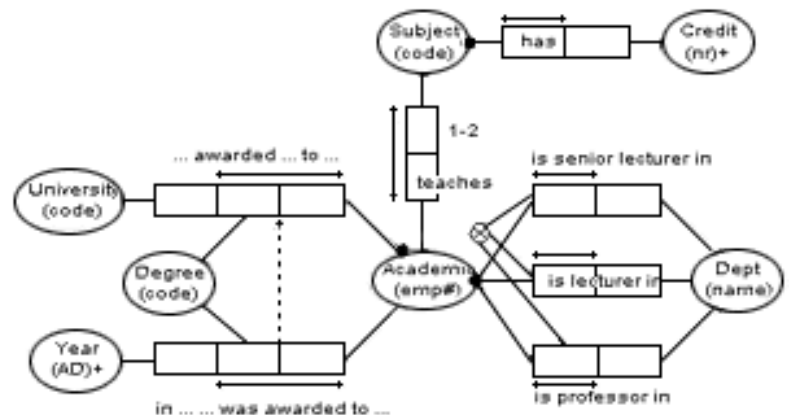



[ORM in Detail](#)
[Modeling Issues](#)
[Conceptual Queries](#)
[UML and ORM](#)
[Resources](#)

ORM IN DETAIL

This section provides a basic overview of Object Role Modeling (ORM), through white papers, articles and a slide presentation. The relationship between ORM and the Entity Relationship (ER) approach is also discussed.



For an in-depth treatment of ORM, see Halpin, T.A. 2001, *Information Modeling and Relational Databases*, published by [Morgan Kaufmann Publishers](#) (ISBN 1-55860-672-6). Details on this book are available at the [book's website](#).

Business Rules and Object Role Modeling

This paper was published in the October 1996 issue of Database Programming & Design, vol. 9, no. 10, pp. 66-72.

This article provides a gentle introduction to Object Role Modeling (ORM), explaining its advantages over entity relationship and object oriented approaches for capturing and validating business rules with subject matter experts. The ORM attribute-free, mixfix predicate approach simplifies verbalization, multiple instantiation and schema evolution, and its rich constraint language enables many rules to be captured graphically and easily validated. The article includes simple examples to illustrate these advantages.



[Business Rules and Object Role Modeling](#) (407K)

Object Role Modeling: An Overview

This white paper provides an overview of Object Role Modeling (ORM), using a case study to illustrate the main ideas. The steps used to design a conceptual schema for an information system are first explained in some detail. To help communicate the ideas, some mistakes are deliberately made. Checking procedures within the design method are later used to remove these errors. A simple example is included to show how the conceptual design may be "optimized" for relational database systems by applying a transformation. An algorithm for mapping this design to a normalized, relational database schema is then outlined. Finally, the paper gives a brief sketch of how ORM can be used as a sound basis for conceptual queries, object oriented modeling, and process/event modeling.



[Object Role Modeling: An Overview](#) (405K)

Microsoft has also published a [Revised version of the above ORM overview](#) as well as a [Quick overview of ORM](#).

Object Role Modeling (ORM/NIAM)

This paper first appeared as chapter 4 of the following book: Bernus, P., Mertins, K. & Schmidt, G. (eds.) 1998, Handbook on Architectures of Information Systems, Springer. Details on this publication are available from [Springer's website](#)

Object Role Modeling (ORM) is a method for modeling and querying an information system at the conceptual level, and mapping between conceptual and logical (e.g. relational) levels. ORM comes in various flavors, including NIAM (Natural language Information Analysis Method). This article provides an overview of ORM, and notes its advantages over entity relationship and traditional object oriented modeling.

This paper and the Object Role Modeling Overview paper overlap substantially; however, the historical details, symbol summary, and references provide additional material.




[Object Role Modeling \(ORM/NIAM\)](#) (125K)

Entity Relationship modeling from an ORM perspective: Part 1

This is a revised version of an article that first appeared in the December 1999 issue of the [Journal of Conceptual Modeling](#).


This paper is the first in a series of articles examining data modeling in the Entity relationship (ER) approach from the perspective of Object Role Modeling (ORM). After a brief historical introduction, this article examines basic aspects of the Barker notation for ER.

 [Entity Relationship modeling from an ORM perspective: Part 1](#)
(80K)

Entity Relationship modeling from an ORM perspective: Part 2

This article first appeared in the February 2000 issue of the [Journal of Conceptual Modeling](#).

It is the second in a series of articles examining data modeling in the Entity relationship (ER) approach from the perspective of Object Role Modeling (ORM). This paper completes a basic review of the Barker notation for ER.

 [Entity Relationship modeling from an ORM perspective: Part 2](#)
(58K)

Entity Relationship modeling from an ORM perspective: Part 3

This article first appeared in the April 2000 issue of the [Journal of Conceptual Modeling](#).

It is the third in a series of articles examining data modeling in the Entity relationship (ER) approach from the perspective of Object Role Modeling (ORM). This paper discusses the Information Engineering (IE) notation for ER.



[Entity Relationship modeling from an ORM perspective: Part 3](#)
(55K)

Modeling business rules using fact-orientation and object-orientation

This PowerPoint presentation was delivered at the Business Rules Conference hosted by the [Data Resource Management Association](#) in May, 1999. The original DRMA99.pps file (926kb) has been zipped for downloading (still a hefty 798kb), and needs to be unzipped (e.g. using Winzip) before it can be viewed.

[DRMA99 slide presentation \(798k\)](#)

[ORM Home](#) [ORM in Detail](#) [Modeling Issues](#)
[Conceptual Queries](#) [UML and ORM](#) [Resources](#)

All diagrams on this site were created with Microsoft Visio.

ORM

ORM in Detail

Modeling
IssuesConceptual
Queries

UML and ORM

Resources

MODELING ISSUES

These articles address various data modeling concepts such as higher-order types, objectified associations, join constraints, elementary fact types, subtyping, conceptual schema transformation/optimization, schema abstraction, and modeling with collection types.

Information Modeling and Higher-Order Types

This paper first appeared in Proc. EMMSAD'04: 8th Int. IFIP WG8.1 Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, and is reproduced here by permission.

While some information modeling approaches (e.g. the Relational Model, and Object-Role Modeling) are typically formalized using first-order logic, other approaches to information modeling include support for higher-order types. There appear to be three main reasons for requiring higher-order types: (1) to permit instances of categorization types to be types themselves (e.g. the Unified Modeling Language introduced power types for this purpose); (2) to directly support quantification over sets and general concepts; (3) to specify business rules that cross levels/metalevels (or ignore level distinctions) in the same model. As the move to higher-order logic may add considerable complexity to the task of formalizing and implementing a modeling approach, it is worth investigating whether the same practical modeling objectives can be met while staying within a first-order framework. This paper examines some key issues involved, suggests techniques for retaining a first-order formalization, and also makes some suggestions for adopting a higher-order semantics



[Information Modeling and Higher-Order Types](#) (538K)

Uniqueness Constraints on Objectified Associations

This article first appeared in the October 2003 issue of the [Journal of Conceptual Modeling](#).

Unlike UML and some ER versions, ORM currently allows a fact type to be objectified only if it either has a spanning uniqueness constraint or is a 1:1 binary fact type. This article argues that this restriction should be relaxed, and replaced by a modeling guideline that allows some n-ary associations to be objectified even if their longest uniqueness constraint spans n-1 roles. The pros and cons of removing this restriction are discussed, and illustrated with examples.



[Uniqueness Constraints on Objectified Associations](#) (341K)

Join Constraints

This paper first appeared in Proc. EMMSAD'02: 7th Int. IFIP WG8.1 Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, and is reproduced here by permission.

Many application domains involve constraints that, at a conceptual modeling level, apply to one or more schema paths, each of which involves one or more conceptual joins (where the same conceptual object plays roles in two relationships). Popular information modeling approaches typically provide only weak support for such join constraints. This paper contrasts how join constraints are catered for in Object-Role Modeling (ORM), the Unified Modeling Language (UML), the Object-oriented Systems Model (OSM), and some popular versions of Entity-Relationship modeling (ER). Three main problems for rich support for join constraints are identified: disambiguation of schema paths; disambiguation of join types; and mapping of join constraints. To address these problems, some notational, metamodel, and mapping extensions are proposed.



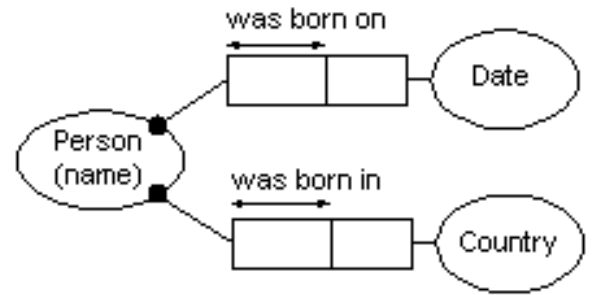
[Join Constraints](#) (364K)

What is an elementary fact?

This paper was presented in 1993 at the first NIAM-ISDM

conference.

Database schemas are best designed by mapping from a high level, conceptual schema expressed in human-oriented concepts. While conceptual schemas are often specified using entity relationship modeling (ER), a more natural and expressive formulation is often possible using Object Role Modeling (ORM). This approach views the world in terms of objects playing roles, and traditionally expresses all information in terms of elementary facts, constraints and derivation rules. Although verbalization in terms of elementary facts has many practical and theoretical advantages, it is difficult to define the notion precisely. This paper examines various awkward but practical cases which challenge the traditional definition. In so doing, it aims to clarify what elementary facts are and how they can be best expressed.

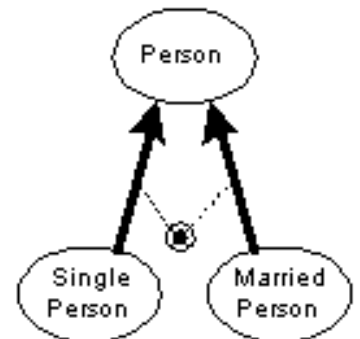


 [What is an elementary fact?](#) (73K)

Subtyping: conceptual and logical issues

This paper first appeared in vol. 23, no. 6 of Database Newsletter. This newsletter has since been renamed DataToKnowledge Newsletter and is published by [Business Rules Solutions, Inc.](#)

Subtyping is an important feature of semantic approaches to conceptual schema design and, more recently, object-oriented database design. However the relational model does not directly support subtyping, and CASE tools for mapping conceptual to relational schemas typically provide only very weak support for mapping subtypes. This paper surveys some of the main issues related to conceptual specification and relational mapping of subtypes, and



indicates how Object Role Modeling solves the associated problems.



[Subtyping: conceptual and logical issues](#) (132K)

Subtyping and Polymorphism in Object Role Modeling

This paper, co-authored with H.A. Proper, was first published in [Data & Knowledge Engineering](#), 15(3), 251-281, 1995, North-Holland, Amsterdam.

Although entity relationship (ER) modeling techniques are commonly used for information modeling, Object Role Modeling (ORM) techniques are becoming increasingly popular, partly because they include detailed design procedures providing guidelines for the modeler. As with the ER approach, a number of different ORM techniques exist. In this paper, we propose an integration of two theoretically well founded ORM techniques: FORM and PSM. Our main focus is on a common terminological framework, and on the notion of subtyping. Subtyping has long been an important feature of semantic approaches to conceptual schema design. It is also the concept in which FORM and PSM differ the most in their formalization. The subtyping issue is discussed from three different viewpoints covering syntactical, identification, and population issues. Finally, a wider comparison of approaches to subtyping is made, which encompasses other ER-based and ORM-based information modeling techniques, and highlights how formal subtype definitions facilitate a comprehensive specification of subtype constraints.



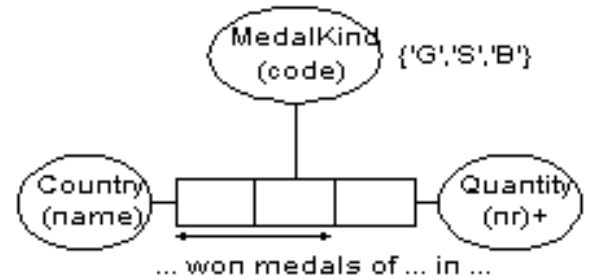
[Subtyping and Polymorphism in Object Role Modeling](#)
(253K)

Database schema transformation and optimization

This paper first appeared in Proc. OOER'95: Object-Oriented and

Entity-Relationship Modeling, [Springer LNCS](#), vol. 1021, pp. 191-203.

An application structure is best modeled first as a conceptual schema, and then mapped to an internal schema for the target DBMS. Different but equivalent conceptual schemas often map to different internal schemas, so performance may be improved by applying conceptual transformations prior to the standard mapping. This paper discusses recent advances in the theory of schema transformation and optimization within the framework of ORM (Object Role Modeling). New aspects include object relativity, complex types, a high level transformation language and update distributivity.



[Database schema transformation and optimization](#) (115K)

Conceptual Schemas with Abstractions: Making flat conceptual schemas more comprehensible

This paper, co-authored with L.J. Campbell and H.A. Proper, first appeared in [Data & Knowledge Engineering](#), 20(1), 39-85, 1996, North-Holland, Amsterdam.

Flat graphical, conceptual modeling techniques are widely accepted as visually effective ways in which to specify and communicate the conceptual data requirements of an information system. Conceptual schema diagrams provide modelers with a picture of the salient structures underlying the modeled universe of discourse, in a form that can readily be understood by and communicated to users, programmers and managers. When complexity and size of applications increase, however, the success of these techniques in terms of comprehensibility and communicability deteriorates rapidly.

This paper proposes a method to offset this deterioration, by

adding abstraction layers to flat conceptual schemas. We present an algorithm to recursively derive higher levels of abstraction from a given (flat) conceptual schema. The driving force of this algorithm is a hierarchy of conceptual importance among the elements of the universe of discourse.



[Conceptual Schemas with Abstractions](#) (427K)

Modeling Collections in UML and ORM

This paper first appeared in Proc. EMMSAD'00: 5th Int. IFIP WG8.1 Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, and is reproduced here by permission.

Collection types such as sets, bags and arrays have been used as data structures in both traditional and object oriented programming. Although sets were used as record components in early database work, this practice was largely discontinued with the widespread adoption of relational databases. Object-relational and object databases once again allow database designers to embed collections as database fields. Should collections be specified directly on the conceptual schema, as mapping annotations to the conceptual schema, or only on the logical database schema?

This paper discusses the pros and cons of different approaches to modeling collections. Overall it favors the annotation approach, whereby collection types are specified as adornments to the pure conceptual schema to guide the mapping process from conceptual to lower levels. The ideas are illustrated using notations from both object-oriented (Unified Modeling Language) and fact-oriented (Object-Role Modeling) approaches.



[Modeling Collections in UML and ORM](#) (106K)

[ORM Home](#) [ORM in Detail](#) [Modeling Issues](#)
[Conceptual Queries](#) [UML and ORM](#) [Resources](#)

All diagrams on this site were created with Microsoft Visio.


 ORM

ORM in Detail

Modeling Issues

Conceptual Queries

UML and ORM

Resources

CONCEPTUAL QUERIES

These papers describe the fundamentals of building conceptual queries within the ORM context.

ConQuer, the language on which these queries are based, makes query building easy for end users and database administrators.

```

✓ Department
  └─ employees ✓ Employee
        └─ achieves Rating
              └─ max(Rating)for Employee >
                  avg (Rating)for Department
  
```

Conceptual Queries

This paper first appeared in vol. 26, no. 2 of Database Newsletter. This newsletter has since been renamed Business Rules Journal and is published by [Business Rules Solutions, Inc.](#)

Formulating non-trivial queries in relational languages such as SQL or QBE can prove daunting to end users. ConQuer, a new conceptual query language based on Object Role Modeling (ORM), enables users to pose complex queries in a readily understandable way, without needing to know how the information is stored in the underlying database. This article highlights the advantages of conceptual query languages such as ConQuer over traditional query languages for specifying queries and business rules.



[Conceptual Queries](#) (94K)

ConQuer: a Conceptual Query Language

This paper first appeared in Proc. ER'96: 15th International Conference on Conceptual Modeling, [Springer LNCS](#), no. 1157, pp. 121-33.

Relational query languages such as SQL and QBE are less than ideal for end user queries since they require users to work

explicitly with structures at the relational level, rather than at the conceptual level where they naturally communicate. ConQuer is a new conceptual query language that allows users to formulate queries naturally in terms of elementary relationships, and operators such as "and", "not" and "maybe", thus avoiding the need to deal explicitly with implementation details such as relational tables, null values, and outer joins. While most conceptual query languages are based on the entity relationship approach, ConQuer is based on Object Role Modeling (ORM), which exposes semantic domains as conceptual object types, thus allowing queries to be formulated in terms of paths through the information space. This paper provides an overview of the ConQuer language.



[ConQuer: a Conceptual Query Language](#) (150K)

Conceptual Queries using ConQuer–II

This paper first appeared in Proc. ER'97: 16th International Conference on Conceptual Modeling, [Springer LNCS](#), no. 1331, pp. 113-26.

Formulating non-trivial queries in relational languages such as SQL and QBE can prove daunting to end users. ConQuer is a conceptual query language that allows users to formulate queries naturally in terms of elementary relationships, operators such as "and", "or", "not" and "maybe", contextual for-clauses and object-correlation, thus avoiding the need to deal explicitly with implementation details such as relational tables, null values, outer joins, group-by clauses and correlated subqueries. While most conceptual query languages are based on the entity relationship approach, ConQuer is based on Object Role Modeling (ORM), which exposes semantic domains as conceptual object types, allowing queries to be formulated via paths through the information space. As a result of experience with the first implementation of ConQuer, the language has been substantially revised and extended to become ConQuer–II. ConQuer–II's new features such as arbitrary correlation and subtyping enable it to be used for a wide range of advanced conceptual queries.



[Conceptual Queries using ConQuer-II](#) (99K)

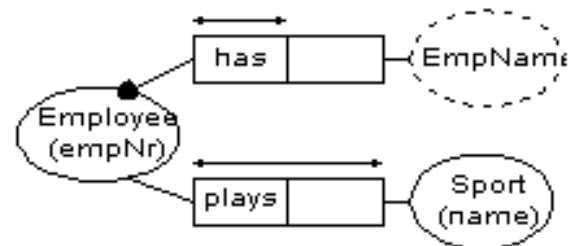
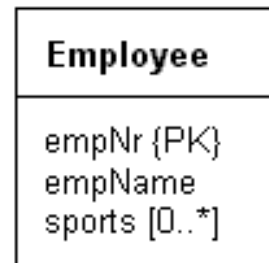
[ORM Home](#) [ORM in Detail](#) [Modeling Issues](#)
[Conceptual Queries](#) [UML and ORM](#) [Resources](#)

All diagrams on this site were created with Microsoft Visio.


[ORM in Detail](#)
[Modeling Issues](#)
[Conceptual Queries](#)
[UML and ORM](#)
[Resources](#)

UML AND ORM

In these articles, Dr. Halpin discusses the Unified Modeling Language within the context of Object Role Modeling (ORM) and shows how ORM models can be used in conjunction with UML models.



UML data models from an ORM perspective: Part 1

This article first appeared in the April 1998 issue of the [Journal of Conceptual Modeling](#).

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 10-part series provides a comparative overview of both approaches.

Part 1 provides some historical background on both approaches, identifies several design criteria for modeling languages, and discusses how object reference and single-valued attributes are modeled in both.



[UML data models from an ORM perspective: Part 1 \(66K\)](#)

UML data models from an ORM perspective: Part 2

This article first appeared in the May 1998 issue of the [Journal of Conceptual Modeling](#).

Second in a series of articles examining data modeling in UML from the perspective of ORM. This paper compares UML multi-valued attributes with ORM relationship types, including basic constraints on both. As part of this discussion, we also consider how these structures may be instantiated, using UML object diagrams or ORM fact tables.



[UML data models from an ORM perspective: Part 2](#) (50K)

UML data models from an ORM perspective: Part 3

This article first appeared in the June 1998 issue of the [Journal of Conceptual Modeling](#).

Third in a series of articles examining data modeling in UML from the perspective of ORM. This paper compares UML associations and related multiplicity constraints with ORM relationship types and related uniqueness, mandatory role and frequency constraints. It also contrasts instantiation of associations using UML object diagrams and ORM fact tables.



[UML data models from an ORM perspective: Part 3](#) (55K)

UML data models from an ORM perspective: Part 4

This article first appeared in the August 1998 issue of the [Journal of Conceptual Modeling](#).

Fourth in a series of articles examining data modeling in UML from the perspective of ORM, this paper examines associations in more detail, contrasting ORM nesting with UML association classes, and ORM co-referencing with UML qualified associations, then discusses exclusion constraints, and summarizes how the two methods compare with respect to terms and notations for data structures and instances.



[UML data models from an ORM perspective: Part 4](#) (53K)

UML data models from an ORM perspective: Part 5

This article first appeared in the October 1998 issue of the [Journal of Conceptual Modeling](#).

Fifth in a series of articles examining data modeling in the UML from the perspective of ORM, this paper discusses ORM subset and equality constraints, and how these may be specified in UML.



[UML data models from an ORM perspective: Part 5](#) (51K)

UML data models from an ORM perspective: Part 6

This article first appeared in the December 1998 issue of the [Journal of Conceptual Modeling](#).

Sixth in a series of articles examining data modeling in the UML from the perspective of ORM, this paper examines subtyping in ORM and in UML.



[UML data models from an ORM perspective: Part 6](#) (56K)

UML data models from an ORM perspective: Part 7

This article first appeared in the February 1999 issue of the [Journal of Conceptual Modeling](#).

Seventh in a series of articles examining data modeling in the UML from the perspective of ORM, this paper discusses some other graphic constraints (value, ring and join constraints.)



[UML data models from an ORM perspective: Part 7](#) (47K)

UML data models from an ORM perspective: Part 8

This article first appeared in the April 1999 issue of the [Journal of Conceptual Modeling](#).

Eighth in a series of articles examining data modeling in the UML from the perspective of ORM, this paper covers some recent updates to the UML standard, then discusses aggregation.



[UML data models from an ORM perspective: Part 8](#) (54K)

UML data models from an ORM perspective: Part 9

This article first appeared in the June 1999 issue of the [Journal of Conceptual Modeling](#).

Ninth in a series of articles examining data modeling in the UML from the perspective of ORM, this paper examines initial values and derived data in ORM and UML.



[UML data models from an ORM perspective: Part 9](#) (51K)

UML data models from an ORM perspective: Part 10

This article first appeared in the August 1999 issue of the [Journal of Conceptual Modeling](#).

Tenth in a series of articles examining data modeling in the UML from the perspective of ORM, this paper discusses changeability and collection types in UML and ORM.



[UML data models from an ORM perspective: Part 10](#) (55K)

A comparison of UML and ORM for data modeling

This paper appeared in Proc. EMMSAD'98 3rd IFIP WG8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, Pisa, Italy, in June, 1998.

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 can be remedied by using a fact-oriented approach for the conceptual modeling, from which UML class diagrams may be derived. This paper examines the relative strengths and weaknesses of UML and Object Role Modeling (ORM) for data modeling, and indicates how models in one notation can be translated into the other.



[A comparison of UML and ORM for data modeling](#) (138K)

Data modeling in UML and ORM revisited

This paper appeared in Proc. EMMSAD'99: 4th IFIP WG8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, Heidelberg, Germany in June, 1999.

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.



[Data modeling in UML and ORM revisited](#) (99K)

Data modeling in UML and ORM: a comparison

This paper appeared in the [Journal of Database Management](#), vol. 10, no. 4 (Oct-Dec, 1999), Idea Group Publishing, Hershey PA, USA.

This paper presents a detailed comparison of the conceptual data modeling capabilities of UML and ORM. It is based on the EMMSAD'98 conference paper listed above, but has been revised and extended for journal publication.



[Data modeling in UML and ORM: a comparison](#) (159K)

Augmenting UML with Fact-orientation

This paper first appeared in the workshop proceedings: UML: a critical evaluation and suggested future, HICCS-34 conference (Maui, January 2001), © 2000 IEEE.

This paper discusses various problems with UML (e.g. poor support for verbalization, weak constraint primitives, and multiplicity constraints that do not scale properly for n-aries) and shows how ORM can compensate for these deficiencies.



[Augmenting UML with Fact-orientation](#) (119K)

Evolving UML: Opportunities and Challenges

This [slide presentation](#) was included in the panel session "Research Issues for the Unified Modeling Language and Unified Process", at the IRMA-2002 Conference held in Seattle May 2002. It includes links to several proposals for UML 2.0, and notes some weaknesses of UML class diagrams in comparison with ORM.

[ORM Home](#) [ORM in Detail](#) [Modeling Issues](#)
[Conceptual Queries](#) [UML and ORM](#) [Resources](#)

All diagrams on this site were created with Microsoft Visio.

The logo consists of the letters "ORM" in a bold, red, sans-serif font, centered within a white oval that has a black border.[ORM in Detail](#)[Modeling Issues](#)[Conceptual Queries](#)[UML and ORM](#)[Resources](#)

RESOURCES

Here you'll find news about ORM tools and courses as well as links to other web resources featuring ORM-related material.

.NET Show on ORM

The [25th episode of the .NET Show](#) focused on ORM, including an interview with Terry Halpin, Pat Hallock and Dick Barden, and demonstrations of the ORM and database modeling features of Microsoft Visio for Enterprise Architects.

ORM tools

(1) Microsoft Visio Enterprise 2000 includes an ORM drawing stencil, as well as an ORM source model stencil that can be used to build database models and generate DDL code. Although functionally powerful, this ORM source model is capable of displaying only basic ORM constraints. Microsoft Visio Professional 2002 includes a basic ORM drawing stencil, but not the ORM modeling solution. For the Visio 2002 releases, Visio Enterprise was discontinued as a separate product. Instead it was significantly enhanced and renamed as Microsoft **Visio for Enterprise Architects** (VEA), and is available only as part of **Visual Studio .NET Enterprise Architect** (VSEA).

VSEA was released mid-January 2002 to MSDN Universal subscribers, by download from MSDN. VSEA is now available for general release in CD or DVD format (see [VS .NET pricing details at MSDN](#)). The VEA component of VSEA includes Visio Professional 2002 as well as enhanced versions of the database and software modeling solutions formerly in Visio Enterprise 2000: its database modeling solution provides deep support for ORM and logical/physical database modeling. Details on a COM API to the database modeling engine for this tool are accessible at websites maintained by John Miller (see below), and a free add-on that uses this API to expose data model details in the form of an XML document has been released by Scot Becker (see below). Microsoft has published the following [Features Overview for Visual Studio .NET Enterprise Architect](#) and the

following [Feature-by-feature comparison of the database and software modeling solutions in Visio Professional 2002 and Visual Studio .NET Enterprise Architect.](#)

In addition to the issues discussed in the ReadMe file for VEA, workarounds for some known bugs in the original VEA release are being published as Knowledge Base articles on Microsoft's technical support site. A draft version of those KB articles is accessible below.



[Knowledge base articles for database modeling solution in Microsoft Visio for Enterprise Architects](#) (146K)

Microsoft Visio for Enterprise Architects 2002 Service Release 1, which is now available as a free download. If you have installed the official first release of VEA, you can install this patch right over the top. The download is large (about 37 MB) since it also incorporates all the bug fixes and new features in the underlying Visio Professional 2002 SR1. As an alternative to download, it is expected that the VEA SR1 patch will also be available later on CD within MSDN upgrades.








Fixes for all the bugs mentioned in the draft version of the KB articles are included in the latest release of Microsoft Visio for Enterprise Architects, which shipped on 2003, April 24 as part of Microsoft Visual Studio .NET 2003 Enterprise Architect edition.

VEA is built on top of Visio Standard 2002 and Visio Professional 2002, for which a [Software Development Kit for Visio 2002](#) and a [Visio Viewer](#) are now available. The free Visio Viewer enables users without Visio installed to view (but not edit) your Visio diagrams.

The first eight of a series of articles on how to use the database modeling solution within Visio for Enterprise Architects were first published in the Journal of Conceptual Modeling (see InConcept entry below). Here are slightly revised versions of these articles:



[Microsoft's new database modeling tool: Part 1](#) (598K)

-  [Microsoft's new database modeling tool: Part 2](#) (447K)
-  [Microsoft's new database modeling tool: Part 3](#) (231K)
-  [Microsoft's new database modeling tool: Part 4](#) (252K)
-  [Microsoft's new database modeling tool: Part 5](#) (255K)
-  [Microsoft's new database modeling tool: Part 6](#) (228K)
-  [Microsoft's new database modeling tool: Part 7](#) (290K)
-  [Microsoft's new database modeling tool: Part 8](#) (505K)

Microsoft has also published revised versions of five of these articles on its MSDN website: [Visio-Based Database Modeling in Visual Studio .NET Enterprise Architect: Part 1](#); [Part 2](#); [Part 3](#); [Part 4](#); [Part 5](#).

Visio for Enterprise Architects is built on top of Visio Professional 2002, which itself is built on top of Visio Standard 2002. A [Software Development Kit for Visio 2002](#) to assist users to customize their own Visio solutions is now available for download.

A free [Visio Viewer](#) to enable users who have not purchased Visio to view Visio files is now available for download.

(2) The former ORM tool known as **VisioModeler** is now freely available as an unsupported product from Microsoft Corporation (as a 25 MB download). Models developed in VisioModeler may be exported to Microsoft's current and future ORM solutions. To obtain the free VisioModeler download, go to <http://download.microsoft.com>, search by selecting Keyword Search, enter the keyword "VisioModeler", select your operating system (e.g. Windows XP, Windows 2000, Windows 95,

Windows 98 or NT 4.0), change the setting for "Show Results for" to "All Downloads", and hit the "Find It!" button. This should bring up a download page that includes the title "VisioModeler (Unsupported Product Edition)". Clicking on this will take you to the link for the download file MSVM31.exe. Click on this to do the download. The 25 MB file takes just over 2 hours to download on a 28.8 k modem. Here is the general [download page](#). VisioModeler includes an online manual to explain its use. In addition, you may download this basic [tutorial on how to use VisioModeler](#) (381k).

(3) A modeling tool caled **CaseTalk** based on the ORM-dialect known as Fully Communication Oriented Information Modeling (FCO-IM) is available from Bommeljé Crompvoets en partners b.v., headquartered in Utrecht, The Netherlands. To find out more about this tool, click this [CaseTalk news page](#).

Books

(1) For an in-depth treatment of ORM, see Halpin, T.A. 2001, *Information Modeling and Relational Databases*, published by [Morgan Kaufmann Publishers](#) (ISBN 1-55860-672-6). Details on this book are available at the [book's website](#), which includes a link to a Companion Website that includes additional appendices, answers to odd numbered questions etc. for download. The book can be ordered online at the publisher's website (above) or at various other sites, for example at [Barnes&Noble](#) or at [Amazon](#).

The April 2001 issue of the Journal of Conceptual Modeling ran an article of mine that overviewed the contents of the above book, including an excerpt from the data warehousing section. Here is a slightly revised version of this article:



[Book overview, and data warehousing](#) (253K)

The first and second printings of the book included a number of errors, as detailed in the [Book Errata](#).

(2) The following book was published on 2003 August 28:

Halpin, T., Evans, K., Hallock, P. & MacLean B. 2003, *Database Modeling with Microsoft Visio for Enterprise Architects*, Morgan Kaufmann Publishers: San Francisco, ISBN 1-55860-919-9.

This is the first book to provide a detailed, authoritative coverage of how to use Microsoft's high end Visio tool to design databases. The book may be ordered online from various booksellers, including [this Amazon website](#) and [this Barnes & Noble website](#).

The first printing of the book included a number of errors, as detailed in the [Book Errata](#).

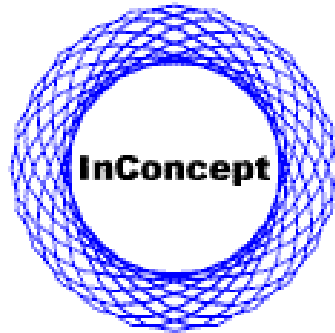
(3) For a discussion of research topics on UML, see Siau, K. & Halpin, T.A. (eds), *UML: Systems Analysis, Design and Development Issues* published by Idea Group Publishing. This book includes a chapter providing an in-depth comparison of ORM and UML. Details on this book are available at the [book's website](#).

ORM courses

Microsoft recently added Course 2090 to its official curriculum. This 3-day, instructor-led course is titled "Modeling Business Requirements to Create a Database Using Microsoft Visual Studio .NET Enterprise Architect". It focuses on the use of ORM and Visio for Enterprise Architects to perform database modeling -- see [Course 2090 details](#). Instruction for this course is available from various qualified ORM instructors and Microsoft Certified Trainers.

Before its acquisition by Microsoft Corporation, Visio certified a number of consulting partners to offer training courses in ORM. One of these partners, InConcept Inc., offers [5-day ORM course](#) at Alto Consulting & Training in Minneapolis.

InConcept, Inc.



[InConcept, Inc.](#) is a database consulting firm dedicated to excellence in data modeling. Emphasis is placed on the conceptual model using Object Role Modeling (ORM). This higher level design is more suitable for review with customers while the logical and physical models, derived from the conceptual model, are more suited to the technical staff.

Modeling a database at the conceptual level significantly reduces design errors, thus reducing overall cost. Using ORM enables the designer and the business user to communicate and capture business rules more readily and easily.

The [Journal of Conceptual Modeling](#) is a free journal produced by InConcept and dedicated to data modeling, design, and implementation issues. The goal of this publication is to promote communication between professionals, share knowledge, and to educate our readers. The target audience is large: database professionals and developers, end users and business professionals, students and teachers, and anyone else using, developing, or considering development of a database system.

Business Rules Community

The [Business Rules Community](#) is an online vertical community for business rules professionals. Membership is free, and includes access to the *Business Rules Journal*, which includes regular columns by renowned experts in the business rules movement, as well as feature articles by leading industry professionals.

This journal now includes a regular column by Terry Halpin. The initial series of articles in this column focuses on verbalization of business rules. Some weeks after the publication of one of these articles on the business rules community website, a pdf version of the article is typically made available below. If available, you

may download the pdf version. Otherwise, click the Business Rules Community (BRC) website link to view it there.

[Modeling Concepts: Setting the Scene](#)



[Verbalizing Business Rules \(part 1\): PDF file \(304K\); BRC link.](#)



[Verbalizing Business Rules \(part 2\): PDF file \(304K\); BRC link.](#)



[Verbalizing Business Rules \(part 3\): PDF file \(301K\); BRC link.](#)



[Verbalizing Business Rules \(part 4\): PDF file \(316K\); BRC link.](#)



[Verbalizing Business Rules \(part 5\): PDF file \(286K\); BRC link.](#)



[Verbalizing Business Rules \(part 6\): PDF file \(269K\); BRC link.](#)



[Verbalizing Business Rules \(part 7\): PDF file \(279K\); BRC link.](#)



[Verbalizing Business Rules \(part 8\): PDF file \(330K\); BRC link.](#)

Professor Robert Meersman

Professor Robert Meersman is one of the original ORM pioneers,

introducing subtyping to the methodology when it was first developed in the Control Data research institute at the Free University of Brussels (VUB). He has been an active researcher in information system semantics and conceptual query technology ever since, and is currently exploring the use of ORM as an ontological basis for the semantic web. He is currently a professor in the department of computer science at the Free University of Brussels, and is the director of its STARlab research laboratory. His [home page](#) includes teaching and research information.

Dr. Arthur ter Hofstede

Dr. Arthur ter Hofstede, a prominent ORM researcher, is an Associate Professor and Leader of the Cooperative Information Systems Special Interest Group within the Faculty of Information Technology at the Queensland University of Technology in Brisbane, Australia. His [home page](#) includes teaching and research information, as well as an extensive list of publications, most of which address data modeling issues.

