and concepts familiar to the application users. In contrast, logical and physical models specify underlying database structures to be used for implementation, and external models specify user interaction details (e.g. design of screen forms and reports). 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 model all conceptually relevant details about the application domain. This is called the 100% Principle [9]. Object Role Modeling is primarily 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, 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. Such additional modeling capabilities of UML and ORM extensions are beyond the scope of this article, which focuses on the conceptual data perspective. For this perspective, ORM diagrams are graphically more expressive than UML class diagrams.

Moreover, ORM diagrams may be used in conjunction with the other UML diagrams, and may even be 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 [9]. *Validation mechanisms* are ways in which domain experts can check whether the model matches the application. For example, static features may be checked by verbalization and multiple instantiation, and dynamic features may be checked by simulation.

Abstraction mechanisms are ways in which unwanted details may be removed from immediate consideration. This is especially 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 [3, 5]. With minor variations, these techniques can be applied to both ORM and UML. ORM also includes an attribute abstraction procedure that can be adapted to generate a UML or ER diagram as a view.

A formal foundation ensures models are unambiguous and executable (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 [8]. 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 comparison focuses on the criteria of expressibility, clarity, stability, relevance and validation.

Object reference

For readers unfamiliar with ORM, some of its main concepts and notations are now summarized. These concepts will also help explain related UML notations. ORM classifies *objects* into *entities* (non-lexical objects) and *values* (lexical objects), and requires each entity to be identified by a well defined *reference scheme* used by humans to communicate about the entity. For example, employees might be identified by employee numbers or social security numbers, and countries by ISO country codes or country names. ORM uses "object", "entity" and "value" to mean "object *instance*", "entity instance" and "value instance", appending "*type*" for the relevant set of all possible instances. For example, you are an instance of the entity type Person. Entities might be referenced in different ways, and typically change their state over time. Glossing over some subtle points, values are constants (e.g. character strings and numbers) that basically denote themselves, so do not require a reference scheme to be declared.

Figure 1(a) depicts explicitly a simple reference scheme in ORM. Object types are shown as named ellipses, using solid lines for entity types (e.g. Employee) and dashed lines for value types (e.g. EmpNr). *Relationship* types are depicted as a named sequence of one or more *roles*, where each role appears as a box connected to the object type that plays it. The number of roles is called the *arity* of the relationship type. In ORM, relationships may be of any arity (1 = unary, 2 = binary, 3 = ternary, 4 = quaternary, 5 = quinary etc.). In base ORM, each relationship must be *elementary* (i.e. it cannot be split into smaller relationships covering the same object types without information loss). For this reason, arities above 5 are rare. In practice, about 80% of relationships are binary.

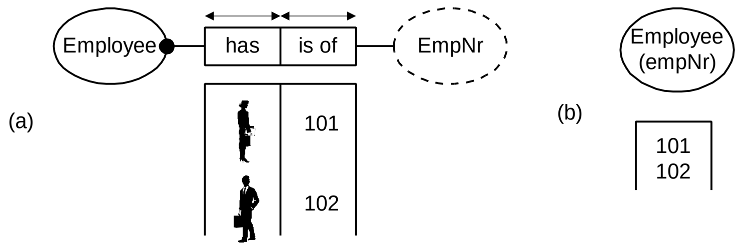

Figure 1: A simple reference scheme in ORM, shown (a) explicitly, (b) implicitly

Figure 1(a) depicts a binary relationship type. Read from left to right, we have: Employee has EmpNr. Read backwards, we have: EmpNr is of Employee. The verb phrases "has" and "is of" are *predicate* names. To enable navigation in any direction around an ORM schema, each n-ary relationship (n > 0) may be given *n* predicate names (one starting at each role), but it is a user preference as to how many of these are simultaneously displayed.

If an entity type has more than one candidate reference scheme, one of these may be declared *primary* to assist verbalization of instances (and sometimes to reflect actual business practice). If an entity type has only one candidate reference scheme, this is the primary one. Relationship types used for primary reference are called *reference types*. All other relationship types are called *fact types*. A primary reference scheme for an entity type maps each instance of that type onto a unique, identifying value (or a combination of values, as discussed in a later issue). In Figure 1(a), the reference type has a sample *population* shown below it in a *reference table* (one column for each role). Here icons are used to denote the real world employee entities.