Dr. Erik Proper

[Dr. H. A. \(Erik\) Proper](#), is a lecturer within the informatics subfaculty at the University of Nijmegen, The Netherlands. His theoretical and industrial research covers many information systems topics, including schema evolution, schema optimization and conceptual query technology. His website includes an extensive list of downloadable research publications, many of them directly related to ORM.

John Miller

[John Miller](#) is the principal of Perpetual Data Systems, a consultancy based in California. John maintains "Wikis" with details about the unsupported COM API to the database modeling solution in Microsoft Visio for Enterprise Architects. Here is his [ORM Wiki](#), and here is his [Viso Modeling Engine Wiki](#).

Scot Becker

[Scot Becker](#) is the principal of Orthogonal Software, a

consultancy based in Minneapolis. Scot has released [Orthogonal Toolbox](#), a free add-on to Visio for Enterprise Architects that exposes most of the model details stored in an ORM source model or a logical database model as an XML document. This information is extracted using the COM API to the modeling engine mentioned above.

Scot has now added an informative blog site [ObjectRoleModeling.com](#) that includes lots of useful tips and news about ORM and related database modeling topics.

Ken North

[Ken North](#) is a database practitioner and author of several publications on databases, including the following articles that discuss ORM: '[Modeling, metadata and XML](#)', '[Modeling, data semantics and natural language](#)' and '[Database design for prime time](#)'.

Conferences

The [7th International Business Rules Forum](#) will be held in Las Vegas on November 7-11, 2004. The conference program includes many sessions by leading practitioners on state-of-the-art approaches to business rules. The Monday Nov 8 program includes a half-day tutorial by Terry Halpin on Verbalizing, Visualizing, and Validating Business Rules.

Recent conferences:

The [Data Management Association](#) is an international body of professionals dedicated to improving the management of data. The [Sixteenth DAMA International Symposium and 8th Meta-Data Conference](#) was held in Los Angeles, California, May 2 - 6, 2004. The program included over 100 speakers from all over the world. Terry Halpin presented in a paper session, a night school, and a panel session.

The [Sixteenth International Conference on Advanced](#)

[Information Systems Engineering](#) was held in Riga, Latvia, June 7 - 11, 2004. In conjunction with this conference, the ninth IFIP WG 8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design was held.

The [European Business Rules Conference](#) was held in Amsterdam, The Netherlands, June 16 - 18, 2004. This conference included a half day tutorial by Terry Halpin on Business Rules and ORM.

The [6th International Business Rules Forum](#) was held in Nashville, TN on November 2-6, 2003. The conference program included many sessions by leading practitioners on state-of-the-art approaches to business rules.

The Entity Relationship Conference series addresses all forms of conceptual modeling (ER, ORM, UML etc.). The [ER-2003 conference](#) was held in Chicago, USA on October 13-16, 2003. The program included both academic and industrial presentations, including two papers on ORM.

The eighth IFIP WG 8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design ([EMMSAD'03](#)) was held in Velden, Austria on June 16-17, 2003, in conjunction with [CAiSE'03](#), the 15th Conference on Advanced Information Systems Engineering. The EMMSAD workshop included two papers related to ORM.

The [Fifteenth DAMA International Symposium and 7th Meta-Data Conference](#) was held in Orlando, Florida, April 27 - May 1, 2003. The program included 130 speakers from all over the world. Terry Halpin presented a session on [Metamodels for ER, ORM and UML: a Critical Review](#), participated in the [Data Modeling Panel: Have Things Really Changed?](#), and was awarded the DAMA International Achievement Award for Education.

The IRMA-2002 Conference (Seattle, 2002 May 19-22) included a panel session on research issues in UML. For a copy of the slides presented by Terry Halpin at this panel, see the UML and ORM section.

The [International DAMA 2002 Symposium](#), held in San Antonio, Texas, April 29 - May 2, 2002, included three presentations related to ORM. Terry Halpin presented an Introduction to ORM, and participated in a panel discussion on data modeling approaches, while Chandrika Shankaranayan provided an overview of the modeling tools in Visual Studio .NET Enterprise Architect.

[Microsoft Tech Ed 2002](#), held April 9-13 in New Orleans, included hundreds of in-depth technical sessions on Microsoft Visual Studio .NET, XML web services, the .NET platform, building secure applications, and other Microsoft technologies and related products. The Visio-based modeling solutions in Visual Studio .NET Enterprise Architect featured in two sessions: "Conceptual Database Design in Visual Studio .NET" (presented by Terry Halpin) and "UML modeling in Visual Studio .NET Enterprise Architect" (presented by Lance Delano).

Microsoft's [Professional Developer Conference 2001 \(PDC 2001\)](#) held in Los Angeles, October 22-26, 2001, included a session co-presented by Terry Halpin and John Miller on the database modeling solution within Visio for Enterprise Architects (included in Visual Studio .NET Enterprise Architect).

[ORM Home](#) [ORM in Detail](#) [Modeling Issues](#)
[Conceptual Queries](#) [UML and ORM](#) [Resources](#)

All diagrams on this site were created with Microsoft Visio

Business Rules and Object Role Modeling

Database Programming & Design, October 1996, reprinted with permission.

*To capture
fast-paced,
complex businesses,
data modelers
must consider
methods that
go beyond
traditional ER
diagramming*

To capture fast-paced, complex businesses, data modelers must consider methods that go beyond traditional ER diagramming.

In spite of remarkable progress in computing technology, many businesses are still struggling with the problem of modeling and accessing data. Although faster hardware and graphical interfaces do help somewhat, they do not address the problem's fundamental cause. A business is basically a complex, evolving "organism", about which we need to communicate efficiently. So our language for modeling and querying must be clear yet detailed enough to capture the business complexity and remain easy to change as the business evolves.

Happily, such a linguistic framework already exists. It's called Object Role Modeling (ORM), and we'll look at some of the key features that distinguish ORM from entity relationship (ER) and object oriented (OO) approaches.

WHAT IS ORM?

ORM is a method for designing and querying database models at the conceptual level, where the application is described in terms readily understood by users, rather than being recast in terms of implementation data structures. This high-level approach is philosophically in tune with the business rules movement evangelized by such industry leaders as Barbara von Halle and Ron Ross.

Typically, a modeler develops an information model by interacting with others who are collectively familiar with

the application. Because these subject matter experts need not have technical modeling skills, reliable communication occurs by discussing the application at a conceptual level, using natural language, analyzing the information in simple units, and working with instances (sample populations).

ORM is specifically designed to improve this kind of communication. It comes in a variety of flavors, including natural language information analysis method (NIAM), which is best known in Europe, where the method originated in the mid-1970s. Since then, ORM has been extended and refined by researchers in Australia, Europe, the U.S., and elsewhere.

Unlike ER, which has dozens of different dialects, ORM has only a few dialects with only minor differences.

Object Role Modeling got its name because it views the application world as a set of objects (entities or values) that plays roles (parts in relationships). We sometimes call it fact-based modeling because ORM verbalizes the relevant data as elementary facts. These facts can't be split into smaller facts without losing information.

Suppose Table 1 includes data about athletes competing in the recent Olympic Games. For simplicity, assume the athletes are identified by their names. The first row contains two elementary facts: the Athlete named "Ann Arbor" represented the Country coded "USA," and the Athlete named "Ann Arbor" was born in the Country coded "USA". The null value "?" indicates the absence of a fact to record Bill Abbot's birthplace. All conceptual facts are ele-

mentary rather than compound, so null values do not feature in verbalization.

Although Table 1 includes five fact instances, it has only two fact types: Athlete represents Country; Athlete was born in Country. Refer to Figure 1 to see how this table is modeled in ORM. Two object types, Athlete and Country, are shown as named ellipses with their reference schemes in parenthesis: Athletes are identified by their names, and countries are identified by codes (for example, "USA").

A role is a part played by an object in a relationship and is shown as a box connected to its object type. In the relationship, Athlete represents Country, Athlete plays the role of representing, and Country plays the role of being represented.

You can permit the same fact to be read in different directions (for example, "Country is birthplace of Athlete" is just the reverse reading of "Athlete was born in Country"). ORM allows relationships with one role (for example, Athlete runs), two roles, three roles, or as many roles as you like. Because facts are elementary, the number of roles rarely exceeds four.

Each role may be associated with a column of the associated fact table. Figure 1 includes fact tables for both fact types. Although sample populations are very useful for checking and understanding constraints, they are not part of the conceptual schema itself.

The black dot is a mandatory role constraint (each Athlete represents a Country). The arrow-tipped bars are uniqueness constraints (for example, each Athlete represents at most one Country).

NO ATTRIBUTES

Unlike ER modeling, ORM does not use attributes. In ER, you might model two fact types by saying the entity type Athlete has the attributes "country-Represented" and "birthplace," both of which are based on the domain Country. If you're used to ER, you might think this approach is a better way of doing things. But it is not. Let's see why.

The first problem with using attributes in the initial model is that they are often unstable. Suppose we decide to add the fact type: Country has Population. This addition would now force us to show Country as an entity type, so we would have to replace our attribute portrayal by relationship types.

In ORM, all we have to do is add the new fact type; nothing else changes, and we have gained the added benefit of revealing the conceptual object types (semantic domains) that bind the schema together. One major benefit is that con-

Athlete	Country	Birthplace
Ann Arbor	USA	USA
Bill Abbot	UK	?
Chris Lee	USA	NZ

TABLE 1. Some data about athletes.

ceptual queries may now be formulated in terms of continuous paths through the schema. Moving from a role through an object type to another role amounts to a conceptual join. ER diagrams typically omit domains, so you must look them up in a table.

Another problem with attributes is that they make it awkward to talk about fact populations. ER diagrams are simply too cumbersome for performing the population checks that are so vital for validating rules with clients.

Displaying some facts as attributes and some as relationships leads to the requirement for different notations to express the same kind of constraint or rule. Apart from this unnecessary complexity, some ER notations don't let you express a constraint on an attribute, even if that constraint could be expressed with the fact modeled as a relationship.

So don't agonize over whether to model a particular feature as an attribute or relationship. Just model it as a relationship. Does this mean you should never use attributes? Not quite. When designing or transforming a model, you should avoid attributes. In other words, you should delay making a commitment on which features are less important than others. However, once you have the full model, it is possible to determine relative importance; displaying less important features as attributes can help provide a compact view of the model.

ORM includes abstraction techniques so that you can display "minor" fact types as attributes. In fact, the best way to obtain an ER diagram is by abstracting it from an ORM schema

"MIXFIX N-ARIES"

A relationship with one role (for example, runs, smokes) is unary. The relationships we saw in Figure 1 were binary (two roles) with the verb phrase written

in "infix" position (between the objects). Now look at Figure 2. The diagram shows the ternary (three roles) fact type: Room at Time is used for Activity. In ORM, as in logic, a predicate is just a sentence with object-holes in it. Each object hole is shown as an ellipsis ("...").

To allow natural expression in English as well as cater to other languages (such as Japanese, where verbs usually come at the end rather than in the middle), ORM allows "mixfix" predicates (that is, the object holes can be mixed into the predicate at any position). If you fill each hole with an object term, you get a sentence that is a fact instance. For example, Room "23" at Time "Mon 9am" is used for Activity "IM class."

If a predicate is postfix unary (placed after the object) or infix binary (inserted between two objects), then the object positions are known. In those cases, predicates may omit the ellipses indicating object holes.

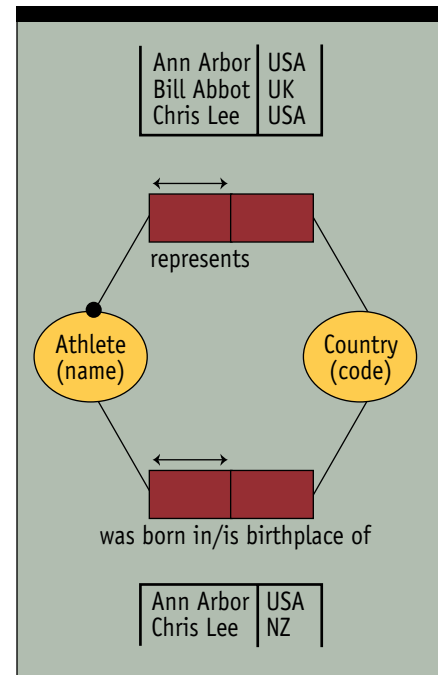


Figure 1. Populated ORM diagram.

Figure 2 includes a sample population for the ternary fact type. If this population is significant, then two uniqueness constraints (as shown in the ORM diagram) exist. The left-most constraint says that the same Room at the same time is used for at most one activity: This statement is probably correct. The right-most constraint says that at most one room is used at the same time by any given activity. This statement is possibly correct.

For checking, you must carefully test the constraint verbalization. To double

check, discuss counterexamples (extra rows that would violate the constraint). For example, to test the right-most constraint, you could add the row: (“50, Mon 9am, TB demo”). The population would then indicate that on Monday at 9 a.m. the Toolbook demonstration uses both rooms 45 and 50. Is this kind of thing possible? Testing fact instances makes it easier for the domain expert to confirm it one way or the other.

Notice how the ternary formulation simplifies modeling and checking. With typical ER and OO tools, you must make the fact type binary by using artificial entity types (for example, Room-Time or Time-Activity), which makes it extremely awkward to populate and perhaps impossible to express all the constraints. For example, using your favorite ER notation, how would you capture the Time-Activity uniqueness constraint?

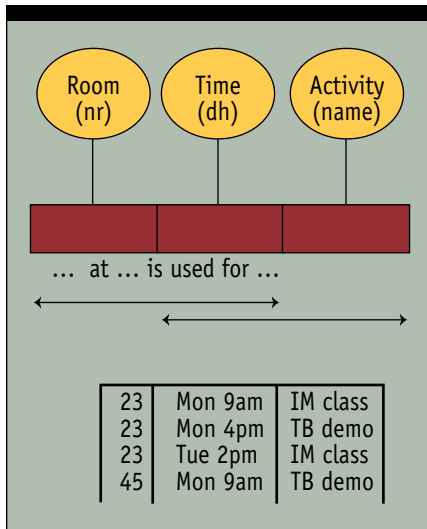


Figure 2. A ternary fact type.

Making all the facts binary is an unwanted burden. Why should you have to break ternary rules into two stages and worry about which pair to take first? ORM lets you model such things naturally without being restricted by infix binary straightjackets.

EXPRESSIVE RULE NOTATION

ORM has a rich language for expressing business rules, either graphically or textually. Consider Figure 3, which shows an ORM schema. A verbal version of the constraints would begin with three simple (n:1) uniqueness constraints: one compound (m:n) uniqueness constraint and two mandatory role constraints, as follows:

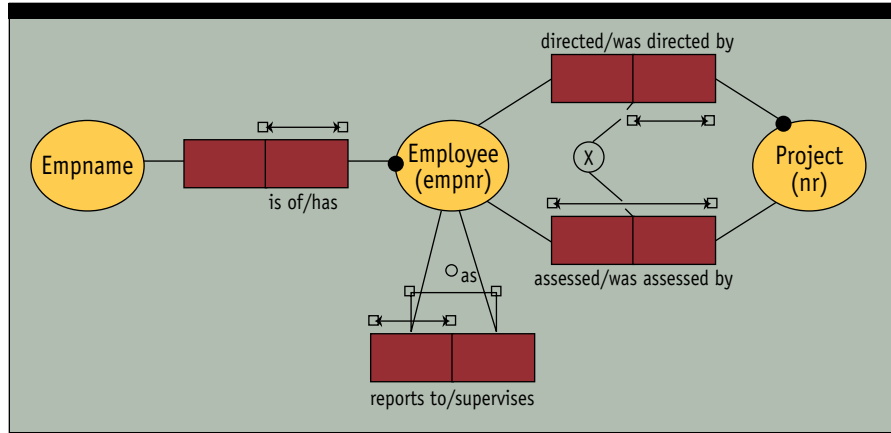


Figure 3. Graphical rule notation in ORM.

Each Employee has at most one Empname.
 Each Project was directed by at most one Employee.
 Each Employee reports to at most one Employee.
 It is possible that some Employee assessed more than one Project and that some Project was assessed by more than one Employee.
 Each Employee has some Empname.
 Each Project was directed by some Employee.
 No Employee directed and assessed the same Project.
 If Employee e1 reports to Employee e2, then it cannot be that Employee e2 reports to Employee e1.

The circled “X” in Figure 3 is a pair-exclusion constraint: No Employee-Project pair may occur in both the director and assessment predicates. This fact is verbalized as “No Employee directed and assessed the same Project.” For example, the constraint is violated if we populate the fact types with: “e1 directed p1”; “e1 assessed p1.”

Finally, the ring symbol with “as” is an asymmetric constraint. You can’t report to yourself or to someone else who reports to you, which is verbalized as: “If Employee e1 reports to Employee e2, then it cannot be that Employee e2 reports to Employee e1.” Because e1 and e2 are not necessarily distinct, this includes the irreflexive case (you can’t report to yourself).

The fact type “Employee reports to Employee” is a ring relationship, in which both roles are played by the same object type. ORM includes other ring constraints such as intransitivity and acyclicity.

Figure 4 illustrates a few more ORM constraints. Each employee is either on contract or tenured but not both, as shown by the black dot connecting the

two relevant roles and the exclusion constraint between them.

The circled “u” is an external uniqueness constraint, indicating that Empname-Dept combinations are unique (that is, within the same department, employees have distinct names). The dotted arrow is a pair-subset constraint: Each manager who heads a department also works for the same department.

The thick arrow indicates that Manager is a subtype of Employee. In ORM, subtypes should be well defined (for example, each Manager is an Employee who has Rank “mgr”). ORM also supports multiple inheritance. For example, we might introduce another subtype ContractEmployee, and then ContractManager, which is a subtype of both ContractEmployee and Manager.

ORM conceptual schemas basically comprise fact types, constraints, and de-

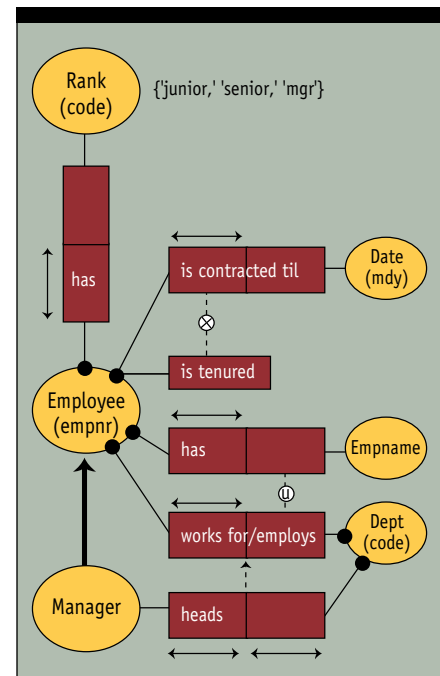


Figure 4. Further constraint examples.

Derivation rules. Derivation rules may be arithmetic or logical. For example, the fact type “Dept has NrStaff” may be derived by counting instances of “Employee works for Dept.” The fact type “Manager manages Employee” may be derived from the path “Manager heads a Dept that employs Employee.”

By now you might be getting the feeling that ORM is too complicated. Actually, it's not. This stuff is taught to school kids back in my home state, and I've already shown you most of the graphic symbols. Because we express all facts in the same way, using roles, the notation is both uniform and simple to populate. So it's easy to illustrate a lot of business rules that actually apply to your business.

SCHEMA TRANSFORMS

Although the fact-based approach gives greater schema stability, it is still possible to describe the same feature in different ways. For implementation, ORM schemas are usually mapped to relational database schemas, in which many fact types may be grouped into a single table. Different but equivalent ORM schemas may map to different target schemas, which differ in efficiency.

Semantic optimization may often be performed before the mapping takes place. ORM includes a vast array of schema transformations as well as optimization heuristics to determine which transformations to use. For a trivial example, see the ORM schema in Figure 5. This schema deals with teams that are mixed doubles (one of each sex). The “2” is a frequency constraint: If a team has any players recorded, then it must have two players recorded.

By default, Figure 6 maps to two relational tables, one for Player and one for Team. For optimization, the original conceptual schema may be transformed into Figure 5 before mapping. Here the Sex object type has been absorbed into the team membership predicate, specializing it into two predicates, one for each sex.

This new schema maps to only one table (Team). If no other facts are recorded about Players, this new schema is more efficient, because queries and updates involve just one table, with no need for a join or referential integrity check.

Note the importance of a rich constraint language. To ensure that the schemas in Figures 5 and 6 actually are equivalent, we must be able to transform any constraints in one to constraints in

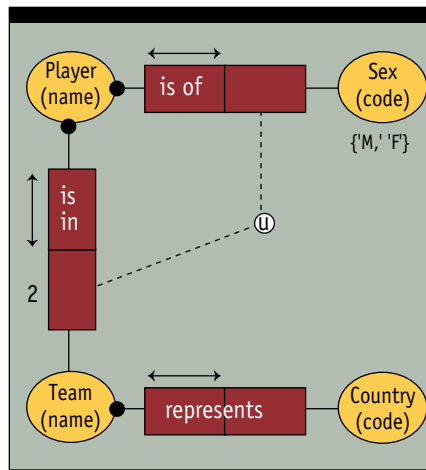


Figure 5. One way to model.

the other. For example, the frequency constraint “2” is transformed into an equality constraint (shown as a dotted line with arrows at both ends) that says a team has a male player if and only if it has a female player.

The uniqueness constraint that each player is of at most one sex is transformed to an exclusion constraint between the two roles of Player. The external uniqueness constraint (for each sex and team there is at most one player) reappears as two simple uniqueness constraints on the first roles of the has-male and has-female predicates.

ORM's expressive rule language and rigorous transformation theory provide a powerful, controlled means to reshape and semantically optimize data models.

DESIGN METHOD

Like any good modeling method, ORM is far more than a notation. It includes various design procedures to help modelers develop and evolve their conceptual models.

For analysis and design, we divide large applications into appropriately

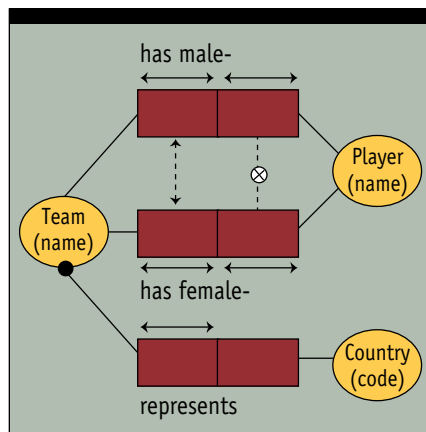


Figure 6. An equivalent model.

sized modules and then model them using the conceptual schema design procedure (CSDP). Finally, the various subschemas obtained in this way are merged into a global schema. The CSDP itself has seven main steps:

1. Transform familiar examples into elementary facts, and apply quality checks.
2. Draw the fact type, and apply a population check.
3. Check for entity types that should be combined, and note any arithmetic derivations.
4. Add uniqueness constraints, and check arity (number of roles) of fact types.
5. Add mandatory role constraints, and check for logical derivations.
6. Add value, set comparison, and subtyping constraints.
7. Add other constraints, and perform final checks.

You can find full explanations of this procedure in the reference section. The key to the CSDP's success is that it begins by verbalizing familiar information

EmployeeNr:	203101
Empname:	Terry O'Farrell
Supervisor:	
Projects Directed:	51, 65, 73, 84
Projects assessed:	70, 76

TABLE 2. An employee form.

examples in terms of simple facts. No matter how information is presented (tables, forms, graphs, and so on), it is always possible to conceptualize it in this way.

For example, a set of employee forms like that shown in Table 2 could have been used as input to the verbalization that eventually resulted in the schema shown earlier in Figure 3.

RELATIONAL MAPPING

ORM includes procedures for mapping and reverse engineering between conceptual models and logical models. By “logical models,” I mean implementation data models such as relational, network, hierarchic, nested relational, and various object-oriented models.

With an ORM tool, ORM models can be automatically mapped to database schemas for implementation on most

popular relational DBMSs. For example, the ORM schema in Figure 3 maps to a relational schema that can be specified in SQL-92. For simplicity, referential actions are omitted, and the exclusion and asymmetry constraints are shown as assertions. Depending on the target system, these assertions might be coded as the following insert triggers or stored procedures:

```

create table Employee(
  empnr smallint not null
    primary key,
  empname varchar(20) not null,
  supervisor smallint
    references Employee)

create table Project(
  projectnr smallint not null
    primary key,
  director smallint not null
    references Employee)

create table Assessment(
  empnr smallint not null
    references Employee,
  projectnr smallint not null
    references Project,
  primary key( empnr, projectnr ))

create assertion "Nobody directed
and assessed the same project"
check( not exists( select *
  from Project X, Assessment Y
  where X.director = Y.empnr
  and X.projectnr = Y.projectnr ))

create assertion "Reporting is
asymmetric"
check( not exists( select *
  from Employee X, Employee Y
  where X.empnr = Y.supervisor
  and X.supervisor = Y.empnr ))

```

OBJECT ORIENTATION

A lot of people have been discussing so-called object oriented approaches to information systems modeling. Although object oriented programming has advantages over traditional programming, OO techniques do not provide the best basis for information modeling.

ProjectNr:	51
ProjectTitle:	Nuclear Fusion
Director:	203101
Assessors:	105123
	107200

TABLE 3. A project form.

OO modeling includes a mixture of conceptual, external, and internal concepts. Some OO concepts, such as subtyping, belong to the conceptual level. Some other aspects, such as hidden object identifiers, are not conceptual because they are not part of human communications in the application world.

OO models, as well as ER and relational models, complicate things by grouping facts into attribute structures (for example, "objects" and tables). When validating facts with clients, it is preferable to deal with one fact at a time. A base ORM schema provides the simplest way of validating facts.

Suppose our Employee-Project application is intended to handle forms or reports like those in Tables 2 and 3.

Some modelers see forms like this and immediately want to model the information in the way the form is structured. This perspective leads to an OO approach. For example, the application might be modeled as Employee and Project objects (Figure 7):

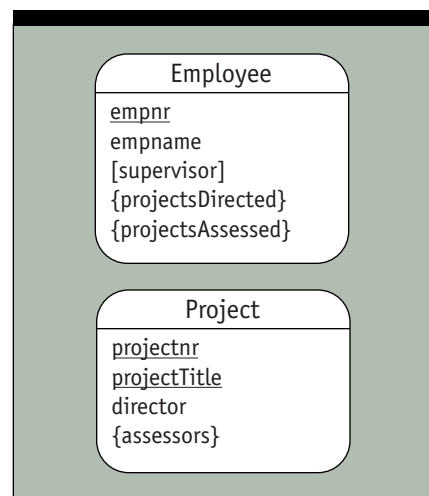


Figure 7. Employee and Project objects.

Here unique attributes are underlined, optional attributes are enclosed in square brackets, and set-valued attributes are enclosed in curly brackets.

This schema is further away from natural verbalization and does not facilitate sample populations (consider checking the uniqueness constraints). Moreover, the director fact type is represented twice, once as the set-valued {projectsDirected} attribute of Employee and again as the director attribute of Project. The same is true of the assessment fact type.

Although this fact type redundancy may be acceptable as a way to implement the model—for example, in an OO database we might do it this way, with forward

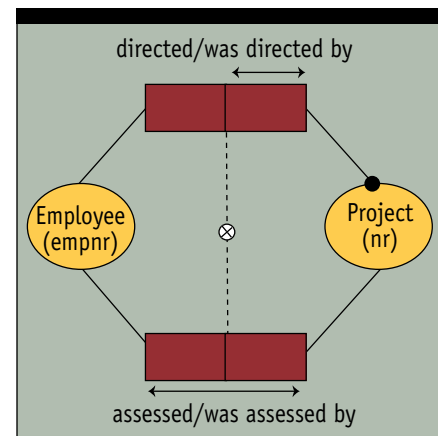


Figure 8. A model fragment.

ward pointers kept synchronized with the inverse pointers—this portrayal is clearly not conceptual.

The same application may be modeled in ORM as in Figure 3, if we add the fact type: "Project has ProjectTitle." For discussion purposes, part of the model is reproduced in Figure 8.

Note that the exclusion constraint is missing from the OO model. Such constraints are not supported directly and must be coded up separately. Even if the exclusion constraint were added, where would we put it?

The OO philosophy is to wrap constraints up inside objects. We could embed it in just the Employee or the Project object; however, at least conceptually, we would forget about it when viewing the other object.

We could embed it in both objects and take care to synchronize this constraint redundancy. This approach is quite nasty because the constraint must be treated differently in the two objects. In Employee, the constraint is enforced by ensuring the intersection of projectsDirected and projectsAssessed is empty. In Project, it is enforced by ensuring director is not a member of assessors. What has this got to do with conceptualizing the application?

Finally, we could fudge by creating another superobject in which to embed the constraint, but this is even more of an implementation issue. Modeling an application is hard enough even at the conceptual level. We certainly don't want to complicate this task by simultaneously worrying about implementation details.

The solution is using ORM first to do the conceptual model, getting all the benefits of its simplicity, populatability, and richness, and then using it to apply mapping procedures to generate other views (such as ER, RM, and OO).

If you're still not convinced, consider

the problem of schema evolution. For example, we might have originally designed our application to have only one assessor for each project. In ORM, the only change is the uniqueness constraint on the assessment fact type (see Figure 9).

If mapped to a relational schema, the change is more dramatic. For example, the separate Assessment table is eliminated in favor of an assessor column in the Project table, and the exclusion constraint is coded as the clause: "check(assessor <> director)."

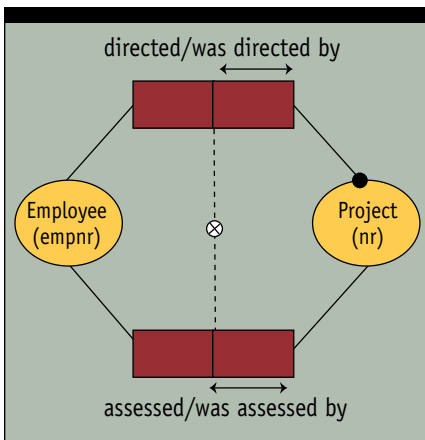


Figure 9. A minor change

In an OO schema, the {assessors} attribute is replaced by a simple assessor attribute, and the exclusion constraint in the Project object must be coded as an inequality instead of nonmembership. Apart from the constraint change, access to assessment facts is now quite different.

CONCEPTUAL QUERIES

Apart from conceptual modeling, ORM is ideal for performing queries at the conceptual level. Using an ORM query tool, you can query a database without any knowledge of how the facts are grouped into implementation structures.

Suppose you want to list the titles of those projects that have an assessor. This request may be formulated as the following ORM query: "List the ProjectTitle of each Project that was assessed by an Employee."

If a project has at most one assessor (as shown in Figure 9), this query generates the following SQL:

```
select projectTitle from Project
where assessor is not null
```

Suppose the application evolves to allow more than one assessor per project (as shown in Figure 8). You do not need to change the ORM query, because constraints have nothing to do with the meaning of our query. Underneath the covers, however, the relational structures have changed and the following SQL query is generated:

```
select X1.projectTitle
from Project X1, Assessment X2
where X1.projectnr = X2.projectnr
```

We can easily formulate more substantial queries as conditioned paths through ORM space. To sum up, ORM simplifies modeling and query formulation and minimizes the impact of schema evolution. With the development of ORM tools, the beginning of the semantic revolution has at last arrived.

REFERENCES

"Black Belt Design," *DBMS*, 8(10), September 1995.

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

Halpin, T.A. "Object-Role Modeling: An Overview."

Terry Halpin, Ph.D, was the head of research for the Database Division at Asymetrix Corp. and a senior lecturer in computer science at the University of Queensland at the time of this writing. He is currently Director of Database Strategy at Visio Corporation .

Object-Role Modeling: an overview

Terry Halpin
Microsoft Corporation

This paper provides an overview of Object-Role Modeling (ORM), a fact-oriented method for performing information analysis at the conceptual level. The version of ORM discussed here is supported in Microsoft Visio for Enterprise Architects, part of Visual Studio .NET Enterprise Architect.

Introduction

It is well recognized that the quality of a database application depends critically on its design. To help ensure correctness, clarity, adaptability and productivity, information systems are best specified first at the conceptual level, using concepts and language that people can readily understand. The conceptual design may include data, process and behavioral perspectives, and the actual DBMS used to implement the design might be based on one of many logical data models (relational, hierarchic, network, object-oriented etc.). This overview focuses on the data perspective, and assumes the design is to be implemented in a relational database system.

Designing a database involves building a formal model of the application area or universe of discourse (UoD). To do this properly requires a good understanding of the UoD and a means of specifying this understanding in a clear, unambiguous way. *Object-Role Modeling* (ORM) simplifies the design process by using natural language, as well as intuitive diagrams which can be populated with examples, and by examining the information in terms of simple or *elementary facts*. By expressing the model in terms of natural concepts, like *objects* and *roles*, it provides a *conceptual* approach to modeling.

Early versions of object-role modeling were developed in Europe in the mid-1970s (e.g. binary relationship modeling and NIAM). The version discussed here is based on the author's formalization of the method, and incorporates extensions and refinements arising from research conducted in Australia and the USA. The associated language FORML (Formal Object-Role Modeling Language) is supported in Microsoft *Visio for Enterprise Architects (VEA)*, part of Visual Studio .NET Enterprise Architect.

Another conceptual approach is provided by Entity-Relationship (ER) modeling. Although ER models can be of use once the design process is finished, they are less suitable for formulating, transforming or evolving a design. ER diagrams are further removed from natural language, cannot be populated with fact instances, require complex design choices about attributes, lack the expressibility and simplicity of a role-based notation for constraints, hide information about the semantic domains which glue the model together, and lack adequate support for formal transformations. Many different ER notations exist that differ in the concepts they can express and the symbols used to express these concepts. For such reasons we prefer ORM for conceptual modeling. In addition to ORM, VEA supports IDEF1X (a hybrid of ER and relational modeling) as a view of ORM.

Although the detailed picture provided by ORM diagrams is often desirable, for summary purposes it is useful to hide or compress the display of much of this detail. Though not discussed here, various abstraction mechanisms exist for doing this. If desired, ER diagrams can also be used for providing compact summaries, and are best developed as views of ORM diagrams.

This overview conveys the main ideas in ORM by discussing a case study. First we explain the steps used to develop a conceptual design. To help communicate the ideas, we deliberately make some mistakes, and later show how the design method helps to correct these errors. We also include a simple example to show how the conceptual design may be “optimized” for relational systems by applying a transformation.

An algorithm for mapping this design to a normalized, relational database schema is then outlined. With VEA, the conceptual design can be entered in either graphical or textual form, and automatically mapped to a relational schema for use in a variety of relational DBMSs. Finally, a brief sketch is given of how ORM may be used as a sound basis for conceptual queries. For a detailed discussion of ORM, see [1]. For a tutorial on how to use the VEA tool to create ORM models and map them to relational models, see [5, 6, 7]. For further resources on ORM, see www.orm.net.

The Conceptual Schema Design Procedure

The information systems life cycle typically involves several stages: feasibility study; requirements analysis; conceptual design of data and operations; logical design; external design; prototyping; internal design and implementation; testing and validation; and maintenance. ORM's *conceptual schema design procedure* (CSDP) focuses on the analysis and design of data. The conceptual schema specifies the information structure of the application: the *types of fact* that are of interest; *constraints* on these; and perhaps *derivation rules* for deriving some facts from others.

With large-scale applications, the UoD is divided into convenient modules, the CSDP is applied to each, and the resulting subschemas are integrated into the global conceptual schema. The CSDP itself has seven steps (see Table 1). The rest of this section illustrates the basic working of this design procedure by means of a simple example.

Table 1 The conceptual schema design procedure (CSDP)

<i>Step</i>	<i>Description</i>
1	Transform familiar information examples into elementary facts, and apply quality checks
2	Draw the fact types, and apply a population check
3	Check for entity types that should be combined, and note any arithmetic derivations
4	Add uniqueness constraints, and check arity of fact types
5	Add mandatory role constraints, and check for logical derivations
6	Add value, set comparison and subtyping constraints
7	Add other constraints and perform final checks

Step 1 is the most important stage of the CSDP. Examples of the kinds of information required from the system are verbalized in natural language. Such examples are often available in the form of output reports or input forms, perhaps from a current manual version of the required system. If not, the modeler can work with the client to produce examples of output reports expected from the system. To avoid misinterpretation, it is usually necessary to have a UoD expert (a person familiar with the application) perform or at least check the verbalization. As an aid to this process, the speaker imagines he/she has to convey the information contained in the examples to a friend over the telephone.

For our case study, we consider a fragment of an information system used by a university to maintain details about its academic staff and academic departments. One function of the system is to print an academic staff directory, as exemplified by the report extract shown in Table 2. Part of the modeling task is to clarify the meaning of terms used in such reports. The descriptive narrative provided here would thus normally be derived from a discussion with the UoD expert. The terms “empNr” and “extNr” abbreviate “employee number” and “extension number”.

Table 2 Extract from a directory of academic staff

<i>Emp Nr</i>	<i>Emp Name</i>	<i>Dept</i>	<i>Room</i>	<i>Phone Ext.</i>	<i>Phone Access</i>	<i>Tenure/ Contract-expiry</i>
715	Adams A	Computer Science	69-301	2345	LOC	01/31/95
720	Brown T	Biochemistry	62-406	9642	LOC	01/31/95
139	Cantor G	Mathematics	67-301	1221	INT	tenured
430	Codd EF	Computer Science	69-507	2911	INT	tenured
503	Hagar TA	Computer Science	69-507	2988	LOC	tenured
651	Jones E	Biochemistry	69-803	5003	LOC	12/31/96
770	Jones E	Mathematics	67-404	1946	LOC	12/31/95
112	Locke J	Philosophy	1-205	6600	INT	tenured
223	Mifune K	Elec. Engineering	50-215A	1111	LOC	tenured
951	Murphy B	Elec. Engineering	45-B19	2301	LOC	01/03/95
333	Russell B	Philosophy	1-206	6600	INT	tenured
654	Wirth N	Computer Science	69-603	4321	INT	tenured
...

A phone extension may have access to local calls only (“LOC”), national calls (“NAT”), or international calls (“INT”). International access includes national access, which includes local access. In the few cases where different rooms or staff have the same extension, the access level is the same. An academic is either tenured or on contract. Tenure guarantees employment until retirement, while contracts have an expiry date.

The information contained in Table 2 is to be stated in terms of *elementary facts*. Basically, an elementary fact asserts that a particular object has a property, or that one or more objects participate in a relationship, where that relationship cannot be expressed as a conjunction of simpler (or shorter) facts. For example, to say that Bill Clinton jogs and is the president of the USA is to assert two elementary facts. See if you can read off the elementary facts expressed on the first row of Table 2 before reading on.

As a first attempt, one might read off the information on the first row as the six facts f1-f6. Each asserts a binary relationship between two objects. For discussion purposes the relationship type, or logical *predicate*, is shown in **bold** between the noun phrases that identify the objects. Object types are displayed here in *italics*. For compactness, some obvious abbreviations have been used (“empNr”, “EmpName”, “Dept”, “extNr”); when read aloud these can be expanded to “employee number”, “Employee name”, “Department” and “extension number”.

- f1 The *Academic* with empNr 715 **has** *EmpName* ‘Adams A’.
- f2 The *Academic* with empNr 715 **works for** the *Dept* named ‘Computer Science’.
- f3 The *Academic* with empNr 715 **occupies** the *Room* with roomNr ‘69-301’.
- f4 The *Academic* with empNr 715 **uses** the *Extension* with extNr ‘2345’.
- f5 The *Extension* with extNr ‘2345’ **provides** the *AccessLevel* with code ‘LOC’.
- f6 The *Academic* with empNr 715 **is contracted till** the *Date* with mdy-code ‘01/31/95’.

Row two contains different instances of these six fact types. Row three, because of its final column, provides an instance of a seventh fact type:

- f7 The *Academic* with empNr 139 **is tenured**.

This is called a unary fact—it specifies one property of an object. A logical predicate may be regarded as a sentence with one or more “object-holes” in it—each hole is filled in by a term or noun phrase that identifies an object. The number of object-holes is called the *arity* of the predicate. Each of these holes determines a different *role* that is played in the predicate. For example, in f4 the academic plays the role of using, and the extension plays the role of being used. In f7 the academic plays the role of being tenured. On a diagram, each role is depicted as a separate box (see later).

Object-Role Modeling is so-called because it views the world in terms of objects playing roles. Facts are assertions that objects play roles. An *n*-ary fact has *n* roles. It is not necessary that the roles be played by different objects. For example, consider the binary fact type: Person voted for Person. This has two roles (voting, and being voted for), but both could be played by the same object (e.g. Bill Clinton voted for Bill Clinton).

In FORML a predicate may have any arity (1, 2, 3 ..), but since the predicate is elementary, arities above 3 or 4 are rare. In typical applications, most predicates are binary. For these, we allow the *inverse predicate* to be stated as well, so that the fact can be read in both directions. For example, the inverse of f4 is:

- f4' The *Extension* with extNr ‘2345’ **is used by** the *Academic* with empNr 715.

To save writing, both the normal predicate and its inverse are included in the same declaration, with the inverse predicate preceded by a slash “/”. For example:

- f4" The *Academic* with empNr 715 **uses /is used by** the *Extension* with extNr ‘2345’.

Typically, predicate names are unique in the conceptual schema. In special cases however (e.g. “has”), the same name may be used externally for different predicates: internally these are assigned different identifiers.

As a quality check at Step 1, we ensure that objects are well identified. Basic objects are either values or entities. *Values* are character strings or numbers: they are identified by constants (e.g. ‘Adams A’, 715). *Entities* are “real world” objects that are identified by a definite description (e.g. the *Academic* with empNr

715). In simple cases, such a description indicates the entity type (e.g. Academic), a value (e.g. 715) and a *reference mode* (e.g. empNr). A reference mode is the manner in which the value refers to the entity. Entities may be tangible objects (e.g. persons, rooms) or abstract objects (e.g. access levels). Composite reference schemes are possible (see later).

Fact f1 involves a relationship between an entity (a person) and a value (a name is just a character string). Facts f2-f6 specify relationships between entities. Fact f7 states a property (or unary relationship) of an entity. In setting out facts f1-f7, the employeeNr is unquoted while both extNr and roomNr are quoted. This indicates the designer treated employeeNr as a number, but considered extNr and roomNr as character strings. However unless arithmetic operations are required for empNr it could have been quoted. Unless extNr and roomNr must permit non-digits (e.g. hyphens or letters), or string operations are needed for them, they could have been unquoted.

As a second quality check at Step 1, we use our familiarity with the UoD to see if some facts should be split or recombined (a formal check on this is applied later). For example, suppose facts f1 and f2 were verbalized as the single fact f8.

f8 The *Academic* with empNr 715 and empname 'Adams A' **works for** the *Dept* named 'Computer Science'.

The presence of the word “and” suggests that f8 may be split without information loss. The repetition of “Jones E” on different rows of Table 2 shows that academics cannot be identified just by their name. However the uniqueness of empNr in the sample population suggests that this suffices for reference. Since the “and-test” is only a heuristic, and sometimes a composite naming scheme is required for identification, the UoD expert is consulted to verify that empNr by itself is sufficient for identification. With this assurance obtained, f8 is now split into f1 and f2.

As an alternative to specifying complete facts one at a time, the reference schemes may be declared up front and then assumed in later facts. Simple reference schemes are declared by enclosing the reference mode in parenthesis. For example, the entity types and their identification schemes may be declared thus:

Reference schemes:

Academic (empNr);
Dept (name);
Room (roomNr);
Extension (extNr);
AccessLevel (code);
Date (mdy)

Then facts f1-f7 may be stated more briefly as follows. Here the names of object types begin with a capital letter.

f1 Academic 715 has EmpName 'Adams A'.
f2 Academic 715 works for Dept 'Computer Science'.
f3 Academic 715 occupies Room '69-301'.
f4 Academic 715 uses Extension '2345'.
f5 Extension '2345' provides AccessLevel 'LOC'.
f6 Academic 715 is contracted till Date '01/31/95'.
f7 Academic 139 is tenured.

These facts are instances of the following fact types:

F1 Academic has EmpName
F2 Academic works for Dept
F3 Academic occupies Room
F4 Academic uses Extension
F5 Extension provides AccessLevel
F6 Academic is contracted till Date
F7 Academic is tenured

Step 2 of the CSDP is to *draw a draft diagram of the fact types and apply a population check* (see Figure 1). Entity types are depicted as named ellipses. Predicates are shown as named sequences of one or more role boxes. Predicate names are read left-to-right and top-to-bottom, unless prepended by “<<” (which

reverses the reading direction). An n -ary predicate has n role boxes. The inverse predicate names have been omitted in this figure. Value types are displayed as named, broken ellipses. Lines connect object types to the roles they play. Reference modes are written in parenthesis: this is an abbreviation for the explicit portrayal of reference types. For example, the notation “Academic (empNr)” indicates an injection (1:1-into mapping) from the entity type Academic to the value type EmpNr.

In this example there are seven fact types. As a check, each has been populated with at least one fact, shown as a row of entries in the associated fact table, using the data from rows 1 and 3 of Table 2. The English sentences listed before as facts f1-f7, as well as other facts from row 3, may be read directly off this figure. Though useful for validating the model with the client and for understanding constraints, the sample population is not part of the conceptual schema diagram itself.

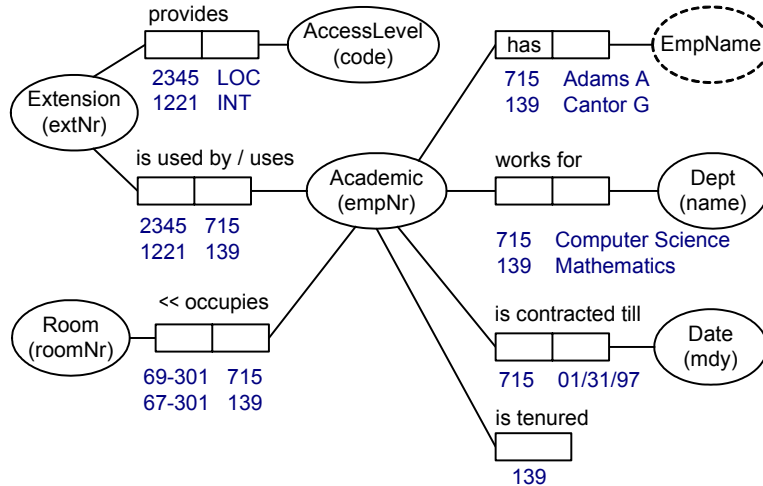


Figure 1 Draft diagram of fact types for Table 2 with sample population

To help illustrate other aspects of the CSDP we now widen our example. Suppose the information system is also required to assist in the production of departmental handbooks. Perhaps the task of schema design has been divided up, and another modeler works on the subschema relevant to department handbooks. Figure 2 shows an extract from a page of one such handbook.

Department:	Computer Science	<i>Home phone of Dept head: 9765432</i>
<i>Chairs</i>	<i>Professors (5)</i>	
Databases	Codd EF	BSc (UQ); PhD (UCLA) (<i>Head of Dept</i>)
Algorithms	Wirth N	BSc (UQ); MSc (ANU); DSc (MIT)
...	...	
<i>Senior Lecturers (9)</i>		
Hagar TA	BInfTech (UQ); PhD (UQ)	
...	...	
<i>Lecturers (8)</i>		
Adams A	MSc (OXON)	
...	...	

Figure 2 Extract from Handbook of Computer Science Department

In this university academic staff are classified as professors, senior lecturers or lecturers, and each professor holds a “chair” in a research area. To reduce the size of our problem, we have excluded many details that in practice would also be recorded (e.g. office phone and faxNr). To save space, details are shown here for only four of the 22 academics in that department. The data are of course, fictitious.

In verbalizing a report, at least one instance of each fact type should be stated. Let us suppose that the designer for this part of the application suggests the following fact set, after first declaring the following reference schemes: Dept (name); Professor (name); SeniorLecturer (name); Lecturer (name); Quantity (nr)+; Chair (name); Degree (code); University (code); HomePhone (phoneNr). The “+” in “Quantity (nr)+” indicates that Quantity is referenced by a number, not a character string, and hence can be operated on by numeric operators such as “+”. For discussion purposes, the predicates are shown here in bold.

- f9 Dept 'Computer Science' **has professors in** Quantity 5.
- f10 Professor 'Codd EF' **holds** Chair 'Databases'.
- f11 Professor 'Codd EF' **obtained** Degree 'BSc' **from** University 'UQ'.
- f12 Professor 'Codd EF' **heads** Dept 'Computer Science'.
- f13 Professor 'Codd EF' **has** HomePhone '965432'.
- f14 Dept 'Computer Science' **has senior lecturers in** Quantity 9.
- f15 SeniorLecturer 'Hagar TA' **obtained** Degree 'Blntech' **from** University 'UQ'.
- f16 Department 'Computer Science' **has lecturers in** Quantity 8.
- f17 Lecturer 'Adams A' **obtained** Degree 'MSc' **from** University 'OXON'.

As a quality check for Step 1 we again consider whether entities are well identified. It appears from the handbook example that within a single department, academics may be identified by their name. Let us assume this is verified by the domain expert. However the complete application requires us to handle all departments in the same information system, and to integrate this subschema with the directory subschema considered earlier.

Hence we must replace the academic naming convention used for the handbook example by the global scheme used earlier (i.e. empNr). Suppose that we can't see anything else wrong with facts f9-17, and proceed to expand the draft schema diagram to include this new information (this is left as an exercise for the reader).

This leads us to *Step 3* of the CSDP: *check for entity types that should be combined, and note any arithmetic derivations*. The first part of this step prompts us to look carefully at the fact types for f11, f15 and f17. Currently these are handled as three ternary fact types: Professor obtained Degree from University; SeniorLecturer obtained Degree from University; Lecturer obtained Degree from University.

The common predicate suggests that the entity types Professor, SeniorLecturer and Lecturer should be collapsed to the single entity type Academic, with this predicate now shown only once, as shown in Figure 3.

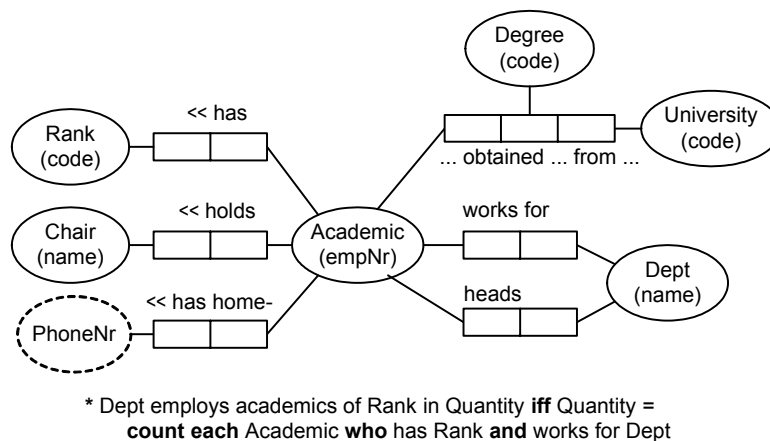


Figure 3 Extra fact types needed to capture the additional information in Figure 2

To preserve the original information about who is a professor, senior lecturer or lecturer we introduce the fact type: Academic has Rank. Let's use the codes "P", "SL" and "L" for the ranks of professor, senior lecturer and lecturer. For example, instead of fact f10 we now have:

- f18 Academic 430 has EmpName 'Codd EF'.
- f19 Academic 430 has Rank 'P'
- f20 Academic 430 holds Chair 'Databases'.

Facts of the kind expressed in f9, f14 and f16 can now all be expressed in terms of the ternary fact type: Dept employs academics of Rank in Quantity. For example, f9 can be replaced by:

- f9' Dept 'Computer Science' **employs academics of Rank 'P' in Quantity 5.**

However, this does not tell us *which* professors work for the Computer Science department. Indeed, given that many departments exist, the verbalization in f9-f17 failed to capture the information about who worked for that department. This information is implicit in the listing of the academics in the Computer Science handbook. To capture this information in our application model, we introduce the following fact type: Academic works for Dept. For example, one fact of this kind is:

- f21 Academic 430 **works for** Dept 'Computer Science'

The second aspect of Step 3 is to see if some fact types can be derived from others by arithmetic. Since we now record the rank of academics as well as their departments, we can compute the number in each rank in each department simply by counting. So facts like f9' are *derivable*. If desired, derived fact types may be included on a schema diagram if they are marked with an asterisk "*" to indicate their derivability. To simplify the picture, it is usually better to omit derived predicates from the diagram. However in all cases a derivation rule must be supplied. This may be written below the diagram (see Figure 3). Here "iff" abbreviates "if and only if".

Step 4 of the CSDP is to *add uniqueness constraints and check the arity of the fact types*. Uniqueness constraints are used to assert that entries in one or more roles occur there *at most once*. A bar across n roles of a fact type ($n > 0$) indicates that each corresponding n -tuple in the associated fact table is unique (no duplicates are allowed for that column combination). Arrow tips at the ends of the bar are needed if the roles are non-contiguous (otherwise arrow tips are optional). A uniqueness constraint spanning roles of different predicates is indicated by a circled "u": this specifies that in the natural join of the predicates, the combination of connected roles is unique.

For example, a fragment of the conceptual schema under consideration is displayed in Figure 4. While these constraints are suggested by the original population, the domain expert should normally be consulted to verify them. It is sometimes helpful to construct a test population for each fact type in this regard, though simple questions are usually more efficient. The internal uniqueness constraints on the binary fact types assert that each academic has at most one rank, holds at most one chair (and vice versa), works for at most one department, and has at most one employee name.

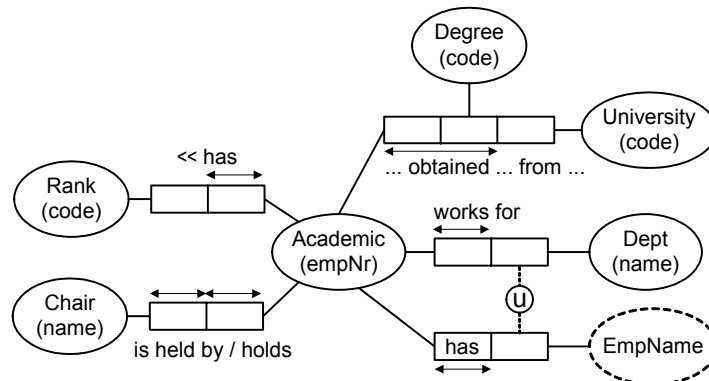


Figure 4 Some of the fact types, with uniqueness constraints added

The external uniqueness constraint stipulates that each (department, empname) combination applies to at most one academic. That is, within the same department, academics have distinct names. The constraint on the ternary says that for each (academic, degree) pair, the award was obtained at only one university.

Once uniqueness constraints have been added, an arity check is performed. A sufficient but not necessary condition for splittability of an n -ary fact type is that it has a uniqueness constraint that misses two roles. For example, suppose we tried to use the ternary in Figure 5(a). Since each academic has only one rank and works for only one department, the uniqueness constraint spans just the first role. This misses two roles of the ternary; so the fact type must be split on the source of the uniqueness constraint into the two binaries Academic has Rank and Academic works for Dept as shown in Figure 5(b).

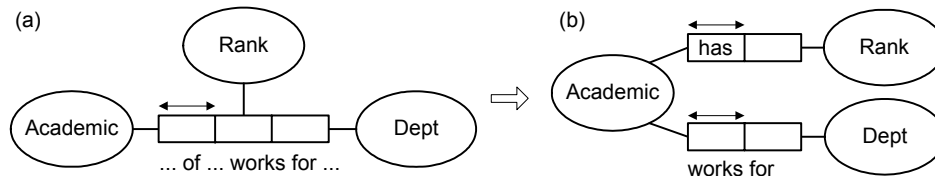


Figure 5 This fact type splits since 2 roles are missed by the uniqueness constraint

If a fact type is elementary all its functional dependencies (FDs) are implied by uniqueness constraints. For example, each academic has only one rank (hence in the fact table for Academic has Rank, entries in the rank column are a function of entries in the academic column). If in doubt, one checks for FDs not so implied; if such an FD is found, the fact type is split on the source of the FD.

Step 5 of the CSDP is to *add mandatory role constraints, and check for logical derivations*. A role is mandatory (or total) for an object type if and only if every object of that type which is referenced in the database must be known to play that role. This is explicitly shown by means of a *mandatory role dot* where the role connects with its object type. If two or more roles are connected to a circled mandatory role dot, this means the *disjunction* of the roles is mandatory (i.e. each object in the population of the object type must play at least one of these roles)—an *inclusive-or* constraint.

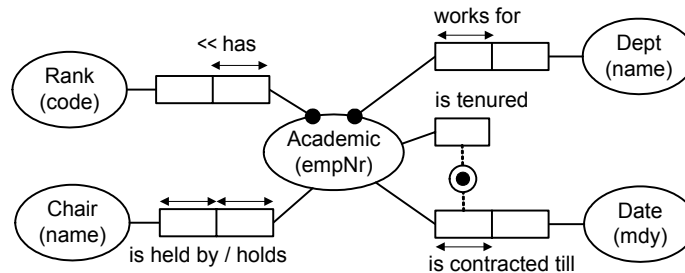


Figure 6 Some of the fact types, with mandatory role constraints added

For example, Figure 6 adds mandatory role constraints to some of the fact types already discussed. These dots indicate that each academic has a rank and works for a department; moreover each academic *either* is tenured *or* is contracted till some date. Roles that are not mandatory are *optional*. The role of having a chair is optional. The roles of being contracted or being tenured are optional too, but their disjunction is mandatory. If an object type plays only one fact role in the global schema, then by default this is mandatory, but a dot is not shown (e.g. the role played by Rank is mandatory by implication).

Now that uniqueness and mandatory role constraints have been discussed, reference schemes can be better understood. Simple reference schemes involve a mandatory 1:1 mapping from entity type to value type. For example, the notation “Rank (code)” abbreviates the binary reference type: Rank has Rankcode. If shown explicitly, both roles of this binary have a simple uniqueness constraint, and the reference role played by Rank has a mandatory role dot.

With composite reference, a combination of two or more values can be used to refer to an entity. For example, while EmpNr provides a simple primary identifier for Academic, the combination of Dept and EmpName provides a secondary identification scheme. Sometimes composite schemes are used for primary

reference. For example, suppose that to help students find their way to lectures, departmental handbooks include a building directory, which lists the names as well as the numbers of buildings. A sample extract of such a directory is shown in Table 3.

Table 3 Extract from a directory of buildings

<i>BuildingNr</i>	<i>Building name</i>
...	...
67	Priestly
68	Chemistry
69	Computer Science
...	...

Earlier we identified rooms by a single value. For example “69-301” was used to denote the room in building 69 which has room number “301”. Now that buildings are to be talked about in their own right, we should replace the simple reference scheme by a composite one that shows the full semantics (see Figure 7). Here RoomNr now means just the number (e.g. “301”) used to identify the room within its building. This is used in conjunction with the buildingNr to identify the room within the whole university. To explicitly indicate that the external uniqueness constraint provides the *primary* reference for Room, the circled “u” may be replaced by a circled “P” (not shown here).

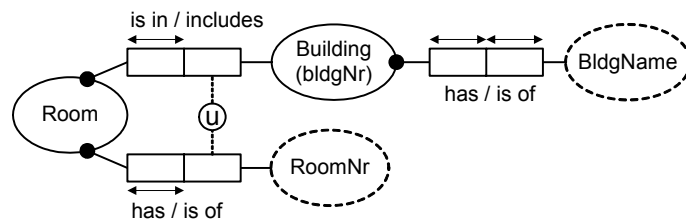


Figure 7 Room has a composite, primary reference scheme

Knowledge of uniqueness constraints and mandatory roles can assist in deciding when to *nest* a fact type. The ternary in Figure 4 could have been modeled by nesting as follows. First declare the binary: Academic obtained Degree. Now objectify this relationship as “DegreeAcquisition” (graphically this is depicted as a soft rectangle enveloping the predicate being objectified). Now attach another binary predicate to this to connect it to University. This yields the nested version: DegreeAcquisition(Academic obtained Degree) was from University.

In this case, the objectified predicate plays only one role, and this role is mandatory. Whenever this happens we prefer the flattened version instead of the nested version, since it is more compact and natural, and it simplifies constraint expression. In all other cases, the nested version is to be preferred (i.e. choose nesting if the objectified predicate plays an optional role, or plays more than one role).

As an example, suppose the application also has to deal with reports about teaching commitments, an extract of which is shown in Table 4. Not all academics currently teach. If they do, their teaching in one or more subjects may be evaluated and given a rating. Some teachers serve on course curriculum committees.

Table 4 Extract of report on teaching commitments

<i>EmpNr</i>	<i>Emp. name</i>	<i>Subject</i>	<i>Rating</i>	<i>Committees</i>
715	Adams A	CS100 CS101	5	
430	Codd EF			
654	Wirth N	CS300		BSc-Hons CAL Advisory

Here the new fact types may be schematized as shown in Figure 8. By default, an objectified predicate is fully spanned by a uniqueness constraint, to ensure elementarity (this is implicit in the frame notation, but may be shown explicitly as in the figure). Since not all (Academic, Subject) pairs involved in Teaching have a rating, nesting is preferred. To flatten this we would need a binary for teaching subjects, and a ternary for rating the teaching of subjects, with a pair subset constraint (see later) between them.

The nested object type Teaching plays only one role, and this role is optional. So instances of Teaching can exist independently without having to play a fact role. This makes teaching an *independent* object type. In VEA, the independent status of an object type is set by checking the “Independent” option in the object type’s properties sheet: this automatically appends “!” to the graphic display of the object type’s name.

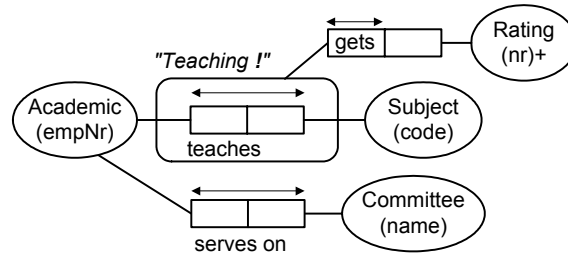


Figure 8 Example of nesting

The second stage of Step 5 is to check for logical derivations (i.e. can some fact type be derived from others without the use of arithmetic?). One strategy here is to ask whether there are any relationships (especially functional relationships) which are of interest but which have been omitted so far.

Another strategy is to look for transitive patterns of functional dependencies. For example, if an academic has only one phone extension and an extension is in only one room, we could use these to determine the room of the academic. However, for our application the same extension may be used in many rooms, so we discard this idea.

Suppose however that our client confirms that the rank of an academic determines the access level of his/her extension. For example, suppose a current business rule is that professors get international access while lecturers and senior lecturers get local access. This rule might change in time (e.g. senior lecturers might be arguing for national access). So to minimize later changes to the schema, we store the rule as data in a table (see Table 5). The rule can then be updated as required by an authorized user without having to recompile the schema.

Table 5 A functional connection between rank and access level

Rank	Access
P	INT
SL	LOC
L	LOC

Suppose we verbalize the fact type underlying Table 5 as: Rank ensures AccessLevel. These three lines of data can be used to derive the access level of the hundreds of academic extensions, using the following derivation rule:

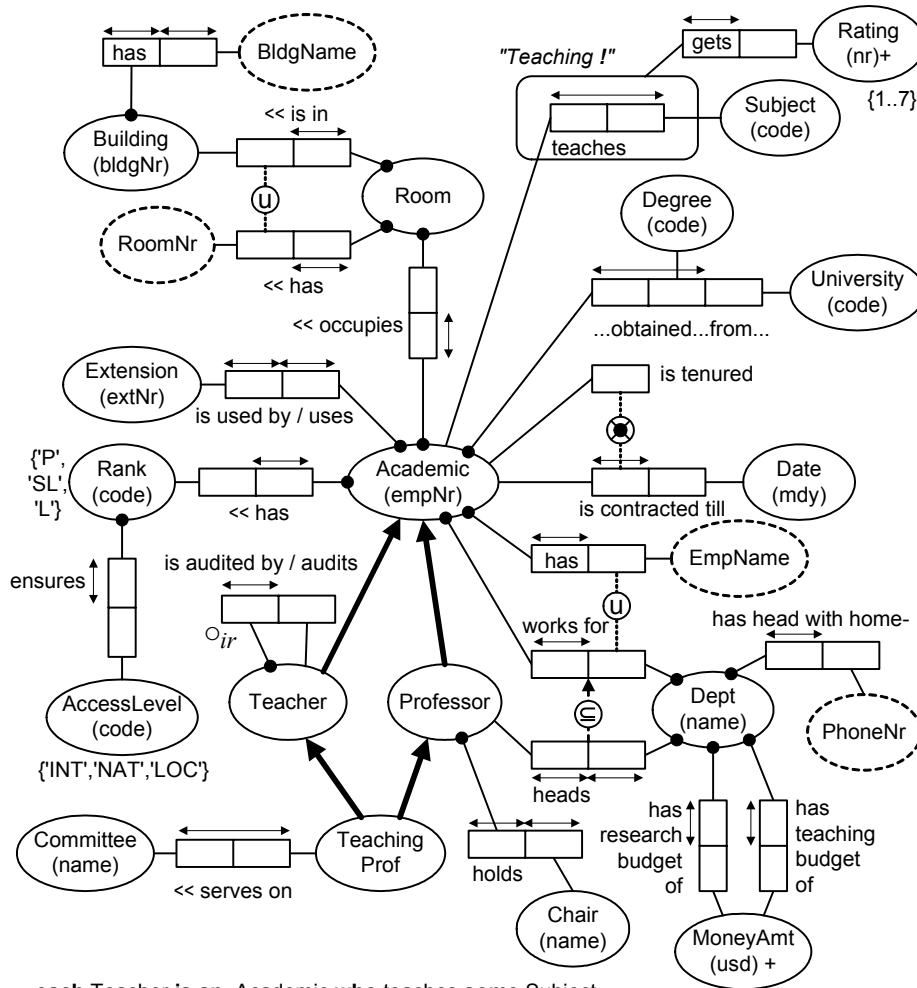
Extension provides AccessLevel iff
 Extension is used by **an** Academic
who has **a** Rank **that** ensures AccessLevel

Examination of the related portion of the schema indicates that this rule is safe only if each extension is used by only one academic, or at least only by academics of the same rank. Let us assume the first, stronger condition is verified by the client. In the case of the weaker condition, the constraint must be

specified textually rather than on the diagram. At any rate, by adding the Rank ensures AccessLevel fact type and the above derivation rule, we can remove the Extension provides AccessLevel fact type from the diagram.

In *Step 6* of the CSDP we add *any value, set comparison and subtyping constraints*. Value constraints specify a list of possible values for a value type. These usually take the form of an enumeration or range, and are displayed in braces besides the value type or its associated entity type. For example, Rankcode is restricted to {'P', 'SL', 'L'} and AccessLevelcode to {'INT', 'NAT', 'LOC'}. These are displayed in the global conceptual schema (Figure 9).

Set comparison constraints specify subset, equality or exclusion constraints between compatible roles or sequences of compatible roles. Compatible roles are played by the same object type (or by object types with a common supertype—see later). A subset constraint from one role sequence to another indicates that the population of the first must always be a subset of the second, and is denoted by a circled " \subseteq " with a dotted arrow from source to target. In Figure 9, a pair-subset constraint runs from the heads predicate to the works for predicate, indicating that a person who heads a department must work for the same department.



each Teacher is an Academic who teaches some Subject
each Professor is an Academic who has Rank 'P'
each TeachingProf is both a Teacher and a Professor

* Dept employs academics of Rank in Quantity **iff** Quantity =
count each Academic who has Rank and works for Dept

* **define** Extension provides AccessLevel as
 Extension is used by **an Academic who has a Rank that ensures AccessLevel**

Figure 9 The final conceptual schema

An equality constraint, denoted by a circled “=”, is equivalent to a pair of subset constraints (one in each direction). For example, in this application a person’s home phone is recorded if and only if the person heads some department. This could be depicted by an equality constraint between the first roles of two fact types: Professor heads Dept; Professor has HomePhoneNr. However we later choose another way of modeling this. The constraint that nobody can be tenured and contracted at the same time is shown by an exclusion constraint, denoted by a circled “X”. In this case, it overlays an inclusive-or constraint (circled dot) so the combination of both constraints appears as “lifebuoy symbol” (*exclusive-or* constraint).

Subtyping is determined as follows. Each optional role is inspected: if the role is played only by some well-defined subtype, a subtype node is introduced with this role attached. Subtype definitions are written below the diagram and subtype links are shown as directed line segments from subtypes to supertypes. Figure 9 contains three subtypes: Teacher; Professor; and TeachingProfessor. In this university, each teacher is audited by another teacher (auditing involves observation and friendly feedback). Moreover, only professors may be department heads, and only teaching professors can serve on curriculum committees (not all universities work this way).

Step 7 of the CSDP adds other constraints and performs final checks. We briefly illustrate two other constraints. The audits fact type has both its roles played by the same object type (this is called a ring fact type). The ^o*ir* notation beside it indicates the predicate is *irreflexive* (no teacher audits himself/herself).

Suppose we also need to record the teaching and research budgets of the departments. We might schematize this as in Figure 10. Here the “2” beside the role played by Dept is a *frequency constraint* indicating that each department that is included in the population of that role must appear there twice. In conjunction with the other constraints, this ensures that each department has both its teaching and research budgets recorded.

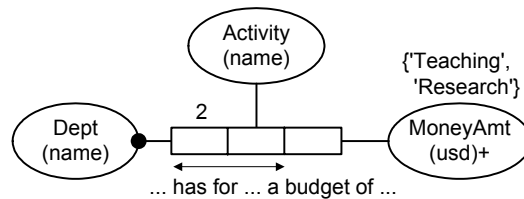


Figure 10 Each department has two budgets

The CSDP ends with some final checks that the schema is consistent with the original examples, avoids redundancy, and is complete. No changes are needed for our example. There is a minor derived redundancy, since if someone heads a department, we know from the subset constraint that this person works for that department; but this is innocuous. Other schematizations are possible (e.g. we can define works in and heads to be pair-exclusive, or use a unary is head instead of the binary heads) but we ignore these alternatives here.

Once the global schema is drafted, and the target DBMS decided, various optimizations can usually be performed to improve the efficiency of the logical schema that results from the mapping. Assuming the conceptual schema is to be mapped to a relational database schema, the fact type in Figure 10 will map to a separate table all by itself (because of its composite uniqueness constraint). Since some other information about departments is mapped to another table, if we want to retrieve all the details about departments in a single query we will have to perform a table join. Joins tends to slow things down.

Moreover, we probably want to compute the total budget of a department, and with the current schema this will involve a self-join of the table since the details of the two budgets are on separate rows. We can avoid all these problems by transforming the ternary fact type in Figure 10 into the following two binaries before we map: Dept has teaching budget of MoneyAmt; Dept has research budget of MoneyAmt. These binaries have simple keys, and will map to the “main” department table.