To conserve space, simple reference schemes may be abbreviated by enclosing the *reference mode* in parentheses below the entity type name (see Figure 1(b)), and an object type's reference table includes values but no icons. References verbalize as existential sentences, e.g. "There is an Employee who has the EmpNr 101". The constraints in the reference scheme (see below) enable entity instances to be referenced elsewhere by definite descriptions, e.g. "The Employee who has the EmpNr 101".

Reference modes indicate the mode or manner in which values refer to entities (e.g. contrast Mass(kg) with Mass(lb)). The black dot where the left role connects to Employee is a *mandatory role* constraint, indicating that role must be played by all population instances of that type (verbalization: each employee has at least one employee number). The arrow-tipped bar over the left role is a *uniqueness constraint*, indicating that each instance in its associated population column appears there only once (verbalization: each Employee has at most one EmpNr). The uniqueness constraint on the right role indicates that each employee number refers to at most one employee. Hence the reference type provides an *injection* (mandatory, 1:1-into mapping) from Employee to EmpNr. The sample population clarifies the 1:1 property. A uniqueness constraint used for primary reference (e.g. the right-hand constraint in Figure 1(a)) may be annotated with a "P".

In a relational implementation, we might choose to use the primary reference scheme to provide value-based identity, or instead use row-ids (system generated, tuple identifiers). In an object-oriented implementation we might use *oids* (hidden, system generated object identifiers). Such choices can be added later as annotations to the model. For analysis and validation purposes however, we need to ensure that humans have a way of identifying objects in their normal communication.

It is the responsibility of humans (not the system) to enforce constraints on primary reference types. This is a conceptual, not an implementation issue. For instance, choosing employee numbers as external identifiers (or oids as internal identifiers) does not magically guarantee that each employee in the real world is actually assigned only one employee number (or only one oid). Various measures can be taken at the point of data entry to help ensure this, but even extreme measures such as DNA checks still have some possibility of error. However, assuming that humans do enforce the reference type constraints, the system may now be used to enforce fact type constraints.

UML classifies *instances* into *objects* and *data values*. UML objects basically correspond to ORM entities, but are assumed to be identified by oids. UML data values basically correspond to ORM values: they are constants (e.g. character strings or numbers) and hence require no oids to establish their identity. Entity types in UML are called *classes*, and value types are called data types. Note that “object” means “object instance”, not “object type”. A relationship instance in UML is called a *link*, and a relationship type is called an *association*.

Because of reliance on oids, UML does not require entities to have a value-based reference scheme. This can make it impossible to communicate naturally at the instance level, and ignores the real world database application requirement that humans have a verbal way of identifying objects. It is important therefore to include value-based reference in any UML class diagram intended to capture all the conceptual semantics about a class. Unfortunately, to do this we often need to introduce non-standard extensions to the UML notation, as seen in the following example.

Single-valued attributes

Like other ER notations, UML allows relationships to be modeled as *attributes*. For instance, in Figure 2(b) the Employee class has eight attributes. Classes in UML are depicted as a named rectangle, optionally including other compartments for attributes and operations. For now, we ignore operations in our discussion. The corresponding ORM diagram is shown in Figure 2(a). True to its name, ORM models the world in terms of just objects and roles, and hence has only one data structure– the relationship type. 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.* As a consequence, ORM diagrams typically take up more room than corresponding UML or ER diagrams, as Figure 2 illustrates. But this is a small price to pay for the resulting benefits. Before discussing these advantages, let’s see how to translate between the relevant notations.