Another optimization may be performed which moves the home phone information to Dept instead of Professor, but the steps underlying this are a little advanced, so we ignore a detailed discussion of this move here. Figure 10 includes both these revisions to the conceptual schema. For a detailed discussion on conceptual schema optimization, see [1, chapter 12].

Relational mapping

Once the conceptual schema has been specified, a simple algorithm is used to group these fact types into normalized tables. If the conceptual fact types are elementary (as they should be), then the mapping is guaranteed to be free of redundancy, since each fact type is grouped into only one table, and fact types which map to the same table all have uniqueness constraints based on the same attribute(s).

Before discussing the mapping, we define a few terms. A *simple key* may be thought of as a uniqueness constraint spanning exactly one role; a *composite key* is a uniqueness constraint spanning more than one role. A *compidot* (*compositely identified object type*) is either a nested object type (an objectified predicate) such as Teaching, or a co-referenced object type (its primary reference scheme is based on an external uniqueness constraint) such as Room. The basic stages in the mapping algorithm are as follows.

- 1 Initially treat each compidot as an atomic “black box” by mentally erasing any predicates used in its identification, and absorb subtypes into their supertype.
- 2 Map each fact type with a composite key into a separate table, basing the primary key on this key.
- 3 Group fact types with simple keys attached to a common object type into the same table, basing the primary key on the identifier of this object type.
- 4 Unpack each mapped compidot into its component attributes.

With stage 3, a choice may arise with 1:1 binaries. If one role is optional and the other mandatory then the fact type is grouped with the object type on the mandatory side. For example, the head-of-department fact type is grouped into the department table. Other refinements to the algorithm have been developed (e.g. other options for 1:1 cases and subtyping, mapping of independent object types, certain derived fact type cases, and partially null keys) but we do not consider these here.

Conceptual constraints and derivation rules are also mapped down. An exhaustive treatment of the mapping procedure is beyond the scope of this paper. The conceptual schema under discussion maps to the relational schema shown in Figure 11. A generic notation (partly graphical) is used to specify the tables and constraints of resulting relational schema, and derivation rules are expressed as SQL views.

Keys are underlined. If alternate keys exist, the primary key is doubly-underlined. A mandatory role is captured by making its corresponding attribute mandatory in its table (not null is assumed by default), by marking as optional (in square brackets) all optional roles for the same object type which map to the same table, and by running an equality/subset constraint from those mandatory/optional roles which map to another table.

Most conceptual constraint notations map down with little change. Constraints on lists of role-lists (e.g. subset, equality, exclusion) map to corresponding constraints on the attributes to which they map. Equality constraints may be shown without arrowheads. Subtype constraints map to qualifications on optional columns or subset constraints (e.g. foreign key constraints).

Conceptual object types are semantic domains: as current relational systems do not support this feature, domain names are usually omitted. Syntactic domains (data types) may be specified next to the column names if desired: if the reference mode has a “+”, the default data type is numeric, else the default is character string; the designer typically chooses more specific data subtypes as appropriate.

The $\langle 2,1 \rangle$ in the pair-subset constraint indicates the source pair should be reversed before the comparison, that is the ordered pairs populating Department(headempNr, deptname) must also occur in the population of Academic(empNr, deptname).

Derived tables are shown below the base tables. The notation “ $R(..) ::=$ ” is short for “**create view** $R(..)$ **as select**”. As with conceptual schemas, relational schemas may be displayed with levels of information hiding (e.g. for a brief overview, some or all of the constraint layers may be suppressed).

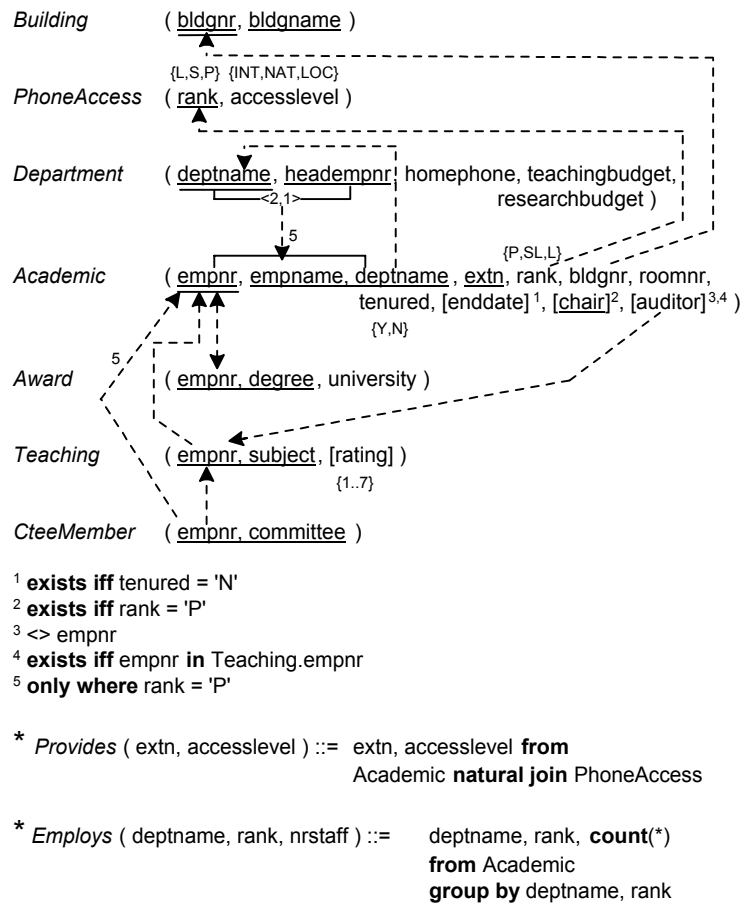
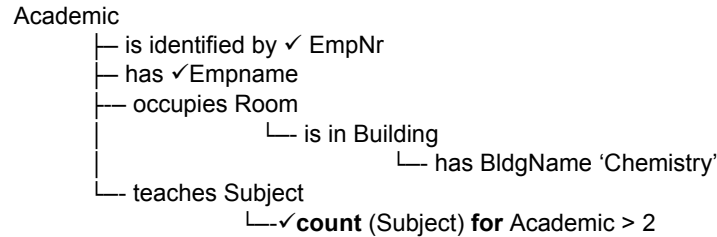


Figure 11 The relational schema mapped from Figure 9

Conceptual queries

Besides information modeling, ORM is also ideal for information querying. In 1997, InfoModelers Inc. released a restricted version of a powerful ORM query tool, named “ActiveQuery”, that enables existing relational models to be reverse engineered to ORM models, which may then be queried directly. Moreover, any ORM model developed in InfoModeler (version 2.0a onwards) or VisioModeler can immediately be queried with ActiveQuery, without the need for any reverse engineering. ActiveQuery enables users to query an ORM directly without prior knowledge of either the conceptual schema or the corresponding relational schema, by dragging object types onto the query pane, selecting predicates of interest, applying restrictions and functions as desired, and ticking the items to be listed. With the acquisition of InfoModelers by Visio Corporation, which in turn was acquired by Microsoft Corporation, the ActiveQuery product is no longer available. However Microsoft has made VisioModeler available as a free download, and it is possible that the technology underlying ActiveQuery may appear in some later tool.

As a simple example of a conceptual query, consider the following English query on our academic database: list the empNr, empname and number of subjects taught for each academic who occupies a room in the Chemistry building and teaches more than two subjects. In ActiveQuery this may be formulated by drag-and-drop basically as follows:



A verbalization of the query is automatically generated, as well as SQL code similar to the following:

```

select X1.empnr, X1.emprname, count(*)
from Academic as X1, Building as X2, Teaching as X3
where X1.bldgnr = X2.bldgnr
      and X1.empnr = X3.empnr
      and X2.bldgname = 'Chemistry'
group by X1.empnr, X1.emprname
having count(*) >2
  
```

It should be obvious that formulating queries in terms of objects and predicates is much easier than deciphering the semantics of the relational schema and coding in SQL or QBE. For further details about conceptual queries in ORM, see [1, 2, 3].

References

1. Bloesch, A.C. & Halpin, T.A. 1996, 'ConQuer: a conceptual query language', Proc. ER'96: 15th Int. Conf. on conceptual modeling, Springer LNCS, no. 1157, pp. 121-33 (online at www.orm.net).
2. Bloesch, A.C. & Halpin, T.A. 1997, 'Conceptual queries using ConQuer-II', Proc. ER'97: 16th Int. Conf. on conceptual modeling, Springer LNCS, no. 1331, pp. 113-26 (online at www.orm.net).
3. Halpin, T.A. 1998, 'Conceptual Queries', Database Newsletter, vol. 26, no. 2, ed. R.G. Ross, Database Research Group, Inc., Boston MA (March/April 1998) (online at www.orm.net).
4. Halpin, T.A. 2001a, *Information Modeling and relational Databases*, Morgan Kaufmann Publishers, San Francisco (www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-672-6).
5. Halpin, T.A. 2001b, 'Microsoft's new database modeling tool: Part 1', *Journal of Conceptual Modeling*, June 2001 issue (online at www.InConcept.com and www.orm.net).
6. Halpin, T.A. 2001c, 'Microsoft's new database modeling tool: Part 2', *Journal of Conceptual Modeling*, August 2001 issue (online at www.InConcept.com and www.orm.net).
7. Halpin, T.A. 2001d, 'Microsoft's new database modeling tool: Part 3', *Journal of Conceptual Modeling*, August 2001 issue (online at www.InConcept.com and www.orm.net).

Object-Role Modeling (ORM/NIAM)

Terry Halpin

Microsoft Corporation, USA

[This paper appeared as ch. 4 of *Handbook on Architectures of Information Systems*, eds P. Bernus, K. Mertins & G. Schmidt, Springer-Verlag, Berlin, 1998, and is reproduced here by permission. Details on the book are online at www.springer.de/cgi-bin/search_book.pl?isbn=3-540-64453-9.]

Abstract: Object-Role Modeling (ORM) is a method for modeling and querying an information system at the conceptual level, and mapping between conceptual and logical (e.g. relational) levels. ORM comes in various flavors, including NIAM (Natural language Information Analysis Method). This article provides an overview of ORM, and notes its advantages over Entity Relationship and traditional Object-Oriented modeling.

1 Introduction

1.1 *ORM: what is it and why use it?*

Object-Role Modeling (ORM) is primarily a method for modeling and querying an information system at the conceptual level. In Europe, the method is often called *NIAM* (Natural language Information Analysis Method). Since information systems are typically implemented on a DBMS that is based on some logical data model (e.g. relational, object-relational, hierarchic), ORM includes procedures for mapping between conceptual and logical levels. Although various ORM extensions have been proposed for process and event modeling, the focus of ORM is on data modeling, since the data perspective is the most stable and it provides a formal foundation on which operations can be defined.

For correctness, clarity and adaptability, information systems are best specified first at the conceptual level, using concepts and language that people can readily understand. Analysis and design involves building a formal model of the application area or *universe of discourse* (UoD). To do this properly requires a good understanding of the UoD and a means of specifying this understanding in a clear, unambiguous way. Object-Role Modeling simplifies this process by using natural language, as well as intuitive diagrams that can be populated with examples, and by expressing the information in terms of elementary relationships.

ORM is so-called because it pictures the world 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. In contrast to other modeling techniques such as Entity-Relationship (ER) and Object-Oriented (OO) approaches, ORM makes no explicit use of *attributes*. For example, instead of using `countryBorn` as an attribute of `Person`, we use the relationship type `Person was born in Country`. This has many important advantages. Firstly, ORM models and queries are more stable (attributes may evolve into entities or relationships). For example, if we decide to later record the population of a country, then our `countryBorn` attribute needs to be reformulated as a relationship. Secondly, ORM models may be conveniently populated with multiple instances (attributes make this too awkward). Thirdly, ORM is more uniform (e.g. we don't need a separate notation for applying the same constraint to an attribute rather than a relationship).

ORM is typically more expressive than ER or OO. Its role-based notation makes it easy to specify a wide variety of constraints, and its object types reveal the semantic domains that bind a schema together. One benefit of this is that conceptual queries may now be formulated in terms of schema paths, where moving from one role through an object type to another role amounts to a conceptual join (see later).

Unlike ORM or ER, popular OO models often duplicate information by wrapping facts up into pairs of inverse attributes in different objects. Moreover, OO notations have weak support for constraints (e.g. a constraint might have to be duplicated in different objects, or even ignored). Unfortunately, OO models are less stable than even ER models when the UoD evolves. For such reasons, OO models should be used only for implementation, not for analysis.

Although the detailed picture provided by ORM is desirable in developing and transforming a model, for summary purposes it is useful to hide or compress the display of much of this detail. Various abstraction mechanisms exist for doing this [e.g. CHP96]. If desired, ER and OO diagrams can also be used for providing compact summaries, and are best developed as views of ORM diagrams. For a simple discussion illustrating the points in this section, see [Hal96].

The rest of this article provides a brief history of ORM, summarizes the ORM notation, illustrates the conceptual design and relational mapping procedures, and mentions some recent extensions before concluding.

1.2 *A brief history of ORM*

In the 1970s, especially in Europe, substantial research was carried out to provide higher level semantics for modeling information systems. Abrial [Abr74], Senko [Sen75] and others discussed modeling with binary relationships. In 1973, Falkenberg generalized their work on binary relationships to n-ary relationships and decided that attributes should not be used at the conceptual level because they involved “fuzzy” distinctions and also complicated schema evolution. Later, Falkenberg proposed the fundamental ORM framework, which he called the “object-role model” [Fal76]. This framework allowed n-ary and nested relationships, but depicted roles with arrowed lines.

Nijssen [Nij76] adapted this framework by introducing the circle-box notation for objects and roles that has now become standard, and adding a linguistic orientation and design procedure to provide a modeling method called ENALIM (Evolving NATural Language Information Model) [Nij77]. Nijssen led a group of researchers at Control Data in Belgium who developed the method further, including van Assche who classified object types into lexical object types (LOTs) and non-lexical object types (NOLOTs). Today, LOTs are commonly called “Entity types” and NOLOTs are called “Value types”. Kent [Ken77] provided several semantic insights and clarified many conceptual issues.

Meersman added subtypes, and made major contributions to the RIDL query language [Mee82] with Falkenberg and Nijssen. The method was renamed “aN Information Analysis Method” (NIAM) and summarized in a paper by Verheijen and van Bekkum [VB82]. In later years the acronym “NIAM” was given different expansions, and is now known as “Natural language Information Analysis Method”. Two matrix methods for subtypes were developed, one (the role-role matrix) by Vermeir [Ver83] and another by Falkenberg and others.

In the 1980s, Falkenberg and Nijssen worked jointly on the design procedure and moved to the University of Queensland, where the method was further enhanced by various academics. Halpin provided the first full formalization of the method [Hal89], including schema equivalence proofs, and made several refinements and extensions to the method. In 1989, Halpin and Nijssen co-authored a book on the method. A second edition of this book, authored by Halpin, was published in 1995 [Hal95]. Another book on the method, written by Wintraecken, was published in 1990 [Win90].

Many researchers have contributed to the ORM method over the years, and there is no space here to list them all. Today various versions of the method exist, but all adhere to the fundamental object-role framework. Although most ORM proponents favor n-ary relationships, some prefer Binary-Relationship Modeling (BRM), e.g. Shoval [SS93]. Habrias [Hab93] developed an object-oriented version called MOON (Normalized Object-Oriented Method). The Predicator Set Model (PSM) was developed mainly by ter Hofstede, Proper and van der Weide [HPW93], and includes complex object constructors. De Troyer and Meersman [DM95] developed another version with constructors called Natural Object-Relationship Model (NORM). Halpin developed an extended version called Formal ORM (FORM), and with Bloesch and others at InfoModelers Inc. developed an associated query language called ConQuer [BH97]; this work is being extended at Visio Corporation. Van der Lek and others [BZL94] allowed entity types to be treated as nested roles, to produce Fully Communication Oriented NIAM (FCO-NIAM). Embley and others [EKW92] developed Object-oriented Systems Analysis (OSA) which includes an “Object-Relationship Model” component that has much in common with standard ORM, with no use of attributes.

2 **Data modeling in ORM**

1.3 *Notation*

A modeling method includes both a notation and a procedure for using its notation. This subsection discusses notation, and later subsections discuss procedures. Each well-defined version of ORM includes a formal, textual specification language for both models and queries, as well as a formal, graphical modeling language. The textual languages are more expressive than the graphical languages, but are mentioned only briefly in this paper. Figure 1 summarizes most of the main symbols used in the graphical language. We now briefly describe each symbol. Examples of these symbols in use are given later.

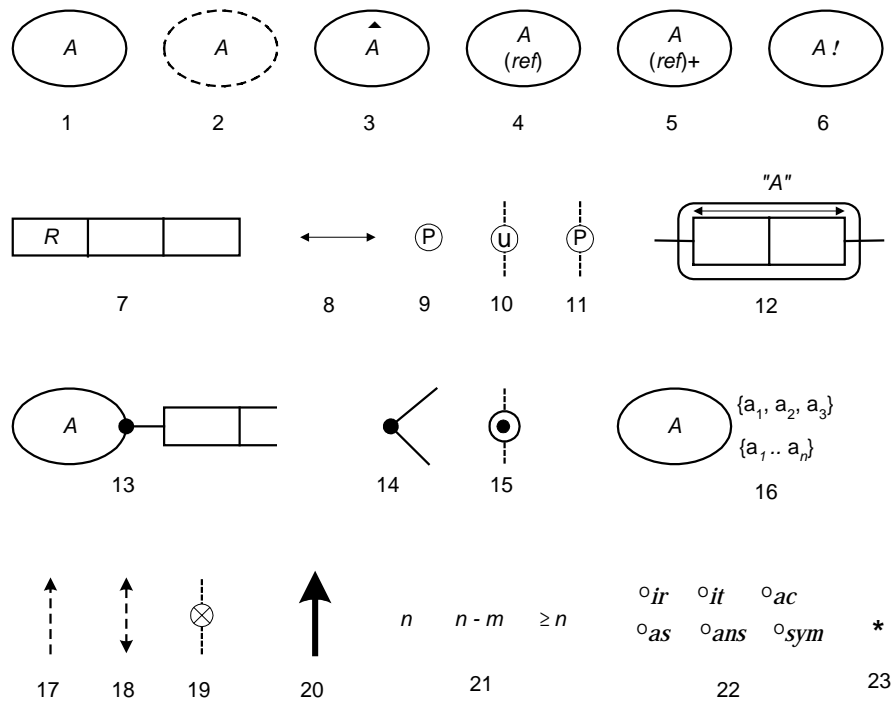


Figure 1 Main ORM symbols

The symbols are numbered for easy reference. An *entity type* is depicted as a named ellipse (symbol 1). A *value type* denotes a lexical object type (e.g. a character string or number) and is usually shown as a named, dotted ellipse (symbol 2). Another notation for value types encloses the value type name in parentheses. Object types that appear more than once in the schema may be tagged with an arrow tip (see symbol 3), that “points” to the existence of another occurrence. Each entity type must have at least one *reference scheme*, which indicates how each instance of the entity type may be mapped via predicates to a combination of one or more values. A simple injective (1:1 into) reference scheme maps entities to single values. For example, each country may be identified by a single country code (e.g. ‘USA’). In such cases the reference scheme may be abbreviated as in symbol 4 by displaying the *reference mode* in parentheses beside the name of the entity type, e.g. Country(code). The reference mode indicates how values relate to the entities. Symbol 5 shows that a plus sign “+” may be added if the values are numeric, e.g. Mass(kg)+. Values are constants with a universally understood denotation, and hence require no reference scheme.

Although not strictly a conceptual issue, it is normal to require each entity type to have a *primary* reference scheme. Relationship types used for primary reference are then called *reference types*. The other relationship types are known as *fact types*. In symbol 6, an exclamation mark is added to declare that an entity type is *independent*. This means that instances of that type may exist without participating in any facts. By default, this is not the case (i.e. we don’t normally introduce an object into the universe unless it takes part in some fact).

Symbol 7 shows a ternary *predicate*, comprised of three *roles*. Each role is depicted as a box, and must be played by exactly one object type. Roles are connected to their players by a line segment (see symbol 13). A predicate is basically a sentence with object holes in it, one for each role. The number of roles is called the *arity* of the predicate. Except for the BRM version, ORM allows predicates of any arity (1 = unary, 2 = binary, 3 = ternary etc.). Predicates are usually treated as ordered, as in predicate logic. In this case, the name of the predicate is written either in or beside the first role box, and if necessary each object hole may be shown as an ellipsis “...”. Different readings may be provided so the information may be read in any direction. FORML allows mixfix predicates so objects may be placed at any position in the predicate. For example, the fact type Room at Time is used for Activity involves the predicate “... at ... is used for ...”. Apart from facilitating natural verbalization of n-ary relationships, mixfix predicates allow binary relationships to be verbalized in languages where the verb is not in the infix position (e.g. in Japanese, verbs come at the end). In some versions of ORM, relationship types are given a name, and each role is also given a name, thus making order irrelevant.

Internal uniqueness constraints are depicted as arrow tipped bars (symbol 8), and are placed over one or more roles in a predicate to declare that instances for that role (combination) in the relationship type population must be unique. For example, adding a uniqueness constraint over the first role of Person was born in Country declares that each person was born in at most one country. A predicate may have one or more uniqueness constraints, at most one of which may be declared *primary* by adding a “P” (symbol 9). An *external uniqueness constraint* shown as a circled “u” may be applied to two or more roles from different predicates by connecting to them with dotted lines (symbol 10). This indicates that instances of the combination of those roles in the join of those predicates are unique. For example, to say that a state is identified by combining its statecode and country, we add an external uniqueness constraint to the roles played by Statecode and Country in the reference types: State has Statecode; State is in Country. To declare an external uniqueness constraint primary, use “P” instead of “u” (symbol 11). An object type may have at most one primary reference constraint.

If we want to talk about a relationship type we may *objectify* it (i.e. make an object out of it) so that it can play roles. Graphically, the objectified predicate is enclosed in either a rounded rectangle (symbol 12) or an ellipse, and named. Objectified predicates are also called *nested* object types. Typically the objectified predicate must have a spanning uniqueness constraint, but 1:1 cases may also be allowed [Hal93].

A *mandatory role constraint* declares that every instance in the population of the role’s object type must play that role. It is usually shown as a black dot (see symbol 13) but a universal quantifier is sometimes used. Mandatory roles are also called *total* roles. A *disjunctive* mandatory constraint may be applied to two or more roles to indicate that all instances of the object type population must play *at least one* of those roles. This may often be shown by connecting the roles to a black dot on the object type (symbol 14) or in general by connecting the roles by dotted lines to a circled black dot (symbol 15).

To restrict an object type’s population to a given list, the relevant values may be listed in braces (symbol 16, top). If the values are ordered, a range may be declared separating the first and last values by “..” (symbol 16, bottom). These constraints are called *value constraints*.

Symbols 17-19 denote *set comparison constraints*, and may only be applied between compatible role sequences (i.e. sequences of one or more roles, where the corresponding roles have the same host object type). A dotted arrow (symbol 17) from one role sequence to another is a *subset constraint*, restricting the population of the first sequence to be a subset of the second. A double-tipped arrow (symbol 18) is an *equality constraint*, indicating the populations must be equal. A circled “X” (symbol 19) is an *exclusion constraint*, indicating the populations are mutually exclusive. Exclusion constraints may be applied between two or more sequences.

A solid arrow (symbol 20) from one object type to another indicates that the first object type is a (proper) *subtype* of the other. For example, Woman is a subtype of Person. Totality (circled black dot) and exclusion (circled “X”) constraints may also be displayed between subtypes, but are implied by other constraints if the subtypes are given formal definitions.

Symbol 21 shows three kinds of *frequency constraint*. Applied to a sequence of one or more roles, these indicate that instances that play those roles must do so exactly n times, between n and m times, or at least n times.

Symbol 22 shows six kinds of *ring constraint*, that may be applied to a pair of roles played by the same host type. These indicate that the binary relation formed by the role population must be irreflexive (ir), intransitive (it), acyclic (ac), asymmetric (as), antisymmetric (ans) or symmetric (sym).

Symbol 23 is an asterisk “*”, which may be placed beside a fact type to indicate that it is derivable from other fact types. Not all versions of ORM support all these symbols, and some versions have a few more symbols. InfoModeler, a popular ORM tool, supports all of the symbols shown, as will a future release of Visio Professional.

1.4 Conceptual schema design procedure

The information systems life cycle typically involves several stages: feasibility study; requirements analysis; conceptual design of data and operations; logical design; external design; prototyping; internal design and implementation; testing and validation; and maintenance. ORM’s *conceptual schema design procedure* (CSDP) focuses on the analysis and design of data. The conceptual schema specifies the information structure of the application: the *types of fact* that are of interest; *constraints* on these; and perhaps *derivation rules* for deriving some facts from others. With large applications, the UoD is divided into convenient modules, the CSDP is applied to each, and the resulting subschemas are integrated into the global conceptual schema.

Table 1 shows the CSDP used in FORM. Although different versions of the CSDP exist, they all agree on the importance of verbalization in terms of elementary facts, population checks, and thorough analysis of business rules. The rest of this section illustrates the basic working of this design procedure by means of an example. Because of space limitations, our treatment is necessarily brief. A much more detailed discussion of the same example can be electronically accessed from [Hal97].

Table 1 The conceptual schema design procedure (CSDP)

Step

1. Transform familiar information examples into elementary facts, and apply quality checks.
 2. Draw the fact types, and apply a population check.
 3. Check for entity types that should be combined, and note any arithmetic derivations.
 4. Add uniqueness constraints, and check arity of fact types.
 5. Add mandatory role constraints, and check for logical derivations.
 6. Add value, set comparison and subtyping constraints.
 7. Add other constraints and perform final checks.
-

Step 1 is the most important. Examples of the information required from the system are verbalized in natural language. Such examples are often available in the form of output reports or input forms, perhaps from a current manual version of the required system. If not, the modeler can work with the client to produce examples. To avoid misinterpretation, a UoD expert (a person familiar with the application) should perform or at least check the verbalization. As an aid to this process, the speaker imagines he/she has to convey the information contained in the examples to a friend over the telephone.

For our case study, we consider a fragment of an information system used by a university to maintain details about its academic staff and academic departments. One function of the system is to print an academic staff directory, as exemplified by the report extract shown in Table 2. Part of the modeling task is to clarify the meaning of terms used in such reports. The descriptive narrative provided here would thus normally be derived from a discussion with the UoD expert. The terms “empnr” and “extnr” abbreviate “employee number” and “extension number”.

A phone extension may have access to local calls only (“LOC”), national calls (“NAT”), or international calls (“INT”). International access includes national access, which includes local access. In the few cases where different rooms or staff have the same extension, the access level is the same. An academic is either tenured or on contract. Tenure guarantees employment until retirement, while contracts have an expiry date.

Table 2 Extract from a directory of academic staff

<i>Empnr</i>	<i>EmpName</i>	<i>Dept</i>	<i>Room</i>	<i>Phone</i>		<i>Tenured/</i>
				<i>Extnr</i>	<i>Access</i>	<i>Contract-expiry</i>
715	Adams A	Computer Science	69-301	2345	LOC	01/31/95
720	Brown T	Biochemistry	62-406	9642	LOC	01/31/95
139	Cantor G	Mathematics	67-301	1221	INT	tenured
430	Codd EF	Computer Science	69-507	2911	INT	tenured
503	Hagar TA	Computer Science	69-507	2988	LOC	tenured
651	Jones E	Biochemistry	69-803	5003	LOC	12/31/96
770	Jones E	Mathematics	67-404	1946	LOC	12/31/95
112	Locke J	Philosophy	1-205	6600	INT	tenured
223	Mifune K	Elec. Engineering	50-215A	1111	LOC	tenured
951	Murphy B	Elec. Engineering	45-B19	2301	LOC	01/03/95
333	Russell B	Philosophy	1-206	6600	INT	tenured
654	Wirth N	Computer Science	69-603	4321	INT	tenured
...

The information contained in this Table is to be stated in terms of *elementary facts*. Basically, an elementary fact asserts that a particular object has a property, or that one or more objects participate in a relationship, where that relationship cannot be expressed as a conjunction of simpler (or shorter) facts without introducing new object types [Hal93]. For example, to say that Bill Clinton jogs and is the president of the USA is to assert two elementary facts.

As a first attempt, one might read off the information on the first data row as the six facts f1-f6. Each asserts a binary relationship between two objects. For discussion purposes the *predicate* is shown in **bold** between the noun phrases that identify the objects, and object type names start with a capital letter. Some obvious abbreviations are used (“empnr”, “EmpName”, “Dept”, “extnr”); when read aloud these can be expanded to “employee number”, “Employee name”, “Department” and “extension number”. The second data row contains different instances of these six fact types. Row three, because of its final column, provides an instance f7 of a seventh fact type, a unary.

- f1 The Academic with empnr 715 **has** EmpName ‘Adams A’.
- f2 The Academic with empnr 715 **works for** the Dept named ‘Computer Science’.
- f3 The Academic with empnr 715 **occupies** the Room with roomnr ‘69-301’.
- f4 The Academic with empnr 715 **uses** the Extension with extnr ‘2345’.
- f5 The Extension with extnr ‘2345’ **provides** the AccessLevel with code ‘LOC’.
- f6 The Academic with empnr 715 **is contracted till** the Date with mdy-code ‘01/31/95’.
- f7 The Academic with empnr 139 **is tenured**.

Different readings may be provided to allow relationships to be read in different directions. For example, the *inverse* reading of f4 is: The Extension with extnr ‘2345’ **is used by** the Academic with empnr 715. To save writing, both the normal predicate and its inverse may be included in the same declaration, with the inverse predicate preceded by a slash “/”. For example:

- f4’ The Academic with empnr 715 **uses /is used by** the Extension with extnr ‘2345’.

Predicate names are usually unique in the conceptual schema. In some cases (e.g. “has”), the same name may be used externally for different predicates: internally these have different identifiers.

As a quality check at Step 1, we ensure that objects are well identified. Values are identified by constants (e.g. ‘Adams A’, 715). *Entities* are “real world” objects that are identified by a definite description (e.g. the Academic with empnr 715). Fact f1 involves a relationship between an entity (a person) and a value (a name is just a character string). Facts f2-f6 specify relationships between entities. Fact f7 states a property (or unary relationship) of an entity.

As a second quality check at Step 1, we use our familiarity with the UoD to see if some facts should be split or recombined (a formal check on this is applied later). For example, suppose facts f1 and f2 were verbalized as: The Academic with empnr 715 and empname ‘Adams A’ **works for** the Dept named ‘Computer Science’. The presence of the word “and” suggests that this may be split without information loss. The repetition of “Jones E” on different rows of Table 2 shows that academics cannot be identified just by their name. However the uniqueness of empnr in the sample population suggests that this suffices for reference. Since the “and-test” is only a heuristic, and sometimes a composite naming scheme is required for identification, the UoD expert is consulted to verify that empnr by itself is sufficient for identification. With this assurance obtained, the composite sentence is now split into f1 and f2.

As an alternative to specifying complete facts one at a time, the reference schemes may be declared up front and then assumed in later facts. For example, suppose we declare the following: Academic(empnr); EmpName(); Dept(name). The empty parentheses after EmpName indicates it is a value type and hence needs no reference scheme. Now facts f1 and f2 may be stated as: Academic 715 has EmpName ‘Adams A’; Academic 715 works for Dept ‘Computer Science’. Facts f1-f7 are instances of the following fact types:

- F1 Academic has EmpName
- F2 Academic works for Dept
- F3 Academic occupies Room
- F4 Academic uses Extension
- F5 Extension provides AccessLevel
- F6 Academic is contracted till Date
- F7 Academic is tenured

Step 2 of the CSDP is to *draw a draft diagram of the fact types and apply a population check* (see Figure 2). As a check, each fact type has been populated with at least one fact, shown as a row of entries in the associated fact table, using the data from rows 1 and 3 of Table 2. The English sentences listed before as facts f1-f7, as well as other facts from row 3, may be read directly off this figure. Though useful for validating the model with the client and for understanding constraints, the sample population is not part of the conceptual schema itself.

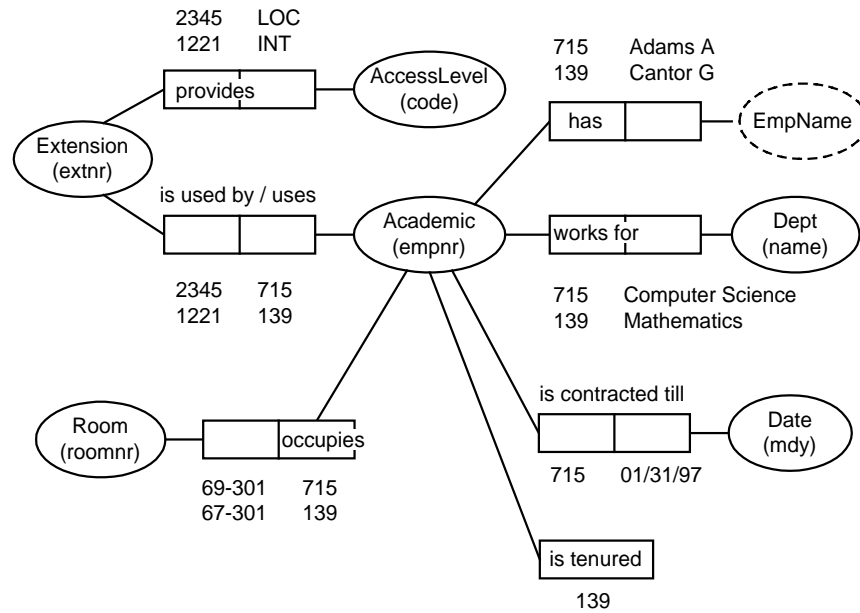


Figure 2 Draft diagram of fact types for Table 2 with sample population

Suppose the information system is also required to assist in the production of departmental handbooks. Figure 3 shows an extract from a page of one such handbook. In this university academic staff are classified as professors, senior lecturers or lecturers, and each professor holds a “chair” in a research area. To reduce the size of our problem, we have excluded many details that in practice would also be recorded (e.g. office phone and fax). To save space, details are shown here for only four of the 22 academics in that department. The data are, of course, fictitious.

Department:	Computer Science	<i>Home phone of Dept head: 9765432</i>
<i>Chairs</i>	<i>Professors (5)</i>	
Databases	Codd EF	BSc (UQ); PhD (UCLA) (<i>Head of Dept</i>)
Algorithms	Wirth N	BSc (UQ); MSc (ANU); DSc (MIT)
...		
<i>Senior Lecturers (9)</i>		
Hagar TA	BInfTech (UQ); PhD (UQ)	
...		
<i>Lecturers (8)</i>		
Adams A	MSc (OXON)	
...		

Figure 3 Extract from Handbook of Computer Science Department

It appears from the handbook example that within a single department, academics may be identified by their name. Let us assume this is verified by the UoD expert. However the complete application requires us to handle all departments in the same information system, and to integrate this subschema with the directory subschema considered earlier. Hence we must replace the academic naming convention used for the handbook example by the global scheme used earlier (i.e. empnr).

We use this report to illustrate *Step 3* of the CSDP: *check for entity types that should be combined, and note any arithmetic derivations*. Suppose we verbalized the degree information in terms of the three

ternary fact types: Professor obtained Degree from University; SeniorLecturer obtained Degree from University; Lecturer obtained Degree from University. The common predicate suggests that the entity types Professor, SeniorLecturer and Lecturer should be collapsed to the single entity type Academic, with this predicate now shown only once. To preserve the original information about who is a professor, senior lecturer or lecturer we introduce the fact type: Academic has Rank. Let's use the codes "P", "SL" and "L" for the ranks of professor, senior lecturer and lecturer.