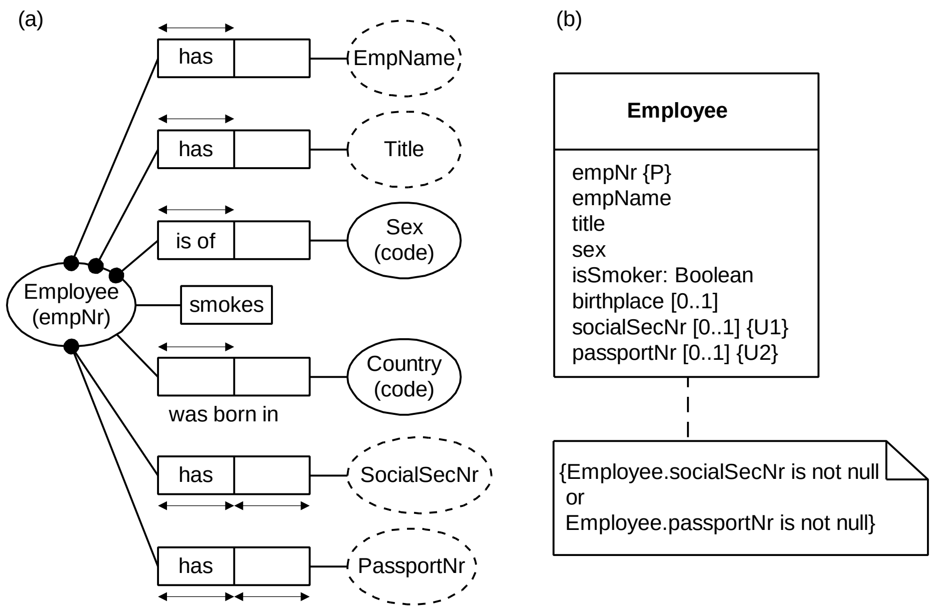

Figure 2: ORM relationship types (a) depicted as attributes in UML (b)

The ORM model indicates that employees are identified by their employee numbers. The top three mandatory role constraints indicate that every employee in the database must have a name, title and sex. The other black dot where two roles connect is a *disjunctive mandatory role constraint*, indicating that the disjunction of these roles is mandatory (each employee has a social security number or passport number, or both). Although each of these two roles is individually optional, at least one of them must be played.

In UML, attributes are mandatory by default. In the ORM model, the unary predicate "smokes" is optional (not everybody has to smoke). UML does not support unary relationships, so it models this instead as the Boolean attribute "isSmoker". In UML the domain of any attribute may optionally be displayed after it (preceded by a colon). In this example, we showed the domain only for the isSmoker attribute. By default, InfoModeler takes a closed world approach to unaries, which agrees with the isSmoker attribute being mandatory. The ORM model also indicates that Sex and Country are identified by codes (rather than names, say). We could convey some of this detail in the UML diagram by appending domain names. For example, "Sexcode" and "Countrycode" could be appended after "sex: " and "birthplace: " to provide syntactic domains.

In the ORM model it is optional whether we record birthplace, social security number or passport number. This is captured in UML by appending [0..1] after the attribute name (each employee has 0 or 1 birthplace, and 0 or 1 social security number). This is an example of an *attribute multiplicity* constraint. UML does not have a graphic notation for disjunctive mandatory roles, so this kind of constraint needs to be expressed textually in an attached note (see bottom of Figure 2(b)). Such *textual constraints* may be expressed informally, or in some formal language interpretable by a tool. In the latter case, the constraint is placed in braces. Although UML provides the Object Constraint Language (OCL) for this purpose, it does not mandate its use, allowing users to pick their own language (even programming code). This of course weakens the portability of the model. Moreover, the readability of the constraint is typically poor compared with the ORM verbalization (**each** Employee has **a** SocialSecNr **or** has **a** PassportNr).

The uniqueness constraints over the left-hand roles in the ORM model (including the empnr reference scheme shown explicitly earlier) indicate that each employee has at most one employee number, employee name, title, sex, country of birth, social security number and passport number. Unary predicates have an implicit uniqueness constraint; so each employee instantiates the smokes role at most once (for any given state of the database). All these uniqueness constraints are implicitly are captured in the UML model, where attributes are single-valued by default (multi-valued attributes will be discussed in a later issue).

The uniqueness constraints on the right-hand roles (including the empnr reference scheme) indicate that each employee number, social security number and passport number refers to at most one employee. UML does not have a standard graphic notation for these "*attribute uniqueness constraints*". It suggests that boldface could be used for this (or other purposes) as a tool extension ([12], p. 25), but clearly this is not portable. We have chosen our own notation for this, appending textual constraints in braces after the attribute names (P = primary identifier, U = unique, with numbers appended if needed to disambiguate cases where the same U constraint might apply to a combination of attributes). The use of "P" here does not imply the model must be implemented in a relational database using value primary keys; it merely indicates a primary identification scheme that may be used in human communication.

Because UML does not provide standard graphic notations for such constraints, and it leaves it up to the modeler whether such constraints are specified, it is perhaps not surprising that many UML models one encounters in practice simply leave such constraints out.

Now that we've seen how single-valued attributes are modeled in UML, let's briefly see why ORM refuses to use them in its base modeling. The main reasons may be summarized thus:

- Attribute-free models are more stable
- Attribute-free queries are more stable
- Attribute-free models are easy to populate with multiple instances
- Attribute-free models facilitate verbalization in sentences
- Attribute-free models highlight connectedness through semantic domains
- Attribute-free models are simpler and more uniform
- Attribute-free models make it easier to specify constraints

- Attribute-free models avoid arbitrary modeling decisions
- Attribute-free models may be used to derive attribute views when desired

Let's begin with semantic stability. ORM models and queries are inherently *more stable*, because they are free of changes caused by attributes evolving into entities or relationships, or vice versa. Consider the ORM fact type: Employee-was-born-in-Country. In ER and OO approaches we might model this using a birthplace attribute (e.g. Figure 2(b)). If we later decide to record the population of a country, then we need to introduce Country as an entity type. In UML, the connection between birthplace and Country is now unclear. Partly to clarify this connection, we would probably reformulate our birthplace attribute as an association between Employee and Country. This is a significant change to our model. Moreover, any object-based queries or code that referenced the birthplace attribute would also need to be reformulated.

Another reason for introducing a Country class is to enable a listing of countries to be stored, identified by their country codes, without requiring all of these countries to participate in a fact. To do this in ORM, we simply declare the Country type to be independent (this is displayed by appending "!" to the type name). The object type Country may be populated by a reference table that contains those country codes of interest (e.g. 'AU' denotes Australia).

A typical counter-argument is this: "Good ER or OO modelers would declare country as an object type in the first place, anticipating the need to later record something about it, or to maintain a reference list; on the other hand, features such the title and sex of a person clearly are things that will never have other properties, and hence are best modeled as attributes". This attempted rebuttal is flawed. In general, you can't be sure about what kinds of information you might want to record later, or about how important some feature of your model will become. Even in the title and sex case, 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 kind of constraint can be captured graphically as a join-subset constraint between the relevant fact types (see later issue), 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). In contrast, attribute usage hinders expression of the relevant restriction association (try expressing and populating this rule in UML).

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 [2]. For example, to list the employees born in a country with a population below ten million, we may formulate our query in ORM thus: **list each** Employee **who** was born in **a** Country **that** has **a** Population < 10000000.

Avoiding attributes also leads to greater *simplicity* and uniformity. For example, we don't need notations to reformulate constraints on relationships into constraints on attributes or between attributes and relationships (more about this in a later issue). Another reason is to minimize arbitrary modeling choices (even experienced modelers sometimes disagree about whether to model some feature as an attribute or relationship).

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, a predicate appears as a named, contiguous sequence of one or more role boxes. Since these boxes are set out in a line, fact types may be conveniently populated with fact 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 can thus take place in a familiar language, backed up by population checks. The practical value of these validation checks is considerable, especially since many clients find it much easier to work with instances rather than types. As discussed in the next issue, attributes and UML-style associations make it harder to populate models with multiple instances, and often lead to unnatural verbalization. UML does provide object diagrams for discussing single instances, but these are of little use for discussing populations with multiple instances.

For summary purposes, ORM includes algorithms for dynamically generating ER-style diagrams as attribute-views [3, 5]. 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. Modeling and maintenance are iterative processes. The importance of a feature can change with time as we discover more of the global model, and the application being modeled itself changes. To promote semantic stability, ORM makes no commitment to relative importance in its base models, instead supporting this dynamically through views. 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.

In short, you can have your cake and eat it too, by using ORM for analysis, and if you want to work with UML class diagrams, you can use your ORM models to derive them.

Later issues

We've barely scratched the surface of UML or ORM, but many of the fundamentals have been introduced. In later issues, we'll compare UML associations with ORM predicates, fact tables with object diagrams, UML multiplicity constraints with ORM mandatory and frequency (including uniqueness) constraints, UML association classes with ORM nesting, and UML qualified associations with ORM co-referencing. We'll also discuss more advanced constraints, aggregation, subtyping, derivation rules and queries.

References

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

2. Bloesch, A. & Halpin, T. 1997, 'Conceptual queries using ConQuer-II', *Proceedings of the 16th International Conference on Conceptual Modeling ER'97* (Los Angeles), D. Embley, R. Goldstein eds, Springer LNCS 1331 (Nov.) 113-126.

3. Campbell, L., Halpin, T. & Proper, H. 1996, 'Conceptual schemas with abstractions: making flat conceptual schemas more comprehensible', *Data & Knowledge Engineering*, 20, 1, 39-85.

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

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

6. Halpin, T. 1996, 'Business rules and Object Role modeling', *Database Prog. & Design*, 9, 10, (Miller Freeman, San Mateo CA), 66-72.

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

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

9. 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.

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

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

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

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

This paper is made available by Dr. Terry Halpin and is downloadable from www.orm.net.