The second aspect of Step 3 is to see if some fact types can be derived from others by arithmetic. Since we now record the rank of academics as well as their departments, we can compute the number in each rank in each department simply by counting. So the fact type Dept employs academics of Rank in Quantity is *derivable*. If desired, derived fact types may be included on a schema diagram if they are marked with an asterisk "*". At any rate, a *derivation rule* must be supplied. This may be written below the diagram (see Figure 4). Here "iff" abbreviates "if and only if".

Step 4 of the CSDP is to *add uniqueness constraints and check the arity of the fact types*. For example, we add a uniqueness constraint to the first role of works for to ensure each academic works for at most one department. An arity check ensures each uniqueness constraint on an n-ary spans at least n-1 roles.

Step 5 of the CSDP is to *add mandatory role constraints, and check for logical derivations*. For example, we need a disjunctive mandatory constraint to declare that each academic *either* is contracted till some date *or* is tenured. Roles that are not mandatory are *optional*. If an object type plays only one fact role in the global schema, then by default this is mandatory, but a dot is not normally shown.

Suppose that departmental handbooks include a building directory, which lists the names as well as the numbers of buildings. A sample fact might be: Building '67' has Buildingname 'Priestly'. Earlier we identified rooms by a single value. For example "67-301" was used to denote the room in building 67 which has room number "301". Now that buildings are to be talked about in their own right, we replace the simple reference scheme by a composite one that shows the full semantics (see Figure 4). Here Roomnr now means just the number (e.g. "301") used to identify the room within its building.

To illustrate nesting, suppose the application requires reports about teaching commitments, an extract of which is shown in Table 3. Not all academics currently teach. If they do, their teaching in one or more subjects may be evaluated and given a rating. Some teachers serve on course curriculum committees. Here the new fact types may be schematized as shown in Figure 4. The nested object type Teaching plays only one role, and this role is optional, so Teaching is an *independent* object type (as shown by the "!").

Table 3 Extract of report on teaching commitments

<i>Empnr</i>	<i>Emp. name</i>	<i>Subject</i>	<i>Rating</i>	<i>Committees</i>
715	Adams A	CS100 CS101	5	
430	Codd EF			
654	Wirth N	CS300		BSc-Hons CAL Advisory

The second stage of Step 5 is to check for *logical derivations* (i.e. can some fact type be derived from others without the use of arithmetic?). One strategy here is to ask whether there are any relationships (especially functional relationships) which are of interest but which have been omitted so far. Another strategy is to look for transitive patterns of functional dependencies. Suppose that our client confirms that the rank of an academic determines the access level of his/her extension. For example, suppose a current business rule is that professors get international access while lecturers and senior lecturers get local access. This rule might change in time (e.g. senior lecturers might be arguing for national access). To minimize later changes to the schema, we store the rule as data in a table. So it can be updated as required by an authorized user without recompiling the schema. The relevant rule is shown at the bottom of Figure 4.

In *Step 6* of the CSDP we add *any value, set comparison and subtyping constraints*. One value constraint is that Rankcode is restricted to { 'P', 'SL', 'L' }. In Figure 4, a pair-subset constraint runs from the heads predicate to the works for predicate, indicating that a person who heads a department must work for the same department. The rule that nobody can be tenured and contracted at the same time is captured by an exclusion constraint. Subtyping is determined as follows. Each optional role is inspected: if the role is played only by some well-defined subtype, a subtype node is introduced with this role attached.

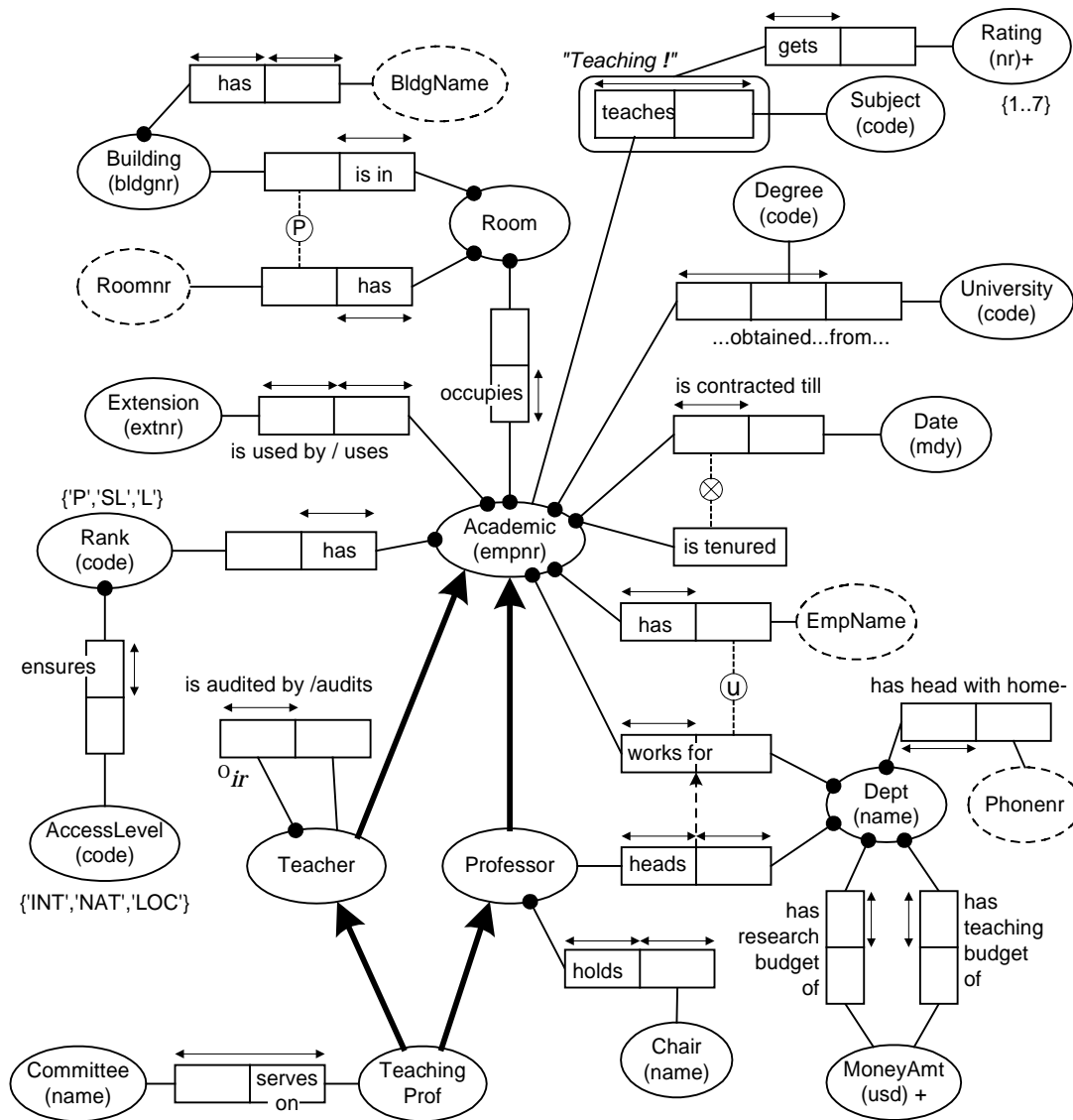


Figure 4 The final conceptual schema

Subtype links and definitions are added. Figure 4 contains three subtypes: Teacher; Professor; and TeachingProfessor. In this university, each teacher is audited by another teacher. Moreover, only professors may be department heads, and only teaching professors can serve on curriculum committees (not all universities work this way).

Step 7 of the CSDP adds other constraints and performs final checks. For example, auditing is *irreflexive* (no teacher audits himself/herself). Suppose we also need to record the teaching and research budgets of the departments. We might schematize this using the ternary Dept has for Activity a budget of MoneyAmt, where Activity has the value constraint {'Teaching', 'Research'} and the first role is mandatory and constrained to a *frequency* of 2.

Once the global schema is drafted, and the target DBMS decided, some optimization can often be performed to improve the efficiency of the logical schema obtained by mapping. Assuming the conceptual schema is to be mapped to a relational database schema, the ternary fact type about budgets will map to a separate table all by itself, leading to extra joins for some queries. We can avoid this problem by transforming the ternary into the following two binaries before we map: Dept has teaching budget of MoneyAmt; Dept has research budget of MoneyAmt. These binaries have simple keys, and will map to the “main” department table. Another optimization may be performed which moves the home phone information to Dept instead of Professor. Figure 4 includes these optimizations. Such *conceptual schema transformations* require a rigorous theory of schema equivalence and optimization strategies. For details on such topics, see [Hal95, ch. 9; HP95b; DeT93].

2.3 Logical Mapping

Once the conceptual schema has been specified, the target data model is selected and the mapping is done. For example, the Rmap algorithm [RH93; Hal95] maps our conceptual schema to the relational schema shown in Figure 5 (domains omitted). If the conceptual fact types are elementary (as they should be), then the mapping is guaranteed to be free of redundancy, since each fact type is grouped into only one table, and fact types which map to the same table all have uniqueness constraints based on the same attribute(s).

Keys are underlined. If alternate keys exist, the primary key is doubly-underlined. A mandatory role is captured by making its corresponding attribute mandatory in its table (not null is assumed by default), by marking as optional (in square brackets) all optional roles for the same object type which map to the same table, and by running an equality/subset constraint from those mandatory/optional roles which map to another table. The $\langle 2,1 \rangle$ in the pair-subset constraint indicates the source pair should be reversed before the comparison. Subtyping is captured by qualified optionals or qualified subset constraints. The word “exists” means “a non-null value exists”.

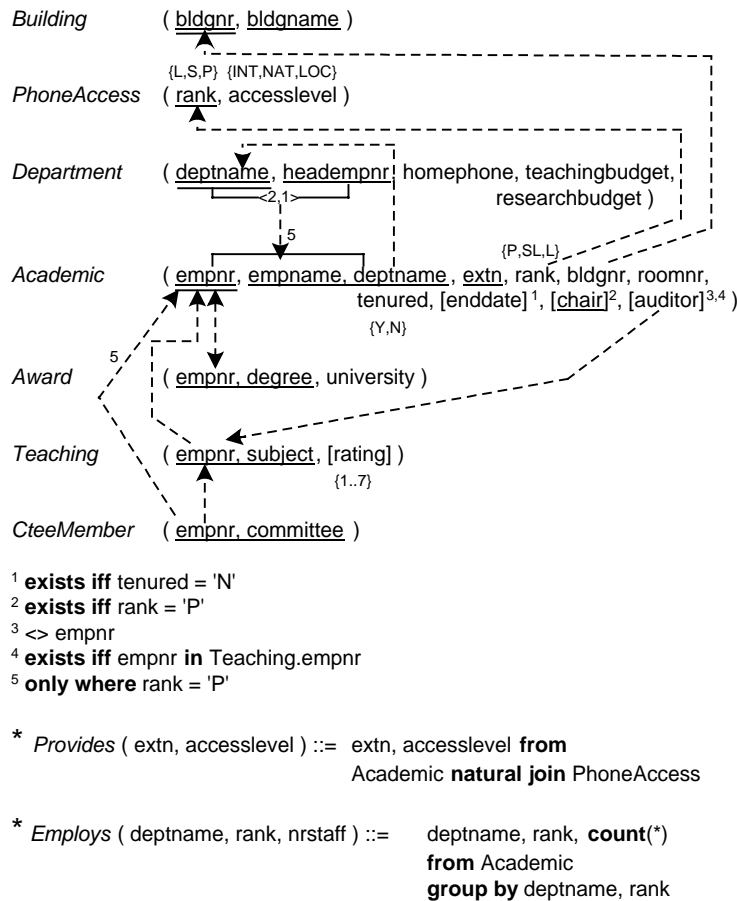


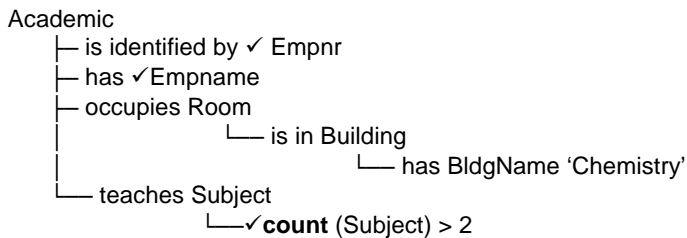
Figure 5 The relational schema mapped from Figure 4

3 Recent extensions

3.1 Conceptual queries

Besides information modeling, ORM is also ideal for information querying. The first significant ORM query language was RIDL [Mee82], a hybrid language with both declarative and procedural components. Temporal aspects were added later to form TRIDL. Currently, research is being carried out on at least three ORM query languages: LISA-D [HPW93]; OSM-QL [EWPC96]; and ConQuer [BH96]. Of these ConQuer (CONceptual QUERy) is the only one to be commercially released. A more powerful version, ConQuer-II [BH97], is currently under development at Visio Corporation.

Using ConQuer, an ORM model may be queried directly without prior knowledge of either the conceptual schema or the corresponding relational schema, by dragging object types onto the query pane, selecting predicates of interest, applying restrictions and functions as desired, and ticking the items to be listed. As a simple example, consider the following English query on our academic database: list the empnr, empname and number of subjects taught for each academic who occupies a room in the Chemistry building and teaches more than two subjects. This may be formulated by drag-and-drop basically as follows:



Notice how easily the conceptual joins are made. A verbalization of the query is automatically generated, as well as SQL code. Formulating queries in terms of objects and predicates is much easier than deciphering the semantics of the relational schema and coding in SQL or QBE. A major benefit of such queries is their semantic stability. For example, ConQuer queries are unaffected by most schema changes (e.g. addition of fact types, or changes to constraints). In contrast, such changes often require the corresponding SQL or ER query to be reformulated, since they depend on attribute structures.

3.2 Other extensions

Researchers are actively investigating several extensions to the basic ORM framework. These include abstraction mechanisms to allow users to control the amount of detail seen at any given time [CHP96], reverse engineering [SS93; CH94], support for complex objects [HW93; DM95], process-event modeling [Hof93], external schema generation [CH93], schema evolution [Pro94], schema optimization [HP95b; Bom94], meta-modeling [FO94], subtype extensions [HP95a], null handling [HR92], object-oriented mapping [ME96], unary nesting [BZL94], and empirical research [Eve94].

Although various versions of ORM have added support for complex objects, they differ in their approaches. Currently there seems to be a growing agreement that constructors (e.g. set, bag, sequence) should only be added after a “flat” ORM model is first developed. There are also different opinions on whether such constructors should be considered part of the conceptual model, or regarded as mapping annotations. Commercial developers of ORM tools are also extending the method. For example, InfoModeler includes extra constructs for mapping to object-relational databases, and extensions of this technology are being incorporated into future Visio products.

4 Conclusion

This article has provided only a brief sketch of the ORM method, emphasizing its fundamental features and touching on some of its advantages. Apart from its sound theoretical basis, the method has been used successfully in many countries, on applications from the small to the very large. The recent emergence of intuitive and powerful ORM tools has led to wider adoption of the method, which is now being successfully taught as early as high school level. Perhaps the greatest strengths of ORM are that it lifts the communication between modeler and client to a level where they can readily understand and validate the

application model using simple sentences, and that it has been designed from the ground up to facilitate schema evolution. This second advantage is very relevant to today's business world where change is ongoing.

In an article this brief, several aspects of ORM have necessarily been glossed over. The reader who is interested in pursuing the area further should consult the cited references, which are included at the end of the handbook.

References

- [Abr74] Abrial, J.R. 1974, 'Data Semantics', *Data Base Management*, eds J.W. Klimbie and K.L. Koffeman, North-Holland, Amsterdam, The Netherlands, pp. 1-60.
- [BZL94] Bakema, G.P., Zwart, J.P.C. & Lek, H. van der 1994, 'Fully Communication Oriented NIAM', *NIAM-ISDM 1994 Conf. Working papers*, eds G.M. Nijssen & J. Sharp, Albuquerque, NM USA, pp. L1-35.
- [BH96] Bloesch, A.C. & Halpin, T.A. 1996, 'ConQuer: a conceptual query language', *Proc. ER'96: 15th Int. Conf. on conceptual modeling*, Springer LNCS, vol. 1157, pp. 121-33.
- [BH97] Bloesch, A.C. & Halpin, T.A. 1997, 'Conceptual queries using ConQuer-II', *Proc. ER'97: 16th Int. Conf. on conceptual modeling*, Springer LNCS, vol. 1331, pp. 113-26.
- [Bom94] Bommell, P. van 1994, 'Implementation selection for Object-Role models', *Proc. First Int. Conf. On Object-Role Modeling (ORM-1)*, eds T.A. Halpin & R.M. Meersman, Magnetic Island, Australia, pp. 103-12.
- [CH93] Campbell, L. & Halpin, T.A. 1993, 'Automated Support for Conceptual to External Mapping', *Proc. 4th Workshop on Next Generation CASE Tools*, eds S. Brinkkemper & F. Harmsen, Univ. Twente Memoranda Informatica 93-32, pp. 35-51, Paris (June).
- [CH94] Campbell, L. & Halpin, T.A. 1994, 'The reverse engineering of relational databases', *Proc. 5th Workshop on Next Generation CASE Tools*, Utrecht (June).
- [CHP96] Campbell, L.J., Halpin, T.A. & Proper, H.A. 1996 'Conceptual Schemas with Abstractions: making flat conceptual schemas more comprehensible', *Data and Knowledge Engineering*, vol. 20, no. 1, pp. 39-85.
- [DeT93] De Troyer, O. 1993, 'On data schema transformations', PhD thesis, University of Tilburg (K.U.B.), Tilburg, The Netherlands.
- [DM95] 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, vol. 1021, pp. 238-49.
- [EKW92] Embley, D.W., Kurtz, B.D. & Woodfield, S.N. 1992, *Object-Oriented Systems Analysis*, Prentice Hall, Englewood Cliffs, NJ.
- [EWPC96] Embley, D.W., Wu, H.A., Pinkston, J.S. & Czejdo, B. 1996, 'OSM-QL: a calculus-based graphical query language', Tech. Report, Dept of Comp. Science, Brigham Young Univ., Utah.
- [Eve94] Everest, G. 1994, 'Experiences teaching NIAM/OR modeling', *NIAM-ISDM 1994 Conf. Working Papers*, eds G.M. Nijssen & J. Sharp, Albuquerque, NM USA, pp. N1-26.
- [Fal76] Falkenberg, E.D. 1976, 'Concepts for modelling information', *Proc. 1976 IFIP Working Conf. on Modelling in Data Base Management Systems*, ed. G.M. Nijssen, Freudenstadt, Germany, North-Holland Publishing, pp. 95-109
- [FO94] Falkenberg, E.D. & Oei, J.L.H. 1994, 'Meta-model hierarchies from an Object-Role Modeling perspective', *Proc. First Int. Conf. On Object-Role Modeling (ORM-1)*, eds T.A. Halpin & R.M. Meersman, Magnetic Island, Australia, pp. 218-227.
- [Hab93] Habrias, H. 1993, 'Normalized Object Oriented Method', in *Encyclopedia of Microcomputers*, vol. 12, Marcel Dekker, New York, pp. 271-85.
- [Hal89] Halpin, T.A. 1989, 'A Logical Analysis of Information Systems: static aspects of the data-oriented perspective', PhD thesis, University of Queensland.
- [Hal93] Halpin, T.A. 1993, 'What is an elementary fact?', *Proc. First NIAM-ISDM Conf.*, eds G.M. Nijssen & J. Sharp, Utrecht, (Sep), 11 pp.
- [Hal95] Halpin, T.A. 1995, *Conceptual Schema and Relational Database Design*, 2nd edn, Prentice Hall Australia, Sydney.
- [Hal96] Halpin, T.A. 1996, 'Business Rules and Object-Role Modeling', *Database Prog. & Design*, vol. 9, no. 10, Miller Freeman, San Mateo CA, pp. 66-72.
- [Hal97] Halpin, T.A. 1997, 'Object-Role Modeling: an overview', electronic paper available on website www.orm.net.

- [HP95a] Halpin, T.A. & Proper, H.A. 1995a, 'Subtyping and polymorphism in Object-Role Modeling', *Data and Knowledge Engineering*, vol. 15, pp. 251-81, Elsevier Science.
- [HP95b] Halpin, T.A. & Proper, H.A. 1995b, 'Database schema transformation and optimization', *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, vol. 1021, pp. 191-203.
- [HR92] Halpin, T.A. & Ritson, P.R. 1992, 'Fact-Oriented Modelling and Null Values', *Proc. 3rd Australian Database Conf.*, eds. B. Srinivasan & J. Zeleznikov, World Scientific, Singapore.
- [Hof93] Hofstede, A.H.M. ter 1993, 'Information modelling in data intensive domains', PhD thesis, University of Nijmegen, The Netherlands.
- [HPW93] Hofstede, A.H.M. ter, Proper, H.A. & Weide, th.P. van der 1993, 'Formal definition of a conceptual language for the description and manipulation of information models', *Information Systems*, vol. 18, no. 7, pp. 489-523.
- [HW93] Hofstede A.H.M. ter & Weide, th.P. van der 1993, 'Expressiveness in conceptual data modelling', *Data and Knowledge Engineering*, vol. 10, no. 1, pp. 65-100.
- [Ken77] Kent, W. 1977, 'Entities and relationships in Information', *Proc. 1977 IFIP Working Conf. on Modelling in Data Base Management Systems*, ed. G.M. Nijssen, Nice, France, North-Holland Publishing, pp. 67-91.
- [Mee82] Meersman, R. 1982, 'The RIDL conceptual language', Research report, Int. Centre for Information Analysis Services, Control Data Belgium, Brussels.
- [ME96] Mok, W.Y & Embley, D.W. 1996, 'Transforming conceptual model to object-oriented database designs: practicalities, properties and peculiarities', *Proc. ER'96: 15th Int. Conf. on conceptual modeling*, Springer LNCS, vol. 1157, pp. 309-24.
- [Nij76] Nijssen, G.M. 1976, 'A gross architecture for the next generation database management systems', *Proc. 1976 IFIP Working Conf. on Modelling in Data Base Management Systems*, ed. G.M. Nijssen, Freudenstadt, Germany, North-Holland Publishing, pp. 1-24.
- [Nij77] Nijssen, G.M. 1977, 'Current issues in conceptual schema concepts', *Proc. 1977 IFIP Working Conf. on Modelling in Data Base Management Systems*, ed. G.M. Nijssen, Nice, France, North-Holland Publishing, pp. 31-66.
- [Pro94] Proper, H.A. 1994, 'A theory of conceptual modelling of evolving application domains', PhD thesis, University of Nijmegen, The Netherlands.
- [RH93] Ritson, P.R. & Halpin, T.A. 1993, 'Mapping Integrity Constraints to a Relational Schema', *Proc. 4th ACIS*, Brisbane (Sep.), pp. 381-400.
- [Sen75] Senko, M.E. 1975, 'Information systems: records, relations, sets, entities and things', *Information Systems*, vol. 1, no. 1, Jan. 1995, Pergamon Press, pp. 3-13.
- [SS93] Shoval, P. & Shreiber, N. 1993, 'Database reverse engineering: from the relational to the binary relational model', *Data and Knowledge Engineering*, vol. 10, pp. 293-315.
- [VB82] Verheijen, G.M.A. & van Bekkum, J. 1982, 'NIAM: an information analysis method', *Information systems Design Methodologies: a comparative review, Proc. IFIP WG8.1 Working Conf.*, Noordwijkerhout, The Netherlands, North Holland Publishing, pp. 537-90.
- [Ver83] Vermeir, D. 1983, 'Semantic hierarchies and abstractions in conceptual schemata', *Information systems*, vol. 8, no. 2, pp. 117-24.
- [Win90] Wintraecken, J.J.V.R. 1990, *The NIAM Information Analysis Method: Theory and Practice*, Kluwer, Deventer, The Netherlands.

Entity Relationship modeling from an ORM perspective: Part 1

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

This is a revised version of a paper that first appeared in the December 1999 issue of the Journal of Conceptual Modeling, published by InConcept.

This paper is the first in a series of articles examining data modeling in the Entity Relationship (ER) approach from the perspective of Object Role Modeling (ORM). This article examines basic aspects of the Barker notation for ER.

Introduction

Entity Relationship modeling (ER) views the application domain in terms of entities that have attributes and participate in relationships. For example, the fact that an employee was born on a date is modeled by assigning a birthdate attribute to the Employee entity type, whereas the fact that an employee works for a department is modeled as a relationship between them. This view of the world is quite intuitive, and in spite of the recent rise of UML for modeling object-oriented applications, ER is still the most popular data modeling approach for database applications.

The ER approach was originally proposed by Peter Chen in 1976, in the very first issue of an influential ACM journal [2]. As shown in Figure 1, Chen's original notation used rectangles for entity types, diamonds for relationships, and ellipses for attributes. The double ellipse indicates unique identifier attributes, and the "n" and "1" indicate the relationship is many to one (each employee works for at most one department, but many employees may work for the same department).

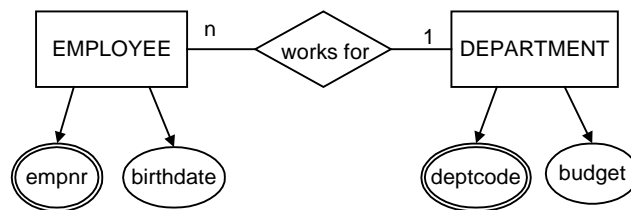


Figure 1 An early ER notation used by Chen

The direction in which relationship names are to be read is formally undecided, unless we add some additional marks (e.g. arrows) or rules (e.g. always read from left to right and from top to bottom). For example, does the employee work for the department, or does the department work for the employee? Although we can use our background knowledge to informally disambiguate this example, it is quite common nowadays to see ER models with relationships whose intended direction can only be guessed at by anybody other than the model's creator. For example, consider the impact of misreading the intended direction for the following: Person killed Animal; Person is loved by Person. This problem is exacerbated if the verb phrase used to name the relationship is shortened to one word (e.g. "work", "love"), unfortunately still a fairly common practice.

Chen's notation evolved over time. His current ER-Designer tool uses hexagons instead of diamonds, and supports n-ary relationships. Outside academia, Chen's notation seems to be rarely used nowadays, so I'll say no more about it here. One of the problems with the ER approach is that there are so many versions of it, with no single standard. In industrial practice, the most popular versions of ER are the Barker and Information Engineering (IE) notations. Another popular data modeling notation is IDEF1X, but since this is a hybrid of ER and relational notation, I don't count it as a true ER representative. As discussed elsewhere [4], UML class diagrams can be regarded as an extended version of ER. The rest of this article focuses on basic aspects of the Barker notation for ER. Later articles will examine IE and IDEF1X.

Barker ER: the basics

I use the term "Barker ER" for the ER notation discussed in the classic treatment by Richard Barker [1]. While Oracle Corporation has long used this notation in its CASE tools, Oracle's Object Designer tool now supports UML as an alternative to its traditional ER notation. For database applications, many modelers still prefer the Barker ER notation in preference to UML, and it will be interesting to see whether this changes over time. Dave Hay, an experienced modeler and ardent fan of the Barker ER notation, argues that "there is no such thing as 'object-oriented analysis'" [6], only object-oriented design, and that "UML is ... not suitable for analyzing business requirements in cooperation with business people" [7].

While I agree with Dave Hay that UML class diagrams are less than ideal for data modeling, I feel that his preferred ER notation shares some of UML's weaknesses in being attribute-based. As I've discussed before in a UML context [4, 5], using attributes in a base conceptual model adds complexity and instability, while making it harder to validate models with domain experts using verbalization and sample populations. Attributes are great for logical design, since they allow compact diagrams that directly represent the data structures (e.g. relations or object-relations) used for the actual design. However when I'm performing conceptual analysis, I just want to know what the *facts* and *rules* are about the business, and I want to communicate this information in sentences, so that the

model can be understood by the domain experts. I sure don't want to bother about how facts are grouped into multi-fact structures. Whether some fact will end up in the design as an attribute is not a conceptual issue to me. As Ron Ross says, "Sponsors of business rule projects must sign off on the sentences—not on graphical data models. Most methodologies and CASE tools have this more or less backwards" [7, p.15]. The ORM reporting facilities in Visio Enterprise allow the domain expert to inspect ORM models fully verbalized into sentences with examples, making validation much easier and safer.

Now that I've stated my bias up front, let's examine the Barker ER notation itself. The basic conventions are illustrated in Figure 2. *Entity types* are shown as soft rectangles (rounded corners) with their name in capitals. *Attributes* are written below the entity type name. Some constraint information may appear before an attribute name. A "#" indicates that the attribute is the primary identifier of the entity type, or at least a component of its primary identification scheme. A "*" or heavy dot "•" indicates the attribute is mandatory (i.e. each instance in the database population of the entity type must have a non-null value recorded for this attribute). A "°" indicates the attribute is optional. Some modelers also use a period "." to indicate the attribute is not part of the identifier.

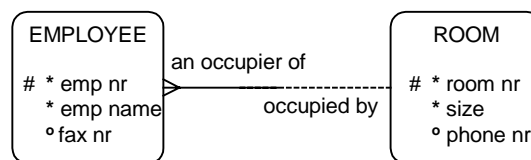


Figure 2 The basic Barker ER notation

Relationships are restricted to binaries (no unaries, ternaries or longer relationships), and are shown as lines with a relationship name at the end from which that relationship name is to be read. This name placement overcomes the ambiguous direction problem mentioned earlier. Both forward and inverse readings may be displayed for a binary relationship, one on either side of the line. This makes the Barker notation superior to UML for verbalizing relationships.

From an ORM perspective, each end (or half) of a relationship line corresponds to a role. Like ORM, Barker ER treats role *optionality* and *cardinality* as distinct, orthogonal concepts, instead of lumping them together into a single concept (e.g. multiplicity in UML). A solid line-half denotes a mandatory role, and a dotted line-half indicates an optional role. For cardinality, a crow's foot intuitively indicates "many", by its many "toes". The absence of a crow's foot intuitively indicates "one". The crow's foot notation was invented by Gordon Everest, who originally used the term "inverted arrow" [3] but now calls it a "fork". Figure 3 shows the correspondence with the ORM notation for uniqueness and mandatory role constraints.

To enable the optionality and cardinality settings to be verbalized, Barker [1, p. 3-5] recommends the following *naming discipline for relationships*. Let $A R B$ denote an infix relationship R from entity type A to entity type B . Name R in such a way that each of the following four patterns results in an English sentence:

each A (must | may) be R (one and only one B | one or more B-plural-form)

Use “must” or “may” when the first role is mandatory or optional respectively. Use “one and only one” or “one or more” when the cardinality on the second role is one or many respectively. For example, the optionality/cardinality settings in Figure 3(a) verbalize as: each Employee must be an occupier of one and only one Room; each Room may be occupied by one or more Employees. This verbalization convention is good for basic mandatory and uniqueness constraints on infix binaries. However it is far less general than ORM’s approach, which applies to instances as well as types, for predicates of any arity, and covers many more kinds of constraint, with no need for pluralization. As a trivial example, the fact instance “Employee ‘101’ an occupier of Room 23” is not proper English, but “Employee ‘101’ occupies Room 23” is good English.

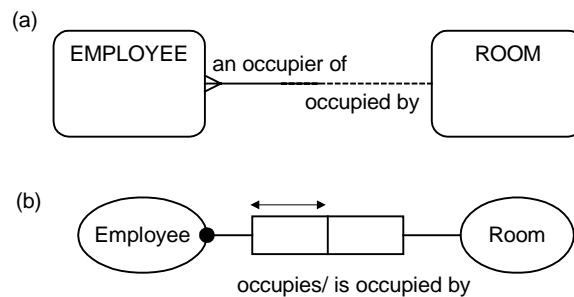


Figure 3 The ER diagram (a) is equivalent to the ORM diagram (b)

If each of the two roles in a binary association may be assigned one of optional/mandatory and one of many/one, there are sixteen patterns. The equivalent Barker ER and ORM diagrams for the first eight of these cases are shown in Figure 4.

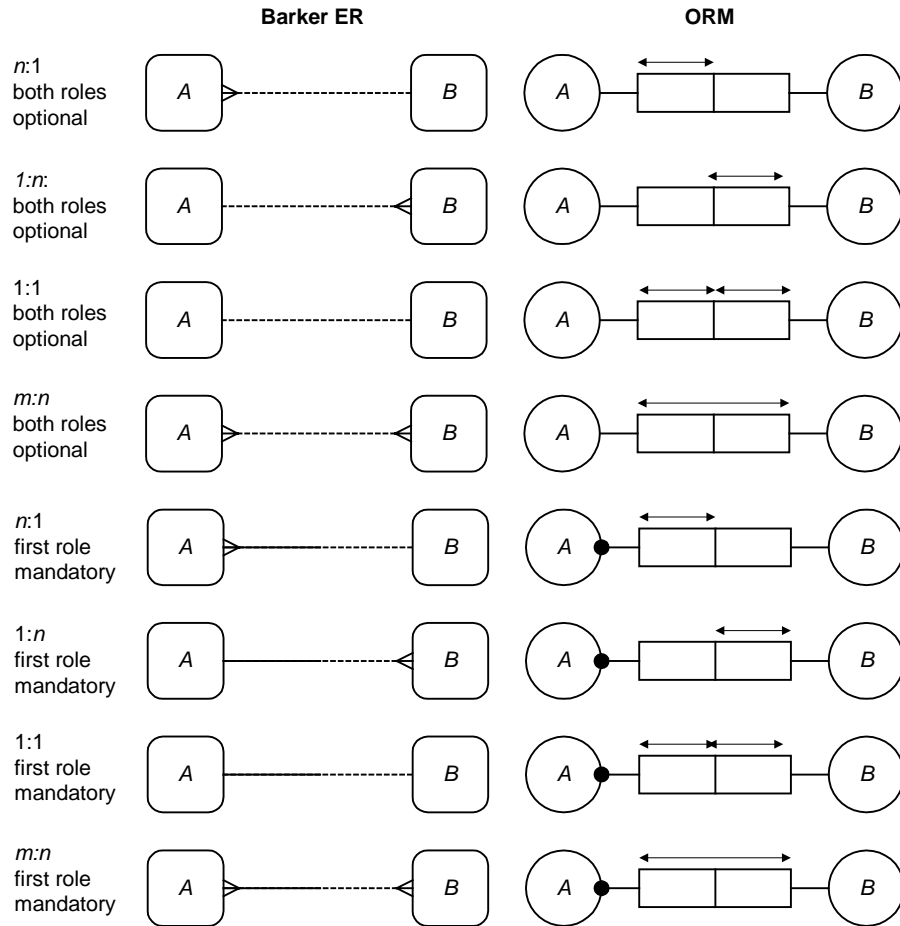


Figure 4 Some equivalent cases

The other eight cases are shown in Figure 5. Although all eight are legal in ORM, the last case where both roles of a many:many relationship are mandatory is considered illegal by Barker.

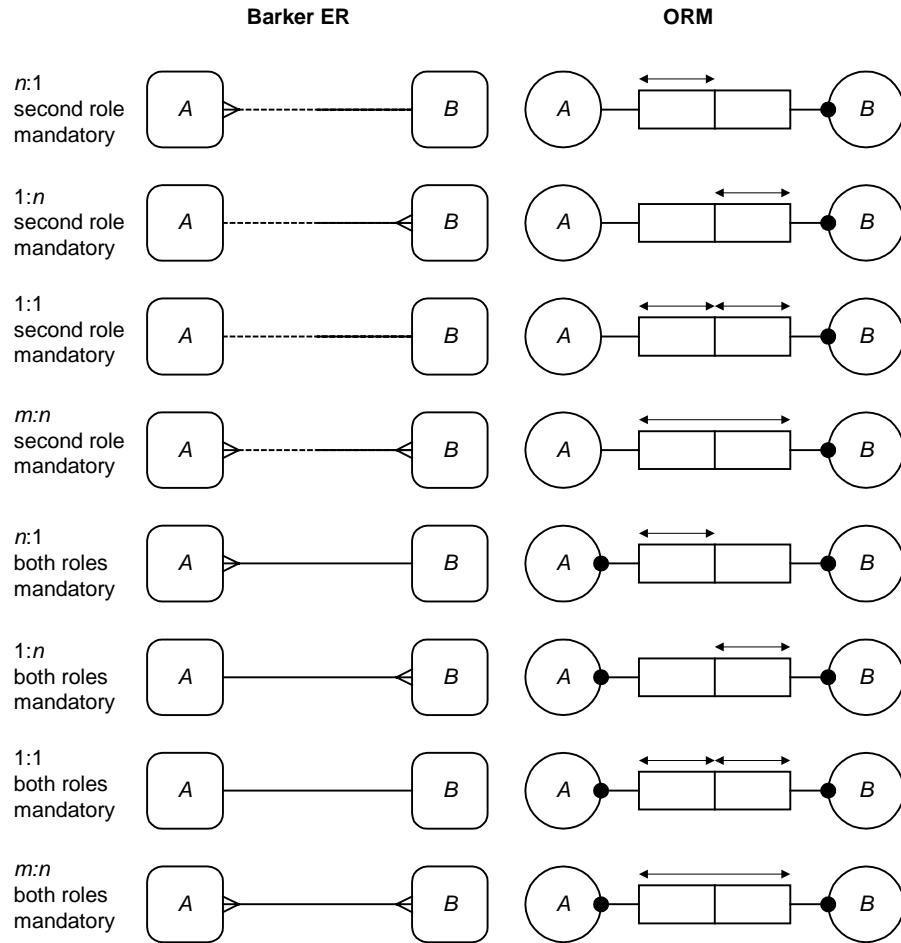


Figure 5 Other equivalent cases

Ring associations that considered illegal by Barker are shown in Figure 6(a). Although rare, they sometimes occur in reality, so should be allowed at the conceptual level, as permitted in ORM. As an exercise, you may wish to invent satisfying populations for the ORM associations in Figure 6 (b). Although considered illegal by Barker, at least some of these patterns are allowed in Oracle's CASE tools.

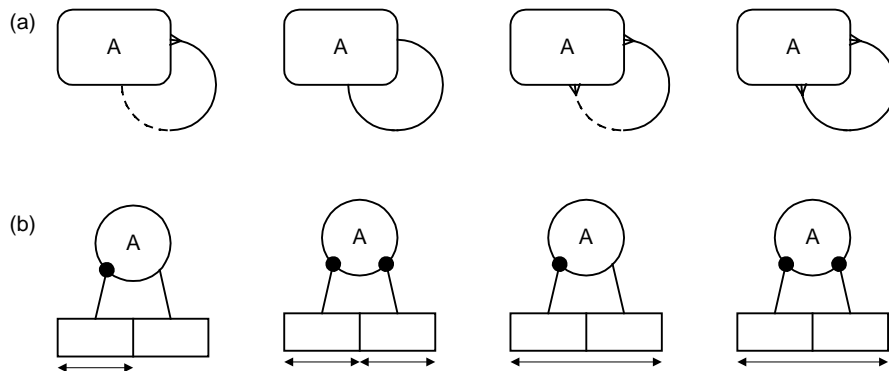


Figure 6 Illegal ring associations in Barker ER (a) that are rare but allowed in ORM (b)

In Barker ER, a bar “|” across one end of a relationship indicates that the relationship is a component of the primary identifier for the entity type at that end. In Figure 7 for example, Employee and Building have simple identifiers, but Room has a composite reference scheme, being identified partly by its room number and partly by the building in which it is included.

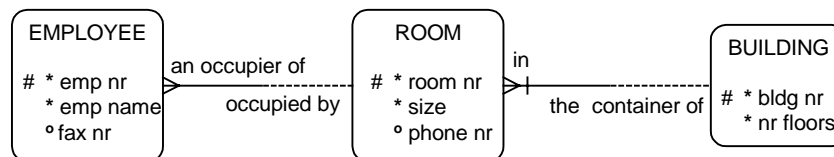


Figure 7 Room is identified by combining its room nr and its relationship to Building

The use of identification bars provides some of the functionality afforded by external uniqueness constraints in ORM. For example, the schemas in Figure 8 are equivalent. The other attributes of Room and Building in ORM would be modeled in ORM as relationships. ORM’s external uniqueness notation seems to me to convey more intuitively the idea that each RoomNr, Building combination is unique (i.e. refers to at most one room). But maybe I’m biased. At any rate, this constraint (as well as any other graphic constraint) can be automatically verbalized in natural language.

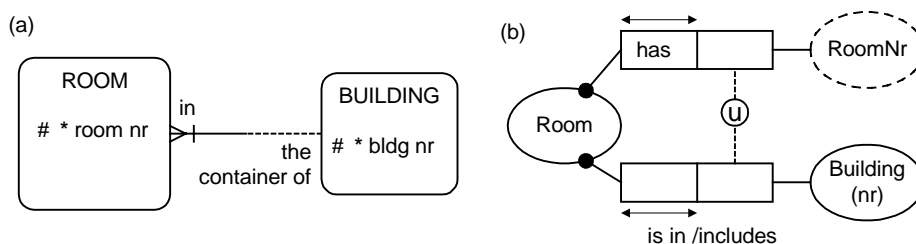


Figure 8 Composite identification in Barker ER (a) and ORM (b)

Some people misread the bar notation for composite identification as a “1”, since this is what the symbol means in many other ER notations. But this isn’t a problem if you don’t have to work with multiple versions of ER. The main problem with the “#” and bar notations is that they cannot be used to declare uniqueness constraints that are not used for a primary identification scheme. A second problem is that they are two very different notations for the same fundamental concept (uniqueness). Because ORM allows constraints to be used wherever they make sense, and always uses relationships instead of attributes, it doesn’t have these problems. An example may help illustrate some of these ideas. Suppose we wanted to model the information shown in Table 1, as well as other facts about rooms.

Table 1 A simple data use case for room scheduling

<i>Room</i>	<i>Time</i>	<i>ActivityCode</i>	<i>ActivityName</i>
20	Mon 9 am	VMC	VisioModeler class
20	Tue 2 pm	VMC	VisioModeler class
33	Mon 9 am	AQD	ActiveQuery demo
33	Fri 5 pm	SP	Staff party
...

The table suggests that rooms can be simply identified by room numbers, so let’s accept that. One way of modeling the situation in Barker ER is shown in Figure 9. Here the bar notation is used to show that RoomTimeSlot is identified by combining its time and room number.

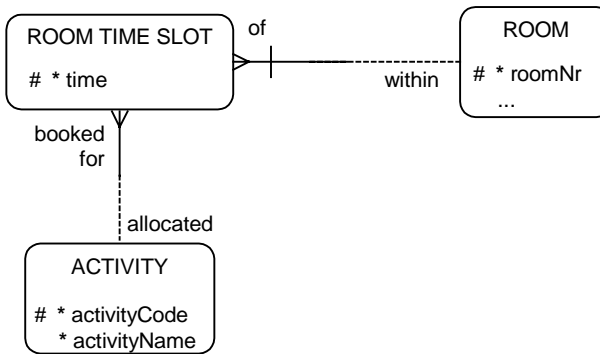


Figure 9 An ER diagram for room scheduling

The use of attributes in this model makes it hard to verbalize and populate the schema for validation purposes. Moreover, there is at least one constraint missing. Compare this with the populated ORM model for the same situation (Figure 10). Here the facts are naturally verbalized as a ternary (Room at Time is booked for Activity) and a binary (Activity has ActivityName). The associated fact tables include the original facts, as well as counter-facts (italicized) to test the constraints. The first counter-row (20, Mon 9 am, AQD) tests the uniqueness constraint that a room at a time is booked for at most one activity. The second counter-row tests the uniqueness constraint that at most one room can be booked for a given activity at a given time. This constraint may well be wrong, but

at least we can express it and test it in ORM. With the ER model there is no way of even specifying the constraint, much less testing it.

The counter rows (SP, Sales phonecalls) and (PTY, Staff party) are designed to check the uniqueness constraints that each Activity has at most one Activity name and vice versa. If these are rejected, the association really is 1:1, as its basic population suggests. Since the ER notation being discussed doesn't include a way of indicating that attributes other than the primary identifier are unique, it isn't very helpful here. As a small point, the Y2K row has been added to the original population to indicate that it is possible for some listed activities to be unscheduled.

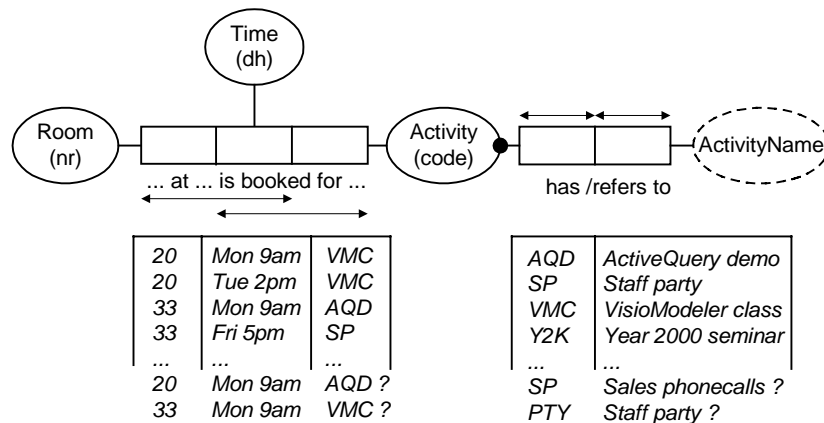


Figure 10 An ORM diagram for room scheduling, with sample and counter data

In case it looks like I'm just bashing attribute-based approaches like ER in this article, let me say again that I find attribute-based models useful for compact overviews and for getting closer to the implementation model. However I generate these by mapping from ORM, which I use exclusively for conceptual analysis. This makes it easier to get the model right in the first place, and to modify it as the underlying domain evolves. Unlike ER (and UML for that matter), ORM was built from a linguistic basis, and its graphic notation was carefully chosen to exploit the potential of sample populations. To reap the benefits of verbalization and population for communication with and validation by domain experts, it's better to use a language that was designed with this in mind. An added benefit of ORM is that its graphic notation can capture many more business rules than popular ER notations.

Next issues

Later articles in this series will consider more advanced aspects of the Barker ER notation, including exclusion constraints, frequency constraints, subtyping and non-transferable relationships, and then examine the Information Engineering notation for ER, before concluding with a discussion of IDEF1X.

References

1. Barker, R. 1990, *CASE*Method: Tasks and Deliverables*, Addison-Wesley, Wokingham, England.
2. Chen, P.P. 1976, 'The entity-relationship model—towards a unified view of data', *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36.
3. Everest, G. 1976, 'Basic Data Structure Models Explained with a Common Example', *Proc. Fifth Texas Conference on Computing Systems*, (Austin, TX, 1976 October 18-19), IEEE Computer Society publications office, Long Beach, CA, pp. 39-45.
4. Halpin, T.A. 1998-9, 'UML data models from an ORM perspective: Parts 1-10', *Journal of Conceptual Modeling*, InConcept, Minneapolis USA.
5. Halpin, T.A. & Bloesch, A.C. 1999, 'Data modeling in UML and ORM: a comparison', *Journal of Database Management*, vol. 10, no. 4, Idea group Publishing Company, Hershey, USA, pp. 4-13.
6. Hay, D.C. 1999, 'There is no object-oriented analysis', *DataToKnowledge Newsletter*, vol. 27, no. 1, Business Rule Solutions, Inc., Houston TX, USA.
7. Hay, D.C. 1999, 'Object orientation and information engineering: UML', *The Data Administration Newsletter*, no. 9, (June 1999), ed. R.S. Reiner, available online at www.tdan.com.
8. Ross, R.G. 1998, *Business Rule Concepts*, Business Rule Solutions, Inc., Houston TX, USA.

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

Entity Relationship modeling from an ORM perspective: Part 2

Terry Halpin
Microsoft Corporation

Introduction

This article is the second in a series of articles dealing with Entity Relationship (ER) modeling from the perspective of Object Role Modeling (ORM). Part 1 provided a brief overview of the ER approach, and then covered the basics of the Barker ER notation [1], that has long been supported in CASE tools from vendors such as Oracle Corporation. In this notation, entity types are depicted as named, soft rectangles, and binary relationships are shown as lines with forward and inverse names. If shown, attributes are listed below the entity type name. A “#” indicates that an attribute is [part of] the primary identifier of the entity type, a “*” indicates the attribute is mandatory and a “o” indicates the attribute is optional. Only binary relationships are allowed, so a role corresponds to a half-line (half a relationship line). If a role is mandatory, its half-line is solid. If a role is optional, its half-line is dashed. For role cardinality, a cross-foot at the end indicates “many” and its absence indicates “1”. A bar “|” across one end of a relationship indicates that the relationship is a component of the primary identifier for the entity type at that end. Part 1 discussed examples of the above notations and compared them with the corresponding ORM notations. This second article briefly discusses verbalization, then examines the Barker ER notation for exclusion constraints, frequency constraints, subtyping and non-transferable relationships.

Barker ER: verbalization

To enable the optionality and cardinality settings to be verbalized, Barker [1, p. 3-5] recommends the following *naming discipline for relationships*. Let $A R B$ denote an infix relationship R from entity type A to entity type B . Name R in such a way that each of the following four patterns results in an English sentence:

each A (must | may) be R (one and only one B | one or more B-plural-form)

Use “must” or “may” when the first role is mandatory or optional respectively. Use “one and only one” or “one or more” when the cardinality on the second role is one or many respectively. For example, the optionality/cardinality settings in Figure 1(a) verbalize as: **each Employee must be an occupier of one and only one Room; each Room may be occupied by one or more Employees.**

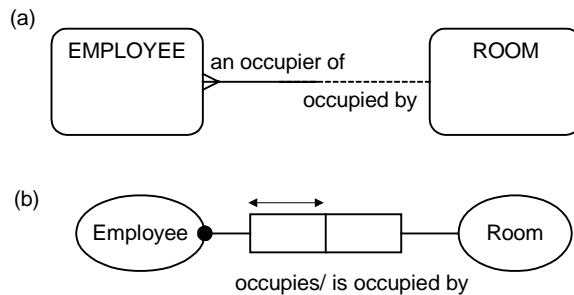


Figure 1 The ER diagram (a) is equivalent to the ORM diagram (b)

The constraints on the left hand role in the equivalent ORM model shown in Figure 1(b) verbalize as: **each Employee occupies some Room; each Employee occupies at most one Room.** If desired, these constraints may be combined to verbalize as: **each Employee occupies exactly one Room.** Since the right-hand role has no constraints, this is not normally verbalized in ORM (unlike Barker ER). However the lack of any uniqueness constraint could be verbalized explicitly as: **it is possible that the same Room is occupied by more than one Employee.** If no inverse reading is available, it can be verbalized as: **it is possible that more than one Employee occupies the same Room.** If you would like this explicit verbalization capability added as a configurable option to the verbalizer in Visio Enterprise, please email me at TerryHa@microsoft.com.

Regarding the lack of an explicit mandatory role constraint on the right-hand role, I am less inclined to want that verbalized explicitly, because it may well be unstable. If Room plays no other fact roles, the role is mandatory by implication (Room has not been declared independent), so verbalization may well confuse here. If Room does play another fact role, and we decide that some rooms may be unoccupied, we could declare this explicitly as: it is possible that some Room is occupied by no Employee. Or equivalently: it is not necessary that each Room is occupied by some Employee. If no inverse reading is available, it could be verbalized thus: it is possible that no Employee occupies some Room. If you would like an option for explicit verbalization of optional roles in Visio Enterprise, please email me your thoughts on this.

To its credit, the Barker verbalization convention is good for basic mandatory and uniqueness constraints on infix binaries. However it is far less general than ORM's approach, which applies to instances as well as types, for predicates of any arity, infix or mixfix, and covers many more kinds of constraint, with no need for pluralization. As a trivial example, the fact instance "Employee '101' an occupier of Room 23" is not proper English, but "Employee '101' occupies Room 23" is good English.

Exclusion constraints

In Barker ER notation, an exclusion constraint over two or more roles is shown as an "exclusive arc" connected to the roles with a small dot or circle. For example, Figure 2(a) includes the constraint that no employee may be allocated both a bus pass and a parking bay. In ORM this constraint is depicted by connecting "⊗" to the relevant roles by a dotted line, as shown in Figure 2(b).

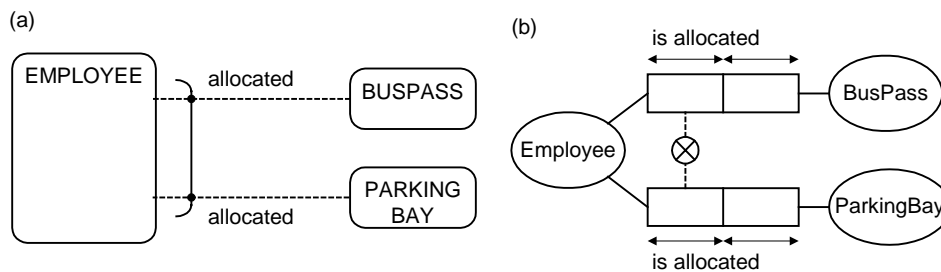


Figure 2 A simple exclusion constraint in (a) Barker ER notation and (b) ORM

To declare that two or more roles are mutually exclusive and disjunctively mandatory, the Barker notation uses the exclusive arc, but each role is shown as mandatory (solid line). For example, in Figure 3(a) each account is owned by a person or a company, but not both. This notation is liable to mislead, since it violates the orthogonality principle in language design. Viewed by itself, the first role of the association Account owned by Person would appear to be mandatory, since a solid line is used. But the role is actually optional, since superimposing the exclusive arc changes the semantics of the solid line to mean the role belongs to a set of roles that are disjunctively mandatory.

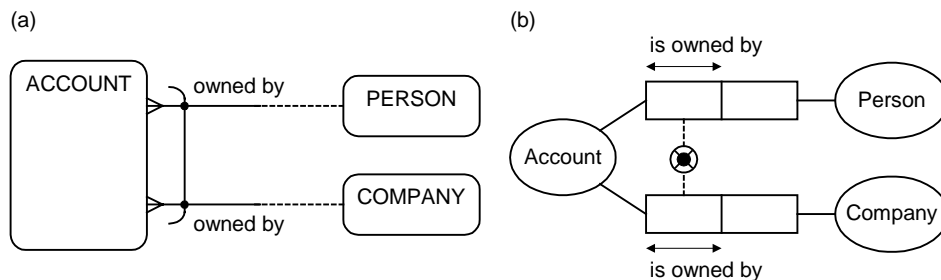


Figure 3 An exclusive-or constraint in (a) Barker ER notation and (b) ORM

Contrast this with the equivalent ORM model shown in Figure 3(b). Here an exclusion constraint \otimes is orthogonally combined with a disjunctive mandatory (inclusive-or) constraint \odot to produce an exclusive-or constraint, shown here by the “lifebuoy” or partition symbol formed by overlaying one constraint symbol on the other. As an alternative, the inclusive-or and exclusion constraints may be displayed separately.

The ORM notation makes it clear that each role is individually optional, and that the exclusive-or constraint is a combination of inclusive-or and exclusion constraints. Suppose we modified our business so that the same account could be owned by both a person and a company. Removing just the exclusion constraint from the model leaves us with the inclusive-or constraint \odot that each account is owned by a person or company. Like UML, the Barker ER notation doesn’t even have a symbol for an inclusive-or constraint, so is unable to diagram this or the many other cases of this nature that occur in practice.

In the Barker notation, a role may occur in at most one exclusive arc. ORM has no such restriction. For example, in Figure 4(a) no student can be both ethnic and aboriginal, and no student can be both an aboriginal and a migrant (these rules come from a student record system in Australia). Even if Barker notation supported unaries (it doesn’t) this situation could not be handled by exclusive arcs. Like UML, Barker ER does not provide a graphic notation for exclusion constraints over role-sequences. For instance, it cannot capture the ORM pair-exclusion constraint in Figure 4(b), which declares that no person who wrote a book may review the same book. Such rules are very common. Moreover, the Barker notation cannot express any ORM subset or equality constraints at all, even over simple roles.

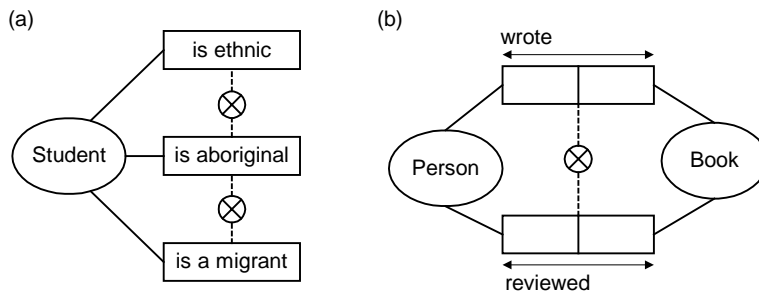


Figure 4 Some ORM exclusion constraints not handled by exclusive arcs in Barker ER notation

Frequency constraints

The Barker ER notation allows simple frequency constraints to be specified. For any positive integer n , a constraint of the form $= n, < n, \leq n, > n, \geq n$ may be written beside a single role to indicate the number of instances that may be associated with an instance playing the other role. For example, the frequency constraint “ ≤ 2 ” in Figure 5 indicates that each person is a child of at most two parents. In the Barker notation, this constraint is placed on the parent role, making it easy to read the constraint as a sentence starting at the other role. In ORM the constraint is placed on the child role, making it easy to see the impact of the constraint on the population (each person appears at most twice in the child role population). Unlike the Barker notation, ORM allows frequency constraints to include ranges (e.g. 2-5) and to apply to role-sequences.

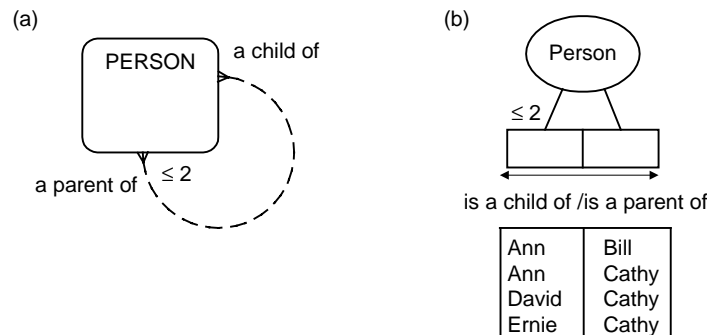


Figure 5 A simple frequency constraint in (a) Barker ER notation and (b) ORM

Subtyping

In Barker ER notation, subtyping is depicted with a version of Euler diagrams. In effect, only partitions (exclusive and exhaustive) can be displayed. For example, Figure 6(a) indicates that each patient is a male patient or female patient but not both. Like UML, ORM displays subtyping using directed acyclic graphs (DAGs). Subtype exclusion and exhaustion constraints are normally omitted in ORM, as in Figure 6(b), since they are implied by the subtype definition and other constraints (e.g. mandatory, uniqueness and value constraints on Patient is of Gender). However they can be explicitly displayed as in Figure 6(c).

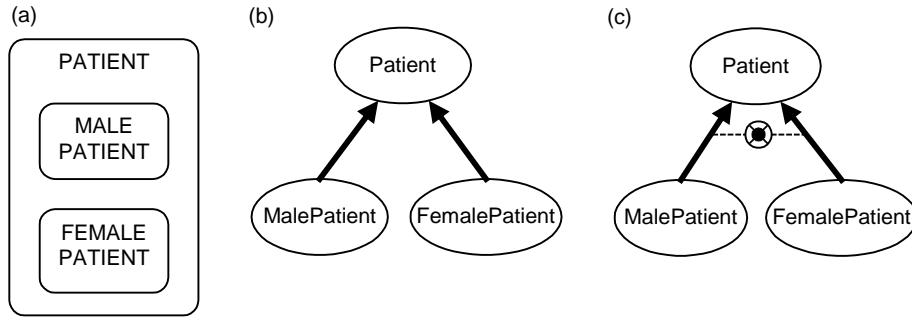


Figure 6 A subtype partition in (a) Barker ER, (b) implicit ORM and (c) explicit ORM notation

Euler diagrams are good for simple cases, since they intuitively show the subtype inside its supertype. However unlike DAGs, they are hopeless for complex cases (e.g. many overlapping subtypes), and they make it inconvenient to attach details to the subtypes. For the latter reason, attributes are often omitted from subtypes when the Barker notation is used.

In the Barker notation, if the original subtype list is not exhaustive, an “Other” subtype is added to make it so, even if it plays no specific role. For example, in Figure 7 a vehicle is a car or truck or possibly something else, and a car is a sedan or wagon or possibly something else.

A major problem with the Barker notation for subtyping is that it does not depict overlapping subtypes (e.g. Manager and FemaleEmployee as subtypes of Employee) or multiple inheritance (e.g. FemaleManager as a subtype of FemaleEmployee and Manager). While it is possible to implement multiple inheritance in single inheritance systems (e.g. Java) by using some low level tricks, for conceptual modeling purposes multiple inheritance should be simply modeled as multiple inheritance.

As a final comparison point about subtyping, Barker ER lacks ORM’s capability for formal subtype definitions and context-dependent identification schemes.

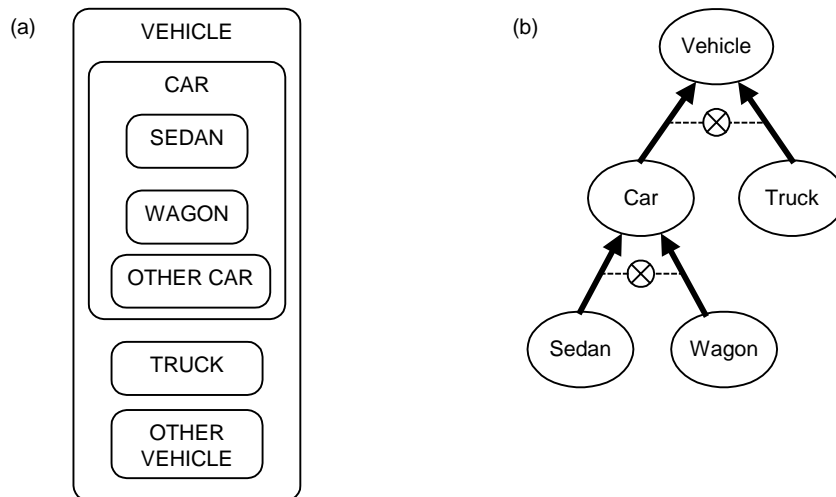


Figure 7 Non-exhaustive, exclusive subtypes in (a) Barker ER and (b) ORM

Non-transferable relationships

In addition to its static constraint notation, Barker ER includes a dynamic “changeability constraint” for marking “*non-transferable relationships*”. This constraint declares that once an instance of an entity type plays a role with an object, it cannot ever play this role with another object. This is indicated by adding an open diamond to the constrained role. For example, Figure 8(a) declares that the birth country of a person is non-transferable.

As indicated in Figure 8(b), ORM does not currently include a notation for this constraint. It would be possible to add a notation for this (as well as UML’s changeability settings of changeable, frozen, addOnly), but it is at least debatable whether this is advisable. If we were to add such a notation, we would need to ensure that the implemented model is still open to error corrections by duly authorized users. For example, if my birth country was mistakenly entered as Austria, it should be possible to change this to Australia. For further discussion on this issue, see my comments on changeability properties in [2]. If you have some strong views on this issue, please email me your thoughts.

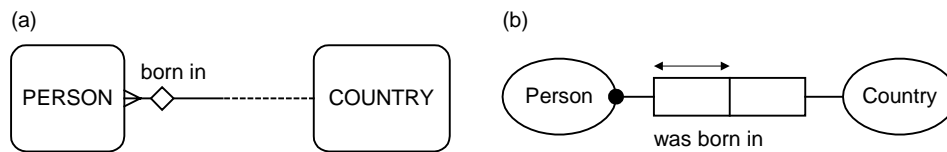


Figure 8 Non-transferable nature of a relationship is declared in (a) Barker ER, but not in (b) ORM

Conclusion

The Barker ER notation does a good job of expressing simple mandatory, uniqueness, exclusion and frequency constraints, simple subtyping and also non-transferable relationships. However, if a feature is modeled as an attribute instead of as a relationship, very few of these constraints can be specified for it. In contrast to ORM, the Barker ER notation does not support unary, n -ary or objectified associations (nesting). Moreover it lacks support for most of the advanced ORM constraints (e.g. subset, multi-role exclusion, ring constraints and join constraints). It does not include a formal textual language such as ConQuer for specifying queries, other constraints and derivation rules at the conceptual level. Nevertheless it is better than many other notations for ER modeling, and is still widely used. If you ever need to specify a model in Barker ER notation, I suggest you first do the model in ORM, then map it to the Barker notation and make a note of any rules that can’t be expressed there diagrammatically.

Next issues

Later articles in this series will examine the Information Engineering notation for ER, before concluding with a discussion of IDEF1X.

References

1. Barker, R. 1990, *CASE*Method: Tasks and Deliverables*, Addison-Wesley, Wokingham, England.
2. Halpin, T.A. 1999, ‘UML data models from an ORM perspective: Part 10’, *Journal of Conceptual Modeling*, InConcept, Minneapolis USA.

Entity Relationship modeling from an ORM perspective: Part 3

Terry Halpin
Microsoft Corporation

Introduction

This article is the third in a series of articles dealing with Entity Relationship (ER) modeling from the perspective of Object Role Modeling (ORM). Part 1 provided a brief overview of the ER approach, and then covered the basics of the Barker ER notation. Part 2 completed the examination of the Barker ER notation by discussing verbalization, exclusion constraints, frequency constraints, subtyping and non-transferable relationships. Both parts compared the Barker notations with the corresponding ORM notations. This article discusses the Information Engineering notation for ER, relating it to relevant ORM constructs.

The *Information Engineering* (IE) approach began with the work of Clive Finkelstein in Australia, and CACI in the UK, and was later adapted by James Martin. Different versions of IE exist, with no single standard. In one form or other, IE is supported by many data modeling tools, and is one of the most popular notations for database design.

Entity types, attributes and associations

In the IE approach, *entity types* are shown as named rectangles, as in Figure 1(a). *Attributes* are often displayed in a compartment below the entity type name, as in Figure 1(b), but are sometimes displayed separately (e.g. bubble charts). Some versions support basic constraints on attributes (e.g. Ma/Op/Unique).

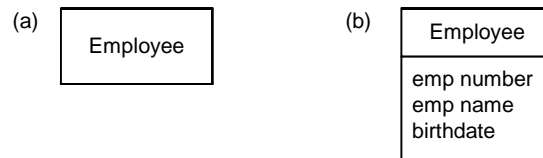


Figure 1 Typical IE notation for (a) entity type and (b) entity type with attributes

Relationships are typically restricted to binary associations only, which are shown as named lines connecting the entity types. As with the Barker notation, a half-line or line end corresponds to a role in ORM. Optionality and cardinality settings are indicated by annotating the line ends. To indicate that a role is *optional*, a circle “○” is placed at the other end of the line, signifying a minimum participation frequency of 0. To indicate that a role is *mandatory*, a stroke “|” is placed at the other end of the line, signifying a minimum participation frequency of 1. After experimenting with some different notations for a cardinality of “many”, Finkelstein settled on the intuitive crow’s foot symbol suggested by Dr. Gordon Everest.

In conjunction with a minimum frequency of 0 or 1, a stroke “|” is often used to indicate a maximum frequency of 1. With this arrangement, the combination “○|” indicates “at most one” and the combination “||” indicates “exactly one”. This is the convention that I will use in this section. However different IE conventions exist. For example, some assume a maximum cardinality of 1 if no crows foot is used, and hence use just a single “|” for “exactly one”. Clive Finkelstein uses the combination “○|” to mean “optional but will become mandatory”, which is really a dynamic rather than static constraint—this can be combined with a crow’s foot. Some conventions allow a crow’s foot to mean the minimum (and hence maximum) frequency is many. So if you are using a version of IE, you should check which of these conventions applies.

Figure 2 shows a simple IE diagram and its equivalent ORM diagram. With IE, as you read an association from left to right, you verbalize the constraint symbols at the right-hand end. Here, each employee occupies exactly one (at least 1 and at most 1) room. Although inverse predicates are not always supported in IE, you can supply these yourself to obtain a verbalization in the other direction. For example:

“Each room is occupied by zero or more employees”. As with the Barker notation, a plural form of the entity type name is introduced to deal with the many case.

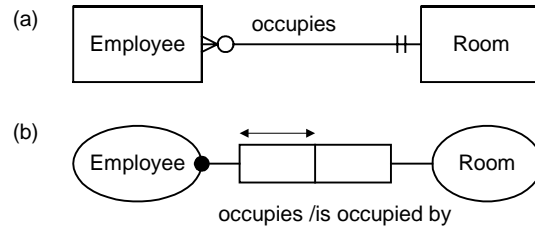


Figure 2 The IE diagram (a) is equivalent to the ORM diagram (b)

The IE notation is similar to the Barker notation in showing the maximum frequency of a role by marking the role at the other end. But unlike the Barker notation, the IE notation shows the optionality/mandatory setting at the other end as well. In this sense, IE is like UML (even though different symbols are used). As discussed in an earlier article, there are sixteen possible constraint patterns for optionality and cardinality on binary associations. Figure 3 shows eight cases in IE notation together with the equivalent cases in ORM.

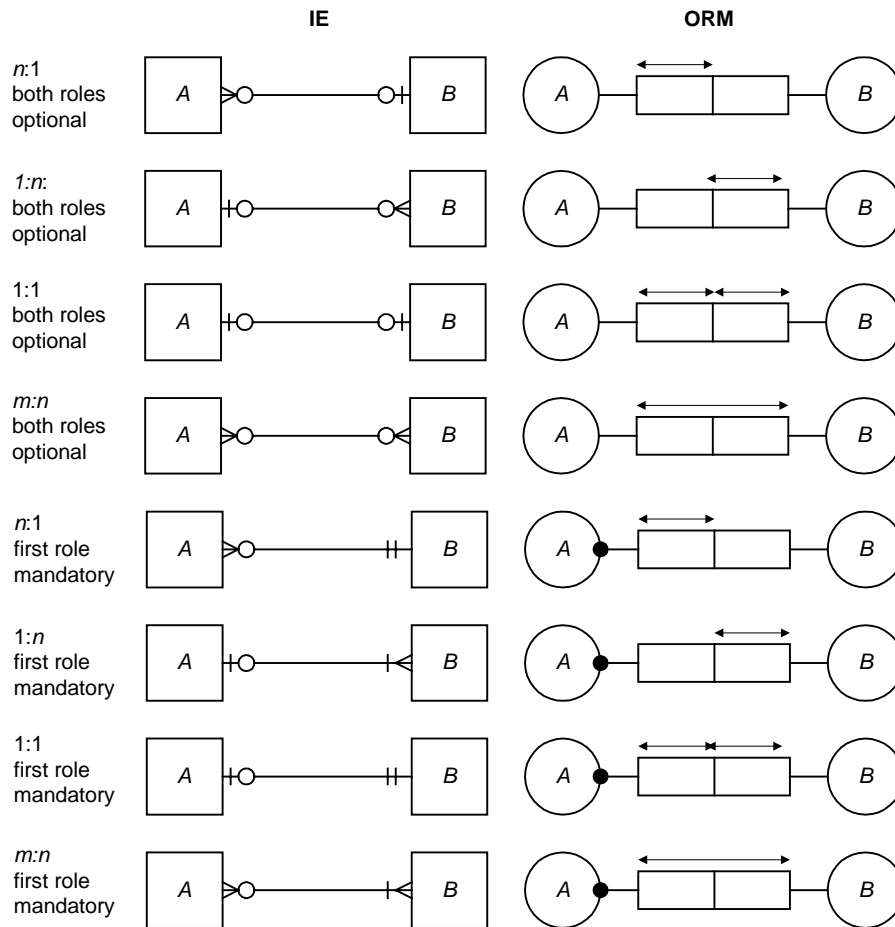


Figure 3 Some equivalent constraint patterns

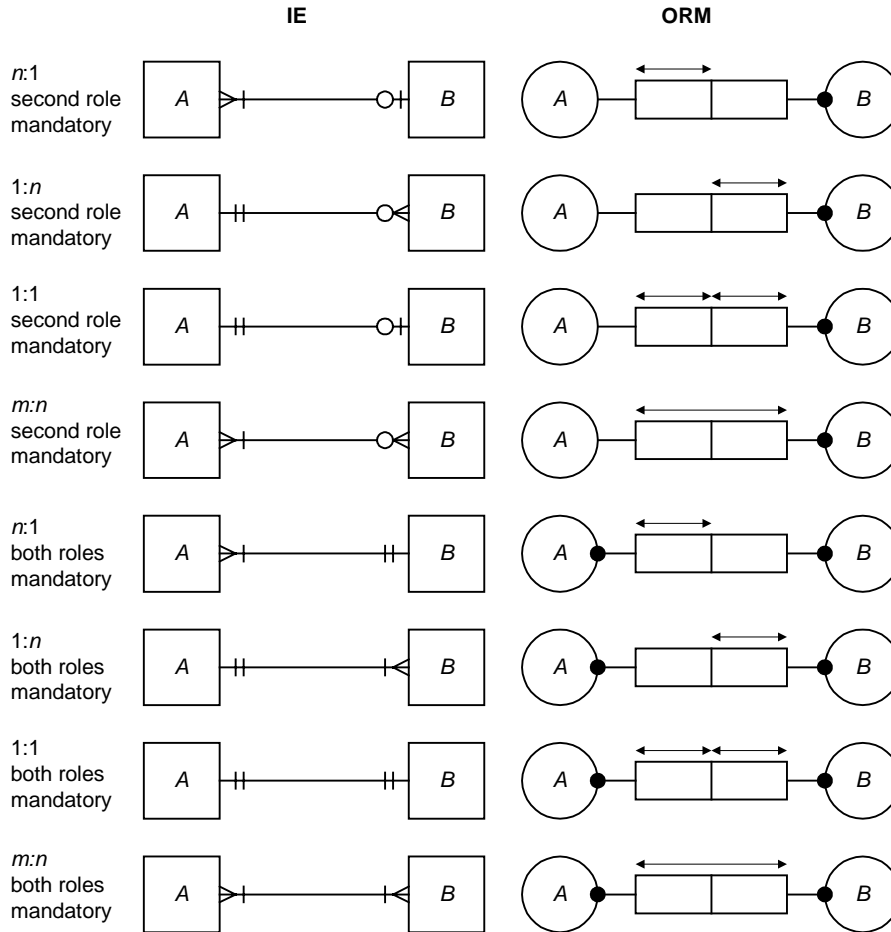


Figure 4 Other equivalent cases

The other eight cases are shown in Figure 4. An example using the different notation for IE used by Finkelstein is shown in Figure 5. Here the single bar on the left end of the association indicates that each computer is located in exactly one office. The circle, bar and crow's foot on the right end of the association collectively indicate that each office must eventually house one or more computers. This "optional becoming mandatory" constraint has no counterpart in ORM, and is not supported in most IE modeling tools.

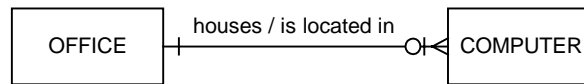


Figure 5 Finkelstein's notation for IE is different from ours

Some modeling tools support the IE notation for $n:1$, $1:n$ and $1:1$ associations but not $m:n$ (many to many) associations. For such tools, four of the sixteen cases cannot be directly represented. In this situation, you can model the $m:n$ cases indirectly by introducing an "intersection entity type" with mandatory $n:1$ associations to the original entity types. For example, the $m:n$ case with both roles optional may be handled by introducing the object type C as shown in Figure 6 for both IE and ORM. In ORM, C is a *co-referenced* object type, and the transformation is an instance of a flatten/coreference equivalence.

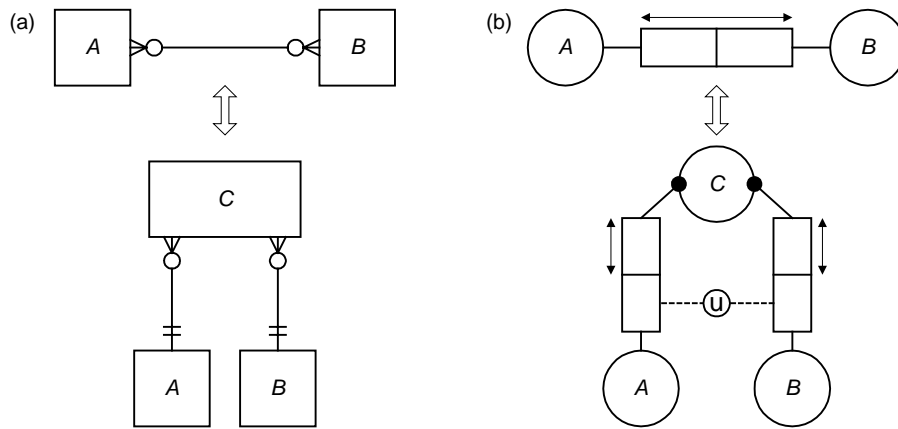


Figure 6 An $m:n$ association remodeled as an entity type with $n:1$ associations

As an example, the $m:n$ association Person plays Sport can be transformed into the mandatory $n:1$ associations: Play is by Person; Play is of Sport. However such a transformation is often very unnatural, especially if nothing else is recorded about the co-referenced object type. So any truly conceptual approach must allow $m:n$ associations to be modeled directly.

Advanced constraints and subtyping

Some versions of IE support an *exclusive-or constraint*, shown as a black dot connected to the alternatives. Figure 7(a) depicts the situation where each employee is allocated a bus pass or parking bay, but not both. The equivalent ORM schema is shown in Figure 7(b). Unlike ORM, IE does not support an inclusive-or constraint. Nor does it support exclusion constraints over multi-role sequences.

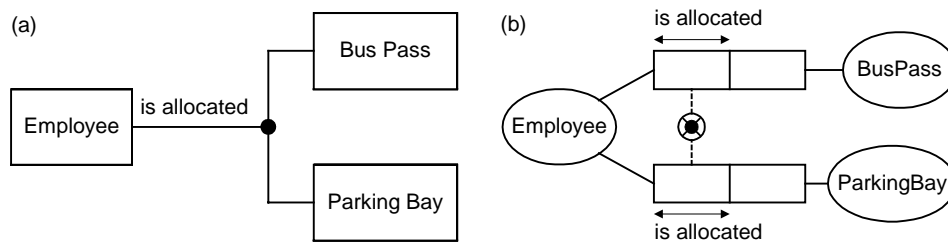


Figure 7 An exclusive-or constraint in (a) IE and (b) ORM

Subtyping schemes for IE vary. Sometimes Euler diagrams are used, adding a blank compartment if needed for “Other”. Sometimes directed acyclic graphs are used, possibly including subtype relationship names and optionality/cardinality constraints. Figure 8 show three different subtyping notations for partitioning Patient into the subtypes MalePatient and FemalePatient. There is no formal support for subtype definitions, and no provision for context-dependent reference. Multiple inheritance may or may not be supported, depending on the version.

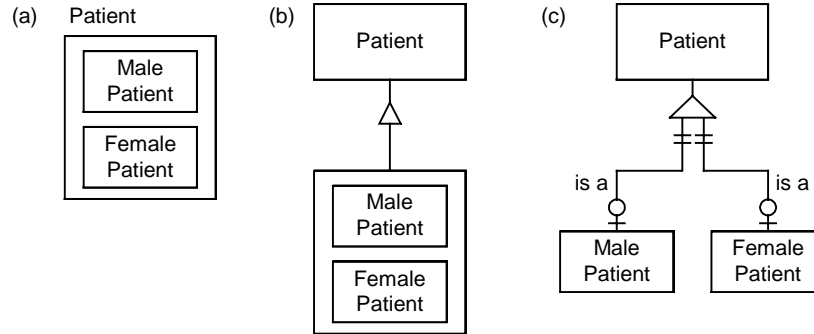


Figure 8 Some different IE notations for subtyping

Conclusion

Although far less expressive than ORM, IE does a good job of covering basic constraints. If you ever need to specify a model in IE notation, I suggest you first do the model in ORM, then map it to IE and make a note of any rules that can't be expressed there diagrammatically. Often referred to as “the father of information engineering”, Clive Finkelstein is an amiable Aussie who is still actively engaged in the information engineering discipline. He developed a set of modeling procedures to go with the notation, extended IE to Enterprise Engineering (EE), and recently began applying data modeling to work in XML (Extensible Markup Language). An overview of IE by Clive can be found in [1], and details on his recent work are given in [2]. For a look at the IE approach used by James Martin, see [3]. Martin's more recent books tend to use the UML notation instead. In practice however, IE is used far more extensively for database design than is UML, which is mostly used for object-oriented code design.

Next issue

The next article in this series discusses the IDEF1X notation.

References

1. Finkelstein, C. 1998, 'Information engineering methodology', *Handbook on Architectures of Information Systems*, eds. P. Bernus, K. Mertins & G. Schmidt, Springer-Verlag, Berlin, Germany, pp. 405-427.
2. Finkelstein, C. & Aiken, P. 2000, *Building Corporate Portals with XML*, McGraw-Hill, New York.
3. Martin, J. 1993, *Principles of Object Oriented Analysis and Design*, Prentice Hall, Englewood Cliffs, NJ.

Information Modeling and Higher-order Types

Terry Halpin

Northface University
Salt Lake City, Utah, USA.
e-mail: terry.halpin@northface.edu

Abstract: While some information modeling approaches (e.g. the Relational Model, and Object-Role Modeling) are typically formalized using first-order logic, other approaches to information modeling include support for higher-order types. There appear to be three main reasons for requiring higher-order types: (1) to permit instances of categorization types to be types themselves (e.g. the Unified Modeling Language introduced power types for this purpose); (2) to directly support quantification over sets and general concepts; (3) to specify business rules that cross levels/metalevels (or ignore level distinctions) in the same model. As the move to higher-order logic may add considerable complexity to the task of formalizing and implementing a modeling approach, it is worth investigating whether the same practical modeling objectives can be met while staying within a first-order framework. This paper examines some key issues involved, suggests techniques for retaining a first-order formalization, and also makes some suggestions for adopting a higher-order semantics.

1 Introduction

Following Codd's use of first-order logic to formally underpin the relational model of data [4], most formalizations of information modeling approaches restricted their logical foundations to first-order (where quantification is permitted over individuals only, not predicates). This is the case for Entity Relationship (ER) modeling [3], as well as Object-Role Modeling (ORM) and its variants [e.g. 2, 12, 13]. A full formalization of ORM's fact-oriented (attribute-free) approach to information modeling was first provided in [11], with alternative formalizations supplied later [5, 15]. In contrast, the Unified Modeling Language (UML) [19, 20, 22] introduced the notion of powertypes, whose instances may themselves be types, thus requiring higher-order semantics.

There appear to be three main arguments for requiring higher-order types to logically underpin information modeling semantics:

- to allow one to think of instances of certain categorization types (e.g. AccountType, CarModel) as being types themselves (as for UML powertypes);
- to formalize very directly the semantics of flexible data structures where attribute entries may themselves denote sets or general concepts (e.g. object-relational tables in non-first normal form);
- to allow one to specify business rules that seem to cross levels/metalevels (or ignore level distinctions) in the same model (e.g. the Finance department is responsible for defining the possible values of AccountType).

As the move to higher-order logic may add considerable complexity to the task of formalizing and implementing a modeling approach, it is worth investigating whether the same practical modeling objectives can be achieved while staying within a first-order framework. This paper examines some key issues involved, suggests techniques to maintain a first-order formalization, and also makes some suggestions for adopting a higher-order semantics. The examples are presented in ORM and/or UML notation, but the issues are relevant to all information modeling approaches.

Section 2 addresses the question of whether instances of categorization types may themselves be types. Section 3 considers what higher-order logic may be appropriate to cater for higher-order types. Section 4 provides an alternate first-order approach to categorization schemes. Section 5 discusses whether higher-order logic is needed to formalize the presence of set-structures or the ability to cross levels/metalevels in the same model. Section 6 summarizes the main results, suggests topics for future research, and lists references for further reading.

2 May instances of categorization-types be types themselves?

Figure 1 depicts a simple schema in (a) ORM and (b) UML notation. The ORM diagram may be interpreted as follows. Four object types (Account, AccountType, SavingsAccount, and InterestRate) are depicted as named ellipses. Here “Account” means bank account. The thick arrow indicates that SavingsAccount is a subtype of Account. As in logic, a predicate is a proposition with object-holes in it. In ORM, a predicate is treated as a named sequence of one or more roles, each of which is depicted as a box. Combining a predicate with its sequence of object types produces a fact type (e.g. Account is of AccountType, SavingsAccount earns InterestRate). Simple identification schemes may be abbreviated in parentheses. For example, Account(Nr) abbreviates the injective (1:1 into) fact type Account has AccountNr.

The value constraint {‘CheckingAccount’, ‘SavingsAccount’} indicates the possible names of account types. Arrow-tipped lines across one or more roles denote uniqueness constraints, indicating that instantiations of that role sequence must be unique. For example, the uniqueness constraint on the first role of Account is of AccountType indicates that entries in the fact column for that role must be unique, which may be formally verbalized as: **each** Account is of **at most one** AccountType.

A solid dot (possibly circled) connected to a set of one or more roles denotes a mandatory constraint over that role set. For example, the mandatory dot connected to the first role of Account is of AccountType indicates that **each** Account is of **some** AccountType. The text beneath the diagram provides a formal definition for the SavingsAccount subtype. For discussion purposes, the two AccountType instances named “CheckingAccount” and “SavingsAccount”, are depicted here as a white dot and shaded dot respectively.

In the UML class diagram, AccountType is an enumerated type with two values¹, and provides the type for the accountType attribute of Account, while SavingsAccount is a subclass of the Account class.

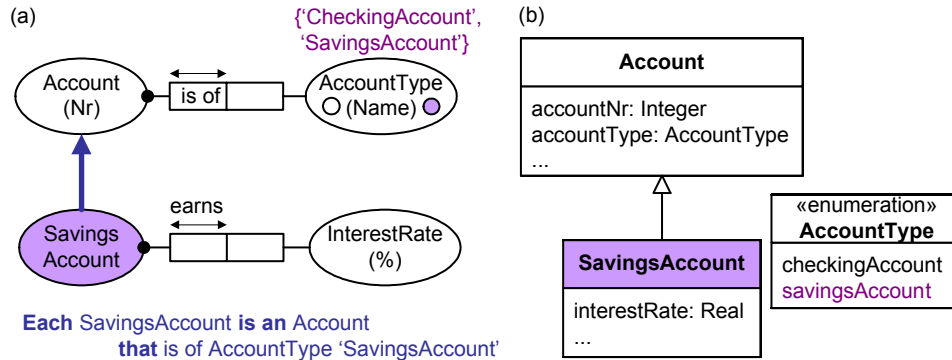


Figure 1 Is the AccountType instance for “SavingsAccount” identical to the subtype SavingsAccount?

In practice, we would normally remove or expand the value constraint, to allow other types of account (e.g. LoanAccount) without modifying the schema. Whether or not we include such a value constraint, we need to address the fundamental question, which may be phrased in ORM terms as: *Is the AccountType instance denoted by the shaded dot identical to the subtype SavingsAccount?* If we answer Yes, then we have a case of an instance being a type in the same model.

In the UML schema, the use of an enumerated type demands a No answer, because UML treats enumeration types as data types whose instances are literals [20, p. 96]. However, as discussed shortly, UML allows us to remodel the situation using a powertype for AccountType, which requires a Yes answer.

As a general point, in specifying a fact type as the application of a predicate to a sequence of object types, we understand that *the predicate applies to instances of the object types*, not the types themselves. For example, consider the fact type Person was born in Country. The “was born in” predicate is understood to apply to ordered pairs of persons and countries (e.g. Niklaus Wirth was born in Switzerland, Terry Halpin was born in Australia). It does not make sense to say the object type Person was born in the object type Country. A similar comment holds in UML when applying associations to a sequence of classes.

¹ In UML 2.0, enumeration types may be extended (by adding further values) in other packages or profiles. This seems inconsistent with a clean approach to subtyping based on substitutability semantics, where subtypes may strengthen but not weaken constraints on their supertypes.

Now let us return to the question as to whether in Figure 1(a) the AccountType instance denoted by the shaded dot is identical to the subtype SavingsAccount. In order for this identity to be even possible, the semantics of the fact type Account is of AccountType should satisfy at least the following necessary conditions:

1. The “*is of*” predicate means “is a member of” or “is an instance of”, i.e. \in (*set membership*).
2. *Only Account instances may be instances of AccountType instances.*

These conditions (which in combination we’ll call *homogeneous set-membership*) arguably follow from the *indiscernibility of identicals* (if $a = b$, then a and b have exactly the same properties). The subtype SavingsAccount includes precisely all the possible savings accounts in the business domain—no other things can be instances of it. If we agree that the instance of AccountType denoted by the shaded dot *is* the subtype SavingsAccount, then only accounts can bear the instance-of relationship to it.

If the modeler wishes to view the model in this way, we should allow this interpretation, as it is not inconsistent. On the other hand, we should not force the modeler to adopt this interpretation, as there are often better ways to model such situations that do not require such a commitment to instances being types in the same model (see later discussion).

If we allow the type = instance interpretation, we must use higher-order logic for formalization, and should apply the semantics of homogeneous set-membership to categorization relationships of this kind. If we reject the type = instance interpretation, we may stay with first-order logic (at least for formalizing categorization relationships of this kind), and may optionally distinguish such categorization relationships as special (which some practitioners feel is an important thing to do), and provide them with relevant formal semantics. From a meta-modeling viewpoint, it is trivial to include one or more meta-fact types to classify fact types to cover this case and others.

To note this distinction, one could adopt a special graphical or textual adornment for such categorization associations. For example, in ORM one might append a colon “:” to the forward reading of any predicate used for this purpose (based on the common use of colons to sometimes but not always introduce types). Applying this suggestion to the model in Figure 1(a) would replace “is of” by “is of:”.

To provide a minimal, common approach to such categorization relationships, whether or not we adopt the type = instance viewpoint, we could use the colon marker to distinguish any such relationship, and give it the semantics of an *asymmetric*, *intransitive*, and *locally-homogeneous* relationship. A fact type of the form $A R B$ is *locally-homogeneous* if and only if B is used as categorization scheme for A , but for no other type (so no other type bears a colon relationship to B). For the example in Figure 1, this means that only Account instances may be instances of AccountType instances. It is convenient to use the same predicate reading (e.g. “is of:”, or even just “:”) for all such categorization predicates, unless this makes the reading awkward. The choice of reading is language-dependent.

The properties of asymmetry and intransitivity seem to be the only properties of the set-membership operator (\in), that are relevant here. If we always use the same reading (e.g. “is of:”) for the categorization relationship, we may think of it as a predefined predicate constant that applies globally (all occurrences of this predicate have the same semantics). If we reserve the colon only for such homogeneous cases, we must not use it in cases where the classification scheme (e.g. Gender) may be applied to more than one type (e.g. Person, Dog).

Note that *any* role played by a type could be used as a basis for categorizing it. So the main reason for marking such is-of associations as categorization relationships is to enforce the formal properties of such relationships. We may now formalize categorization relationships of this kind (Figure 2), assuming the same predicate reading “is of:” for all (if this is not so, then without loss of generality, begin by replacing each reading by “is of:”). We use “ \sim ” for negation (“it is not the case that”), “ $\&$ ” for conjunction (“and”), and “ \rightarrow ” for material implication (“implies”). The asymmetric and intransitive properties may be declared independent of the object types, unlike local homogeneity.

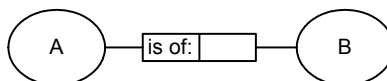


Figure 2 The categorization relationship is *asymmetric*, *intransitive*, and *locally-homogeneous*.

For the *first-order logic interpretation*, all types are first-order, so instances of B are individuals, not types. We use lower-case letters (possibly subscripted) to range over individuals. For our *higher-order logic interpretation*, we use capital letters (in italics) to denote type variables of any order. The order (1, 2,

3, ...) of any type is implicit, since it can be derived by inspecting the full schema². Ignoring the case of crossing meta-levels, assign the order of a type to be 1, plus the number of relationships in a contiguous chain of zero or more categorization relationships that end at the type.

Using $\exists!$ for Stephen Kleene’s “there exists exactly one” quantifier, we may now formalize the constraints on the categorization relationship in Figure 3, which has a mandatory and uniqueness constraint. The first-order formalization treats AccountType as a type of individuals. The higher-order formalization treats AccountType as a type of first-order types.

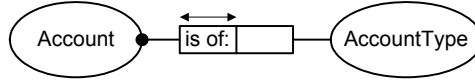


Figure 3 A categorization relationship with two additional constraints.

First-order formalization:

- $\forall xy [x \text{ is of: } y \rightarrow \sim (y \text{ is of: } x)]$ -- asymmetric
- $\forall xyz [x \text{ is of: } y \ \& \ y \text{ is of: } z \rightarrow \sim (x \text{ is of: } z)]$ -- intransitive
- $\forall xy [\text{AccountType } y \ \& \ x \text{ is of: } y \rightarrow \text{Account } x]$ -- local homogeneity
(only accounts are of account types)
- $\forall x [\text{Account } x \rightarrow \exists!y (\text{AccountType } y \ \& \ x \text{ is of: } y)]$ -- mandatory and unique constraints

Higher-order formalization:

- $\forall xY [x \text{ is of: } Y \rightarrow \sim (Y \text{ is of: } x)]$ -- asymmetric
- $\forall xYZ [x \text{ is of: } Y \ \& \ Y \text{ is of: } Z \rightarrow \sim (x \text{ is of: } Z)]$ -- intransitive
- $\forall xY [\text{AccountType } Y \ \& \ x \text{ is of: } Y \rightarrow \text{Account } x]$ -- local homogeneity
(only accounts are of account types)
- $\forall x [\text{Account } x \rightarrow \exists!Y (\text{AccountType } Y \ \& \ x \text{ is of: } Y)]$ -- mandatory and unique constraints

Alternatively, the higher-order formalization may replace any expression of the form α is of: β , where β is a type variable, by $\beta\alpha$, since it regards α is of: β in such cases to entail that α instantiates the β predicate. This leads to the higher-order formalization: $\forall xY (Yx \rightarrow \sim xY); . \forall xYZ (Yx \ \& \ ZY \rightarrow \sim Zx); \forall xY (\text{AccountType } Y \ \& \ Yx \rightarrow \text{Account } x); \forall x [\text{Account } x \rightarrow \exists!Y (\text{AccountType } Y \ \& \ Yx)]$.

UML Powertypes

One motivation for distinguishing such categorization relationships is to facilitate transformation to or from UML models that include so-called *powertypes*, which UML includes specifically to model such categorization schemes. Figure 4 shows a simplified version of an example often used to illustrate the need for powertypes, using the old UML 1.4 notation. This way of classifying trees is botanically wrong, but that’s irrelevant to the issue. Let’s assume that trees can be classified into object species such as Oak, Elm, and Willow in this simple way. Here TreeSpecies is a class analogous to the object type AccountType in Figure 1.

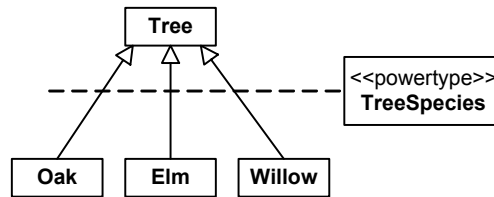


Figure 4 Powertype example in UML 1.4 notation (now retired).

² If it is desired to explicitly show the order of a type, a pre-superscript may be used (e.g. ²B indicates B is a second order type, i.e. its instances are types whose instances are individuals). Post-superscripts are typically used to denote arity, and post-subscripts are often used to distinguish variables of the same type.

To indicate that the subclasses Oak, Elm and Willow are each instances of the class TreeSpecies, a dashed line connects the inheritance links to TreeSpecies, which is marked with the stereotype name “powertype”. If the name “powertype” derives from the notion of power set (the power set of a set A is the set of all subsets of A), the term is misleading, as the powertype TreeSpecies excludes many instances in the power set of the set of trees (e.g. the null set, the set of all trees, and many other tree sets). For this reason, the term “higher-order type” seems more appropriate than “powertype”.

At any rate, this diagram does not explicitly include an association such as Tree is a member of TreeSpecies (analogous to Account is of AccountType). We may treat this association as implicit here, but in practice an explicit version of this association would typically be needed, since we would normally want to know the species of any given tree, and with hundreds of tree species it would be diagrammatically extravagant to introduce subtypes for all of them.

In UML 2.0, powertypes are defined within the superstructure [20, sec. 7.17]. The old stereotype notation has been retired [20, p. 597]. Instead, a colon “:” prepends the powertype name (e.g. “:TreeSpecies”) to annotate the collection of displayed subtype-supertype connections that belong to the set of all possible generalization relationships (called a GeneralizationSet) based on the categorization scheme provided by the association that relates the supertype to the powertype. If the subtypes are connected to the supertype using different arrowheads, the powertype annotation is placed next to a dashed line that crosses the relevant subtype connections (see Figure 5a). If a common arrowhead is used, the annotation is placed next to that (see Figure 5b).

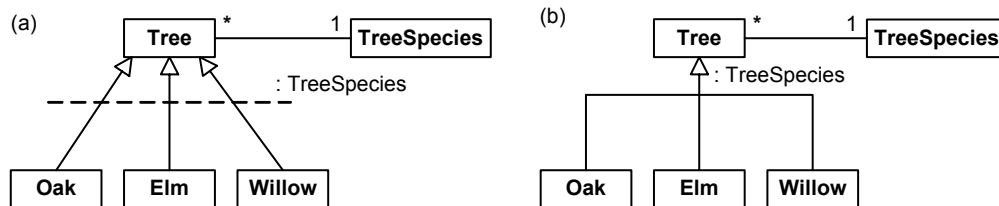


Figure 5 Powertype example in UML 2.0 notation.

Without the old stereotype notation, there seems to be no way to directly indicate that a class is a powertype. If this is the case, we can know that a class is a powertype only if its name used in a generalization set constraint. We may not assume that any binary association from an object type to a type marked in some way as a “powertype” is of this nature (moreover, UML 2.0 retired this stereotype). For example, in addition to Tree is a member of TreeSpecies we might have fact types like Person named TreeSpecies.

Figure 6 shows an example from the UML 2.0 specification [20, p. 127] that provides a different way to model a variation of our earlier account example. Here AccountType is modeled as a powertype rather than an enumeration, and the fact type Account is of AccountType is modeled as an association rather than as an attribute³. It is possible to include this association in the model *without explicitly introducing any subtypes* (SavingsAccount etc.). In that case, there is *no way to formally capture the categorization semantics of the association, or to declare that AccountType is a powertype*.

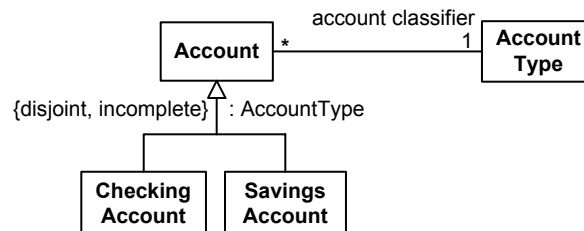


Figure 6 Powertype example from the UML 2.0 Superstructure specification (p. 127).

³ This example differs slightly in allowing more than two account types. If an enumeration constraint holds for AccountType (e.g. {SavingsAccount, CheckingAccount, LoanAccount}), it is unclear how to add this constraint to the powertype, except by resorting to a textual specification of the constraint (e.g. by using OCL).

While the role name “account classifier” *informally* suggests these additional semantics, this has no formal force (other examples in the UML specification use different role names such as “vehicle category” etc.). So *we need to distinguish the categorization association itself*, for example by appending a colon to the relationship reading (e.g. “is of:”) and/or the relevant association role names.

Even if generalization relationships are annotated with the relevant powertype name, this does not always formally guarantee the required semantics. For example, suppose we classify employees as part-time or full-time, and also record their preferred employment status (part-time or full-time) if any. We might model this in UML as shown in Figure 7, with two associations between Employee and EmployeeType. There is no formal way of knowing which of these two associations is used as the basis for membership in the subtypes. To solve this problem, we could require role names in the annotation, or adopt the ORM practice of supplying formal subtype definitions⁴.

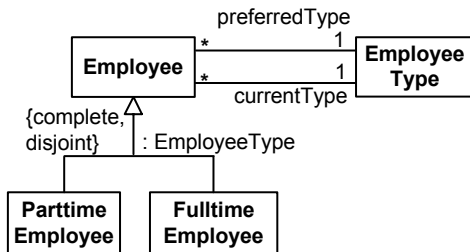


Figure 7 Powertypes do not guarantee an unambiguous classification scheme.

UML does allow generalization annotations to include names for the generalization set (presumably prepended to :powertype names when powertypes are involved, although the UML specification does not clarify this possibility). However such names are treated as informal comments. There is no formal requirement to link these back to properties (attributes or far-roles) of the supertype, as is done in those database modeling techniques that use discriminators to indicate a basis for subtyping.

3 What kind of higher-order logic is appropriate?

The current ORM formalization uses only first-order logic, basic arithmetic, and bag comprehension. Once we adopt higher-order logic, we need to allow any order (not just second-order), because in principle one might always introduce new types to categorize existing types, whether or not those types are first-order. If we adopt standard semantics for higher-order logic, where quantifiers may range over any imaginable predicates, we lose some useful properties of first-order logic. For example, completeness, compactness, and the Skolem-Löwenheim theorems don’t hold in standard second-order logic.

Moreover, care is needed to avoid some well known paradoxes. Russell’s paradox considers the set of all sets that are not members of themselves: is this set a member of itself? If Yes, then No, and if No then Yes, leading to a contradiction. Grelling’s paradox deals with self-predicable or autological properties (properties that apply to themselves). For example, we might argue that the property of being nonhuman is itself nonhuman (and hence is a self-predicable property) whereas the property of being human is not itself human (and hence is a non-self-predicable property). If predicates may instantiate themselves, then using *N* and *H* for the properties of being nonhuman and human we might formalize this as *NN* and $\sim HH$ respectively. But then what about the property of being non-self-predicable? Is this self-predicable or not? If it is, then it is not, and if it is not, then it is. Either way, we have a contradiction.

To avoid such paradoxes, Bertrand Russell developed a type theory in which types are ordered in a hierarchy, and it is meaningful to say that a type is an instance of another type only if the second type is on the next level of the hierarchy. Similarly, predicates of higher-order apply only to predicates or objects of lower orders. In particular, no predicate may apply to any predicate of the same order. Hence no predicate may apply to itself. Essentially the paradoxes are avoided by forbidding predicates to apply to themselves, by adopting a hierarchy of levels in which types can have instances at lower levels only.

⁴ The types Employee, ParttimeEmployee, and FulltimeEmployee are time-*deictic*, because their sense is determined in part by the time that their terms are uttered/inscribed [23, pp. 9-10; 17, esp. pp. 304, 312-13]. Deixis has a significant impact on information modeling, but space precludes a proper discussion here.

While this seems a reasonable approach to adopting higher-order logic, there are other versions of higher-order logic that do not take this approach. For example, the logic underlying the Knowledge Interchange Format (KIF) allows predicates to instantiate themselves, so expressions such as $(R R)$ are allowed [16]. An extension of this logic has been proposed as an ISO “Common Logic” standard to facilitate interchange of logical formulae between different knowledge representation tools [21]. Unfortunately, this Common Logic proposal includes a number of “bizarre” properties that make it unintuitive for direct use. Nevertheless, the common logic is very relaxed, and has mappings to several logics, so it seems quite feasible to have mappings between it and a more intuitive logic.

To make the implementation of higher-order logic more tractable, it seems best to adopt a non-standard semantics, similar to Henkin semantics [18, pp. 378-80], to limit the range of predicates/functions over which we may quantify, in order to retain useful properties of first-order logic (e.g. completeness). With standard semantics, a monadic first-order predicate may range over the power set of the domain of individuals (objects: lexical or non-lexical). To deal with categorization-types (e.g. AccountType) where we wish to assert that instances are types, it seems that the only extension we need beyond first-order logic is to allow quantification over object types that are instances of a declared categorization-type (whether or not these instances have been explicitly declared as a subtype).

As a separate decision, if we wish to allow crossing metalevels in the same model (see later), we should allow quantification over object types (primitive or derived), of any order, that are explicitly declared in the schema. If we do allow this, there seems to be no compelling case to allow quantification over polyadic predicates, so this relaxation may be regarded as a restricted case of Henkin semantics.

Given that a move to higher-order logic adds work to the formalization and implementation tasks⁵, is explicit support for quantification over predicates worth the extra effort? With respect to the categorization relationship, the only motivations seem to be in noting what kind of relationship it is, in providing direct support for those who view the relationship as an instance-type relationship, and in providing a convenient slot for mapping to/from powertypes in UML. With respect to categorization, these motivations seem non-compelling, given that a first-order interpretation seems reasonable, and formal subtype definitions (as in ORM) provide the connections between subtypes and the predicates and object types used to define them. The next section provides some justification for a first-order interpretation.

4 A first-order logic approach to categorization

Consider the ORM schema in Figure 8, which is a classic case where higher-order logic proponents would demand that CarModel is a second-order type, whose instances are subtypes of Car (e.g. FordFutura2004). When I think of an instance of Person and Car (concrete concepts), I’m thinking of an actual person or car. When I think of an instance of CarModel (an abstract concept), I think of an abstraction that is essentially a car design—a car *structure or specification* that might be denoted by a schematic diagram, for example.

For any given business domain, I define a *type* as a *set of possible instances, where for any given state of the business domain, exactly one subset of the type is the population of the type in that state*. At any given time, the *population* of a type is the set of instances of that type that exist in the business domain at that time. This definition is similar to Fitting’s notion of an intensional type as a function from possible worlds to extensions [8, p. 84], except that here a possible world corresponds to a state of business domain, sometimes called a “time-slice” or “instant state” in temporal logic [10, p. 143; 9, p. 121]. The temporal aspect is needed to distinguish between types that have the same *extension over time* but may differ in *extension at some time* (e.g. HumanPerson, and HumanBaby). Note the two senses of “extension”. A type’s extension at some time is its population at that time. A type’s extension over time is effectively the atemporal union of all its state-extensions, past, present, or future, which is fixed. “Type” conveys both state-dependent (variable extension) and state-independent (fixed extension) notions, not just set semantics.

By the Principle of Extensionality, sets are defined by their extension, i.e. given any sets A and B , we say that $A = B$ iff $\forall x(x \in A \equiv x \in B)$. So unless we resort to non-well-founded set theories, we must regard sets to be fixed—they cannot change over time. It seems clear that for any given business domain, the current population of a type (e.g. Person or Country) may change over time, but the type itself does not.

⁵ For a plea in favor of higher-order logic, and for some examples where first-order theorems are much easier to prove in higher-order logic see [1].

Given the above definition of type, I personally don't think of an instance of a car model as a set of possible cars of that model (a subtype). Each CarModel instance is in 1:1 correspondence with such a subtype (implicit or explicit), but it's not identical to a subtype. It seems to me that this distinction can always be made. If so, we can treat instances of such "categorization types" as ordinary individuals, that are not types, so first-order logic is enough.

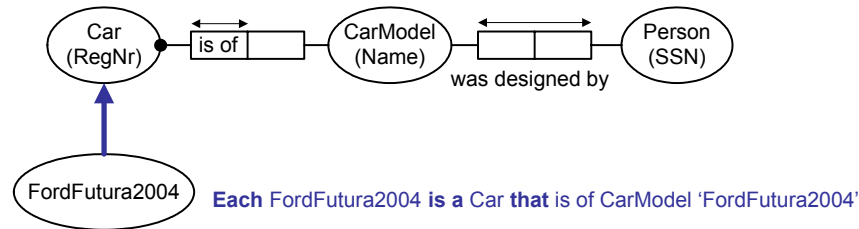


Figure 8 The population of a subtype (or any type) typically varies over time.

Suppose we explicitly introduce a car subtype called "FordFutura2004", as shown in Figure 8. The subtype definition provides the formal connection between the subtype and the car model instance. Using the standard reference mode semantics for ORM, the expression "CarModel 'FordFutura2004'" in the definition abbreviates "CarModel that has CarModelName 'FordFutura2004'". In ORM, a subtype is essentially a derived object type—its population is determined by applying the derivation rule in its definition to the populations of the object types referenced in the definition.

Suppose we add the existential fact **There exists a CarModel that** has CarModelName 'FordFutura2004' to the information base before any cars of that model are produced. At this time, the population of the car subtype FordFutura2004 is the null set. Suppose at a later time, 100 cars of that model are produced. At that later time, the population of the car subtype FordFutura2004 includes 100 cars. At any time, the subtype FordFutura2004 includes those 100 cars, as well as all other cars of that model that will ever be produced. So it's perfectly OK to regard an instance of CarModel to be in 1:1 correspondence with a set of (possible) cars. This is true whether or not we think of the car model instance as the set of possible cars of that model, or (as I do) as the fundamental car structure or design to which the car instances conform.

Hence it is reasonable to think of an instance of a "categorization type" such as CarModel or AccountType as an individual (e.g. a structural pattern) that is ontologically distinct from a type (in the sense of a set of possible instances). This seems sufficient to stay with first-order logic for such cases. Although the word "Type" in "AccountType" may suggest that its instances are themselves types, this stems more from an unfortunate naming choice for the type rather than from any fundamental intuition.

The second approach that allows one to avoid higher-order types for categorization is to *avoid uninformative categorization schemes*. The term "AccountType" is uninformative, because it does not provide any basis for categorizing accounts. In principle, any object type such as Account might be categorized in many different ways, leading to different types of bank account. For example, we could define an AccountKind {Local, National, International}, an AccountCategory {Taxable, Nontaxable}, and so on. These are all categorization schemes, which we may wish to use in the same model, and names such as "AccountType" and "AccountKind" don't inform us at all about the criterion used by a given categorization scheme to place accounts into account categories. One might argue that the value constraint placed on the categorization scheme provides this criterion, but this requires the modeler to induce the criterion based on his/her informal understanding of what the names for those values mean, an understanding that is not formally accessible to an automated system. More importantly, we may wish to introduce a categorization scheme without committing to a fixed set of instances.

As a pragmatic issue then, it seems reasonable to encourage the modeler to *choose informative names that reveal the basis for classification schemes*. If we adopt this approach, the type = instance issue typically disappears for the categorization case. In Figure 9 for example, the subtype Savings account is defined based on its primary function. The AccountFunction instance named 'Savings' and denoted by the shaded dot is clearly not identical to the subtype SavingsAccount (a function is not the same thing as a bank account). One may introduce other informative categorization schemes such as Account may be used in Region, etc. Similarly, our Car is of CarModel relationship might be renamed "Car conforms to CarModel".

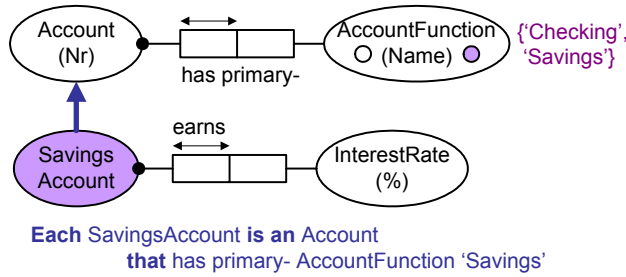


Figure 9 An informative categorization scheme explains the basis for categorization.

Any binary fact type used for an enumerated categorization scheme may be replaced by one or more unary fact types. For example, instead of Account is of AccountType {Checking, Savings}, we may use the fact types Account is used primary for savings and Account is used primarily for checking (the mandatory and uniqueness constraints are then captured by an xor constraint). Instead of Account is of AccountCategory {Taxable, Nontaxable}, we may use the fact type Account is taxable, applying the closed world assumption to determine nontaxable accounts. This is yet another way to avoid higher-order types for enumerated categorization schemes.

In many cases, *an object type may be used to categorize more than one kind of object*. For example, Figure 10 includes two categorization fact types Person is of Gender and Animal is of Gender, where Gender has two possible instances identified by the gender codes ‘M’ (for male gender) and ‘F’ for female gender). Here the semantics of the categorization fact types does not involve homogeneous set membership. Clearly, instances of Gender (MaleGender, FemaleGender) may not be identified with any of the four subtypes shown. This kind of categorization scheme is quite common, and clearly excludes any type = instance identities.

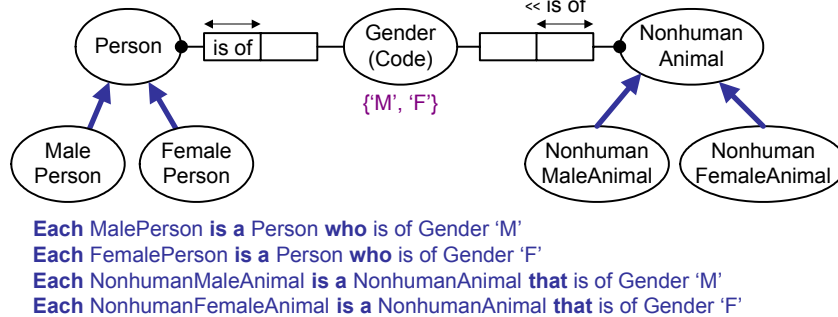


Figure 10 A typical case where instances of the categorization scheme are not identified with subtypes.

The use of Gender in the above model to define the subtypes seems better than using two powertypes PersonType {MalePerson, FemalePerson} and NonhumanAnimalType {NonhumanMaleAnimal, NonhumanFemaleAnimal}. Note that while UML allows this case to be modeled using Gender as an attribute or non-powertype, any use of a generalization set name (e.g. ‘gender’) is merely an informal comment, with no formal connection to the model element used as the basis of the categorization.

5 Set-structures, and metalevel crossing

Consider a categorization scheme whose instances intuitively correspond to set-like containers. Figure 11 includes a model that, apart from some constraints, is structurally similar to the account example in Figure 1. In this business domain, only A-team members may earn privileges, so a subtype is created to record facts of this nature. The shaded dot denotes team A and the white dot denotes team B. The shaded subtype A-TeamMember is the type whose population at any time is the set of A-team members at that time.

It seems natural to think of Team A at any point in time as more than just a set of people. The concept of a team brings in other semantics (a social unit whose members work together for a common purpose). While the thing denoted by the shaded dot appears to have this additional informal semantics, the subtype A-TeamMember does not—at any point in time its denotation seems to be no more than a set of people who just happen to be members of team A. If these intuitions are correct, and team membership is considered to be a categorization scheme, here is another example where it is reasonable not to identify the subtype with the instance of the categorization scheme.

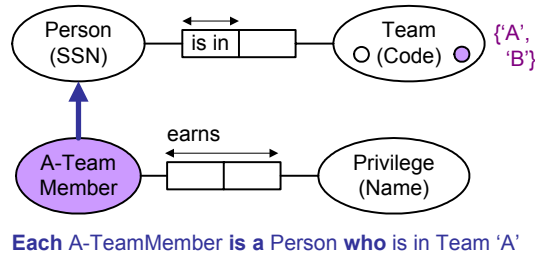


Figure 11 Is the Team instance denoted by the shaded dot identical to the subtype A-TeamMember?

Now consider Table 1, which is a slightly simplified version of an example by Fitting [7], used to motivate a formalization of databases that uses higher-order modal logic. The table has two aspects that are unusual. First, it is in non-first normal form, allowing unnamed sets as entries (e.g. ColorChoices). This is permitted in some object-relational databases. Secondly, its final attribute (column) allows as entries unnamed sets whose instances appear to be attributes themselves, thus crossing levels/metalevels.

Table 1 Table with entries that may be sets of individuals or attributes.

Car	CarModel	ColorChoices	AirConditioning	CustomerChosen
1	Ford Escape 2003	{red, green, black}	Yes	{ColorChoices}
2	Ford Escape 2003	{red, green, black}	No	{}
3	Mazda MPV 2004	{green, sand-mica}	Yes	{ColorChoices, AirConditioning}

Fitting’s formalization of this situation is higher-order, as he treats the structure directly as it stands. The price paid for this directness is deep complexity, and an implementation nightmare. These disadvantages can be avoided by transformation into a first-order model that is cleaner and pragmatically easier to implement. In practice, it would be realistic to record the color chosen for a car. With this additional fact type, and omitting for now the air-conditioning aspects of the model, the situation may be modeled by the ORM schema shown in Figure 12. Here the color choice sets are handled in the usual normalized way, with a many:many association. The subset constraint (circled “ \subseteq ”) ensures that colors chosen for a car belong to those available for its car model. The fact whether a customer chose the color for a car is catered for by instantiating the unary predicate applied to the objectified CarColor association. The airconditioning aspects can be catered for in a similar way. For a simpler example of level-crossing, see [6, p. 20].

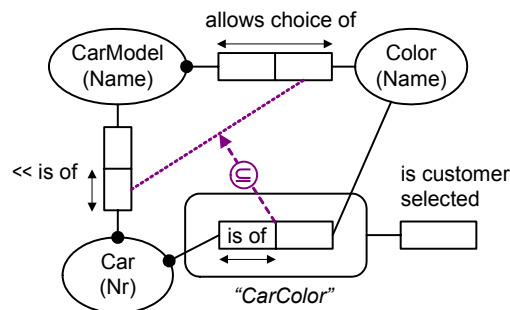


Figure 12 A first-order solution for part of the un-normalized table.

In cases where there are many attributes about which information is to be recorded, and the attributes are not all known in advance, this may be modeled by introducing Attribute as a first-order type, along with fact types that record its name and value. By thus demoting meta-data to data, we remain at first-order. This technique is well-known and quite effective in practice.

A final argument for using higher-order types is to allow expression of business rules that appear to cross levels/metalevels (or ignore level distinctions) in the same model (e.g. the Finance department is responsible for defining the possible values of AccountType). If somebody really wants to formalize such cases directly, then higher-order types are clearly needed. However, as a pragmatic alternative, it is usually possible to handle such rules in a first-order way, either by separating the meta-aspects into a separate first-order model where the former meta-types are now ground types, or by demoting meta-data as data, or simply ignoring the cross-level identities.

As a simple example, we might build into the core package of the metaschema such metafact types as FactRole is played by ObjectType, BusinessRule has IllocutionaryForce etc, while in the management package of the metaschema we include metafact types dealing with aspects of security and authorization etc., e.g. UserGroup has AccessRight to FactType. Populating the latter metafact types in the metamodel may then allow us to add rules of the desired kind without crossing metalevels.

For example, consider the simple business model in Figure 13. Here there are two elementary fact types (F2 and F4), and three existential fact types (F1, F3, and F5) depicted in abbreviated form using parenthesized reference modes. In this case, the current values of AccountType are stored in a reference table instead of being rigidly declared in a value type constraint. One of our business rules is that only the marketing department may choose what the account type instances may be.

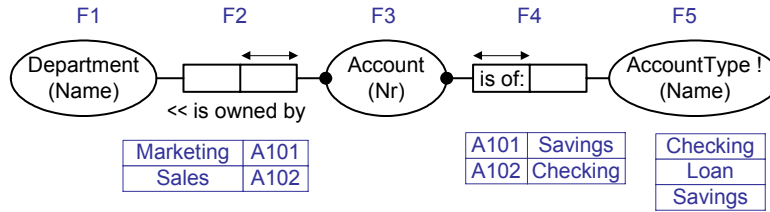


Figure 13 A populated business model with 5 fact types (3 existential and 2 elementary).

With this approach, the one column fact table for AccountType may be treated just like any other fact table in the model, including the way in which we determine who has what kind of access to what fact type. A simplified metafragment for dealing with security is shown in Figure 14. The business rule mentioned earlier is now handled by populating this metafact type as shown.

The metamodel in Figure 14 exists at a level above the business model in Figure 13. Each of these models can be formalized separately, using either first-order logic or the higher-order logic extension for categorization discussed earlier. What is missing from this picture is the ability to identify the marketing department mentioned in the business model with the marketing department that appears as a user group in the metamodel. There does not appear to be any compelling business case to require formal support of this identity, as businesses seem to run perfectly well without such support (e.g. in SQL systems, the application tables and metatables are typically accessed separately).

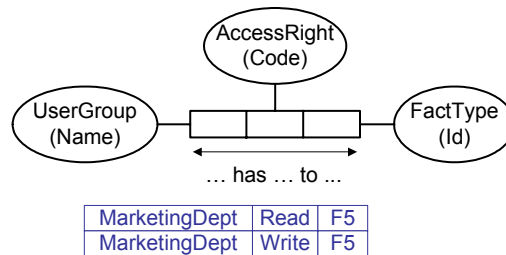


Figure 14 A model fragment from the management package of the metamodel.

6 Conclusion

This paper addressed issues relating to the necessity or otherwise of including higher-order types to deal adequately with three aspects of information modeling: categorization schemes, un-normalized structures, and crossing levels/metalevels within the same model. It argued that higher-order types may be used, so long as the underlying higher-order logic retains certain important first-order properties (e.g. by adopting Henkin semantics). It also suggested a number of ways in which higher-order types may be avoided, by treating types as intensional objects whose instances may sometimes be in 1:1 correspondence (but not identical) to subtypes, by requiring subtype definitions to be informative, by remodeling (including demotion of metadata to data), and by treating types as individuals in separate models.

Various formal properties were established for those categorization types that on first glance appear to require a higher-order solution, and a number of deficiencies were identified in the current treatment of powertypes within UML.

Related future research will investigate the detailed impact of deixis in information modeling, and how issues of semantic equivalence and co-extension are affected by one's stance with regard to the absolute or relative nature of types. For example, when we use terms for object types (e.g. Person, Gender, Country) in modeling, are we thinking only about a given business domain, or a more general concept of the type?

Acknowledgement: The presentation of some ideas in this paper has benefited from discussion with Andy Carver of Northface University, and Don Baisley of Unisys Corporation.

References

1. Andrews, P. B. 2002, *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*, Kluwer Academic Publishers, Dordrecht.
2. Bakema, G., Zwart, J. & van der Lek, H. 1994, 'Fully Communication Oriented NIAM', *NIAM-ISDM 1994 Conf. Working Papers*, eds G. M. Nijssen & J. Sharp, Albuquerque, NM, pp. L1-35.
3. Chen, P. P. 1976, 'The entity-relationship model—towards a unified view of data'. *ACM Transactions on Database Systems*, 1(1), pp. 9–36.
4. Codd, E. F. 1970, 'A Relational Model of Data for Large Shared Data Banks', *CACM*, vol. 13, no. 6, pp. 377-87.
5. De Troyer, O. & Meersman, R. 1995, 'A Logic Framework for a Semantics of Object Oriented Data Modeling', *OOER '95, Proc. 14th International ER Conference*, Gold Coast, Australia, Springer LNCS 1021, pp. 238-249.
6. Fitting, M. 2000, 'Modality and Databases', *Automated Reasoning with Analytic Tableaux and Related Methods*, Springer Lecture Notes in Artificial Intelligence 1847, Roy Dyckhoff (ed), pp 19--39, 2000. [© Springer-Verlag, URL: <http://www.springer.de/comp/lncs/index.html>]
7. Fitting, M. 2000, 'Databases and Higher Types', *Computational Logic --- CL2000*, Springer Lecture Notes in Artificial Intelligence 1861, John Lloyd et. al. (ed), pp 41--52, 2000. [© Springer-Verlag, URL: <http://www.springer.de/comp/lncs/index.html>]
8. Fitting, M. 2002, *Types, Tableaus, and Gödel's God*, Kluwer Academic Publishers, Dordrecht.
9. Girle, R. 2000, *Modal Logics and Philosophy*, McGill-Queen's University Press, Montreal & Kingston.
10. Girle, R. 2003, *Possible Worlds*, McGill-Queen's University Press, Montreal & Kingston.
11. Halpin, T. A. 1989, 'A Logical Analysis of Information Systems: static aspects of the data-oriented perspective', doctoral dissertation, University of Queensland.
12. Halpin, T. A. 1998, 'ORM/NIAM Object-Role Modeling', *Handbook on Information Systems Architectures*, eds P. Bernus, K. Mertins & G. Schmidt, Springer-Verlag, Berlin, pp. 81-101.
13. Halpin, T. A. 2001, *Information Modeling and Relational Databases*, Morgan Kaufmann, San Francisco.
14. Halpin, T. A. 2002, 'Metaschemas for ER, ORM and UML: A Comparison', *Journal of Database Management*, Idea Group Publishing, Hershey PA, pp. 4-13.
15. ter Hofstede, A. H. M., Proper, H. A. & Weide, th. P. van der 1993, 'Formal definition of a conceptual language for the description and manipulation of information models', *Information Systems*, vol. 18, no. 7, pp. 489-523.
16. Hayes, P. & Menzel, C., 'A Semantics for Knowledge Interchange Format', *Proceedings of 2001 Workshop on the IEEE Standard Upper Ontology*, August 2001. The paper itself is available online at: <http://reliant.tekknowledge.com/IJCAI01/HayesMenzel-SKIF-IJCAI2001.pdf>.
17. Lyons, J. 1995, *Linguistic Semantics: An Introduction*, Cambridge University Press: Cambridge, UK.
18. Mendelson, E. 1997, *Introduction to Mathematical Logic*, Chapman & Hall/CRC: Boca Raton.
19. Object Management Group 2003, *UML 2.0 Infrastructure Specification*. URL: www.omg.org/uml.
20. Object Management Group 2003, *UML 2.0 Superstructure Specification*. URL: www.omg.org/uml.
21. Menzel, C. 'The Common Logic Standard', Online at: <http://cl.tamu.edu/CL-ISO.pdf>.
22. Rumbaugh J., Jacobson, I. & Booch, G. 1999, *The Unified Language Reference Manual*, Addison-Wesley, Reading, MA.
23. Thomas, J. 1995, *Meaning in Interaction: An Introduction to Pragmatics*, Longman: London.

Conceptual Queries

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

This paper first appeared in vol. 26, no. 2 of Database Newsletter and is reproduced by permission.

Formulating non-trivial queries in relational languages such as SQL or QBE can prove daunting to end users. ConQuer, a new conceptual query language based on Object Role Modeling (ORM), enables users to pose complex queries in a readily understandable way, without needing to know how the information is stored in the underlying database. This article highlights the advantages of conceptual query languages such as ConQuer over traditional query languages for specifying queries and business rules.

Four query levels

There are four main levels at which humans may communicate with an information system:

- External
- Conceptual
- Logical
- Physical

The *external* level deals with the actual interfaces and input/output representations used to work directly with the system (e.g. screen forms and printed reports). At the *conceptual* level, the information is expressed in its most fundamental form, using concepts and language familiar to the users (e.g. Employee drives Car) and ignoring implementation and external presentation aspects. At the *logical* level, a commitment is made to the general type of data model to be used for storage (e.g. relational or object-oriented) and the information is expressed using the logical constructs of that model (e.g. tables and keys). At the *physical* level, a specific DBMS is chosen (e.g. MS Access or DB2) and all the detailed internal details are fleshed out (e.g. indexes and clustering).

Since a conceptual schema expresses the structure of an application from a human rather than a machine perspective, it facilitates communication between modeler and subject matter experts during the modeling process, and it can be mapped automatically to a variety of DBMS structures. Although software tools are often used for conceptual modeling and mapping, they are rarely used for querying the conceptual model directly. Instead, queries are typically formulated either at the external level using forms, or at the logical level using a language such as SQL or QBE.

Query-By-Form (QBF) enables users to enter queries directly on a screen form, by entering appropriate values or conditions in the form fields. This form-based interface is

well suited to simple queries where the scope of the query is visible on a single form, and no complex operations are involved. However this cannot be used to express complicated queries. Moreover, saved QBF queries may rapidly become obsolete as the external interface evolves. For such reasons, QBF is too restrictive for serious work.

For relational databases, SQL and QBE (Query-By-Example) are more expressive. However, complex queries and even queries that are easy to express in natural language (e.g. who does not speak more than one language?) can be difficult for non-technical users to express in these languages. Moreover, an SQL or QBE query often needs to be changed if the relevant part of the conceptual schema or internal schema is changed, even if the meaning of the query is unaltered. Finally, relational query optimizers ignore many semantic optimization opportunities arising from knowledge of constraints.

Logical query languages for post-relational DBMSs (e.g. object-oriented and object-relational) suffer similar problems. Their additional structures (e.g. sets, arrays, bags and lists) often lead to greater complexity in both user formulation and system optimization. For example, OQL [3] extends SQL with various functions for navigation as well as composing and flattening structures, thus forcing the user to deal directly with the way the information is stored internally. At the physical level, programming languages may be used to access the internal structures directly (e.g. using pointers and records), but this very low level approach to query formulation is totally unsuitable for end users.

Conceptual query languages

Given the disadvantages of query formulation at the external, logical or physical level, it is not surprising that many *conceptual query languages* have been proposed to allow users to formulate queries directly on the conceptual schema itself [1, 2]. Most of these language proposals are academic research topics, with at best prototype tool support. One commercial tool, English Wizard, provides some ability for users to enter queries directly in English, but the tool currently suffers from problems with ambiguity and expressibility, as well as the correctness of its SQL generation. By and large, current conceptual query language tools based on Entity-Relationship (ER) or deductive models are challenging for naïve users, and their use of attributes exposes their queries to instability, since attributes may evolve into entities or relationships as the application model evolves.

This instability is avoided by using a query language based on Object Role Modeling (ORM), a conceptual modeling approach that pictures the application world in terms of objects that play roles (individually or in relationships), thus avoiding the notion of attribute. ORM facilitates detailed information modeling since it is linguistically based, is semantically rich and its notations are easily populated. An overview of ORM may be found in [7] and a detailed treatment in [5].

The use of ORM for conceptual and relational database design is becoming more popular, partly because of the spread of ORM-based modeling tools. However, as with ER, the use of ORM for conceptual queries is still in its infancy. The first significant ORM-based query language was RIDL [9], a hybrid language with both declarative and procedural aspects. Although RIDL is very powerful, its advanced features are not easy to

master, and while the modeling component was implemented in the RIDL* tool, the query component was not supported. Another ORM query language is LISA-D [8]; although very expressive, it is technically challenging for end users, and currently lacks tool support.

Like ORM, the OSM (Object-oriented Systems Modeling) approach avoids the use of attributes as a base construct. An academic prototype has been developed for graphical query language OSM-QL [4] based on this approach. For any given query, the user selects the relevant part of the conceptual schema, and then annotates the resulting subschema diagram with the relevant restrictions to formulate the query. Negation is handled by adding a frequency constraint of “0”, and disjunction is introduced by means of a subtype-union operator. Projection is accomplished by clicking on the relevant object nodes and then on a mark-for-output button.

Another recent ORM query language is ConQuer (the name derives from “CONceptual QUERY”). ConQuer is more expressive than OSM-QL [2], easier for novice users, and its commercial tool transforms conceptual queries into SQL queries for a variety of back-end DBMSs. Moreover, the ConQuer tool does not require the user to be familiar with the conceptual schema or the ORM diagram notation. The first version of ConQuer was released in InfoAssistant [1]. Feedback from this release led to the redesign of both the language and the user interface for greater expressibility and usability, resulting in a new tool called ActiveQuery [2], a restricted version of which is available as an OLE control for Windows applications. As well as complying with Microsoft’s user interface standards, the tool provides an intuitive interface for constructing almost any query that might arise in an industrial database setting. Typical queries can be constructed by just clicking on objects with the mouse, and adding conditions.

The rest of this paper suggests some design principles for conceptual query languages, and then illustrates how these principles are realized in ConQuer, as supported by ActiveQuery. A brief outline of the underlying ORM framework is included, as well as examples of how queries are formulated and mapped to SQL. Finally some examples are given of how the query language can also be used to provide high level declaration of business rules.

Language design criteria

The following four criteria were used in designing the ConQuer language and tool support, and seem appropriate for conceptual query languages in general.

- Semantic strength
- Semantic clarity
- Semantic relevance
- Semantic stability

Semantic strength is a measure of a language’s expressibility (i.e. the range of queries that can be expressed in the language). Ideally, the language should allow you to formulate any question that is relevant to your application. In practice, something less

than this ideal is acceptable. For most business applications, if the language can express whatever is possible to formulate as a sequence of SQL-89 queries, this is often good enough. In more complex cases, this might not be adequate. For example, a bill of materials query requires recursion, which while supported by recursive union in the long awaited SQL3 standard, is still not available in many commercial SQL dialects. ActiveQuery was designed to translate ConQuer queries into a sequence of SQL statements on the back end DBMS, and hence is limited in practice by the power of the chosen SQL back end. In comparison with the low expressive power of QBF however, this is a very mild limitation.

Semantic clarity is a measure of how easy it is to understand and use the language. To begin with, the language must be unambiguous (i.e. there is only one possible meaning). Since any ConQuer query corresponds to a qualified path through an ORM schema, where all the object types and predicates are well defined, the meaning of the query is essentially transparent. As we discuss shortly, the ActiveQuery tool automatically reveals the relevant part of the application to the user, so that ConQuer queries can be formulated without requiring any prior knowledge of the information space. Although this context revelation is a feature of the tool rather than the language, even a manual formulation of ConQuer queries requires no sub-conceptual knowledge (e.g. knowledge about how the information is actually stored in a database). This is in sharp contrast to a query language such as SQL, QBE or OQL, where the query needs to be formulated in terms of the storage structures themselves.

Semantic relevance requires that only the information relevant to the intent of the query needs to be stated. In order to formulate a query, the user must not be forced to include other features of the application that have no bearing on the question that he or she wants to ask.

Semantic stability is a measure of how well queries retain their original intent in the face of changes to the application. Because ConQuer queries are based on ORM, they continue to produce the desired result so long as their meaning endures. In other words, you never need to change a ConQuer query if the English meaning of the question still applies. In particular, ConQuer queries are not impacted by typical changes to an application, such as addition of new fact types or changes to constraints or the relative importance of some feature. This ensures *semantic independence* (i.e. the conceptual queries are independent of changes to underlying structures when those changes have no effect on meaning).

If the discussion so far seems pretty abstract or hard to follow, it should all become clear with a few examples. The rest of the article is mainly concerned with illustrating these four design criteria with sample queries based on a small application. The underlying ORM schema for this application is explained in the next section.

A sample ORM schema

Although knowledge of the ORM diagram notation is not needed to formulate ConQuer queries, some familiarity with it will help you to understand the basis of the query

technology. A ConQuer query can be applied only to an ORM schema. Using a software tool, an ORM schema may be entered directly, or instead reverse engineered from an existing logical schema (e.g. a relational or object-relational schema). While reverse engineering is automatic, some refinement by a human improves the readability (e.g. the default names generated for predicates are not always as natural as a human can supply).

ORM is a bit like ER without attributes. If you are familiar with ER, just use a relationship instead whenever tempted to portray some feature as an attribute, and you have the basic ORM view of the world as a set of objects playing roles (parts in relationships).

Figure 1 is an ORM conceptual schema fragment for an application about a company with branches in various countries. Object types are shown as named ellipses. Entity types have solid ellipses with their simple reference schemes abbreviated in parenthesis (these references are often unabbreviated in queries). For example, “Car (regnr)” abbreviates “Car is identified by RegNr”.

If an entity type has a compound reference scheme, this may be shown explicitly using an external uniqueness constraint (circled “u”). In our example, a City is identified by combining its Cityname and State (which in turn is identified by combining its Statecode and Country). For instance, I live in a city called “Bellevue” that is in Washington state; this state is in the country named “USA” and has the statecode “WA” (whereas Western Australia is in the country named “Australia” and has the statecode “WA”). Value types have dotted ellipses (e.g. “Statecode”), and a “+” indicates numeric reference.

A predicate is a sentence with holes in it for object-terms. Predicates are shown as named role sequences, where each role is depicted as a box. A role is just a part in a relationship. In the example, all the relationships are binary (two roles) except for the ternary (three role): USBranch achieved Rank in Year. Predicates may have any arity (number of roles) and may be written in mixfix form (i.e. the object holes may be mixed in at any position within the predicate— this is essential for languages like Japanese where the verb comes at the end). A relationship type not used for primary reference is a fact type. An n -ary relationship type has $n!$ readings, but only n are needed to guarantee access from any object type. Figure 1 shows forward and inverse readings (separated by “/” if needed) for some of the relationships.

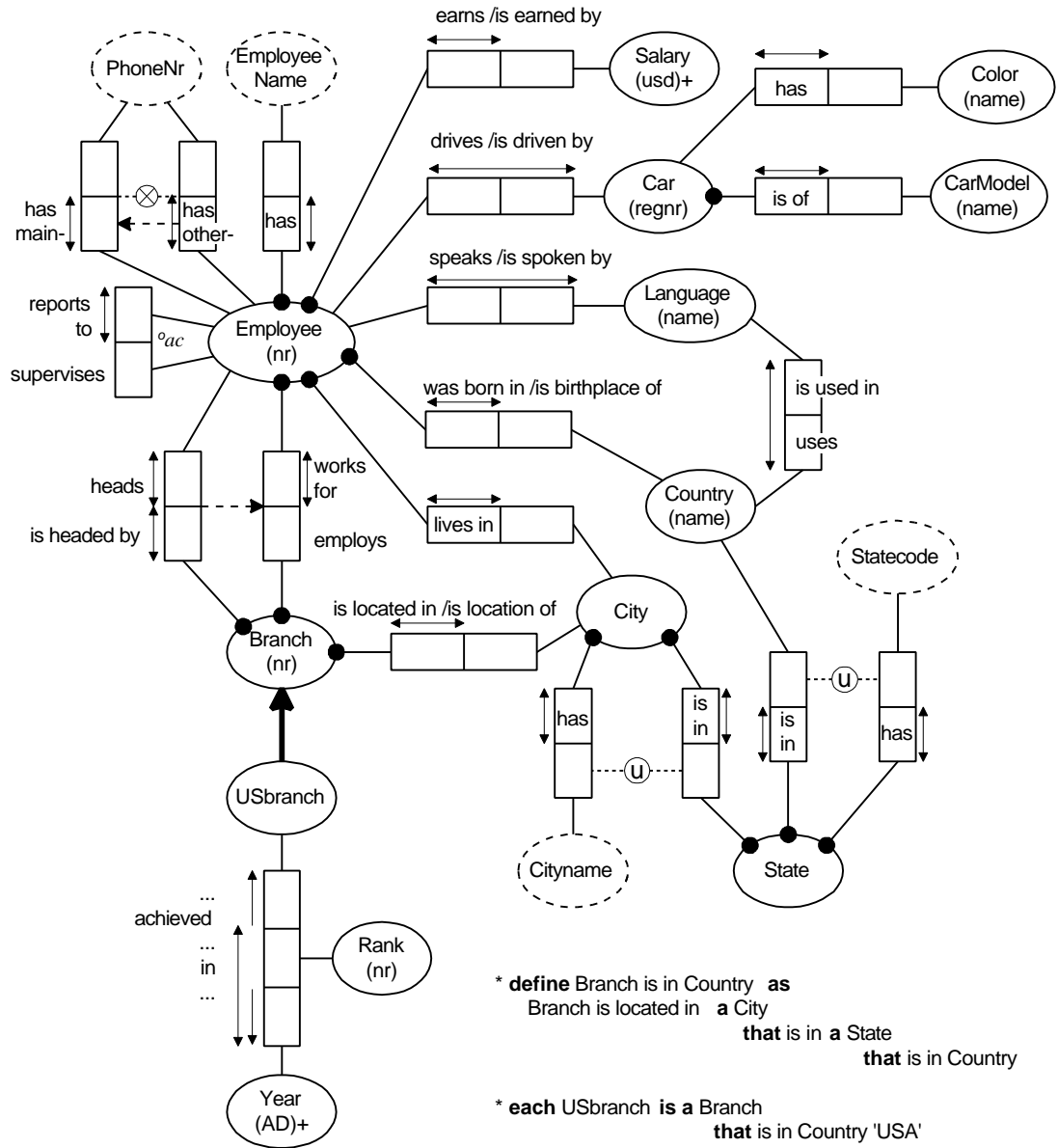


FIGURE 1: A sample ORM conceptual schema.

An arrow-tipped bar across a role sequence depicts an internal uniqueness constraint. For example, each employee has at most branch, but the same branch may employ more than one employee. The ternary has two uniqueness constraints: the right-hand one declares that a USbranch may achieve at most one rank in a given year; the left-hand one states that a given rank in a given year is achieved by at most one USbranch (i.e. no ties).

A black dot connecting a role to an object type indicates that the role is mandatory (i.e. each object in the database population of that object type must play that role). Subtypes are connected to their supertype(s) by arrows, and given formal definitions. Here we have only one subtype (USbranch). The two asterisked rules at the bottom of the figure declare

a derived fact type, and a subtype definition: these textual rules are essentially ConQuer queries.

The circled “X” in the top right corner is a pair-exclusion constraint (an employee’s main phone number must differ from his/her other phone number). The dotted arrow just below the exclusion constraint is a simple subset constraint (if an employee has another phone number, he/she must have a main phone number). The dotted arrow from the heads predicate to the works-for predicate is a pair-subset constraint (each employee who heads a branch also works for that branch). The “⁰ac” constraint on the reporting relationship indicates this relationship is acyclic (no loops back to itself). ORM has other kinds of constraint not shown here. InfoModeler’s verbalization ability allows schemas to be entered or output in English sentences, so that it is not necessary to understand the diagram notation.

Sample ConQuer queries and SQL mapping

Although ConQuer queries are based on ORM, users don’t need to be familiar with ORM or its notation. A ConQuer query is set out in textual (outline) form (basically as a tree of predicates connecting objects) with the underlying constraints hidden, since they have no impact on the meaning of the query.

With ActiveQuery, a user can construct a query without any prior knowledge of the schema. On opening a model for browsing, the user is presented with an object pick list. When an object type is dragged to the query pane, another pane displays the roles played by that object in the model. The user drags over those relationships of interest. Clicking an object type within one of these relationships causes its roles to be displayed, and the user may drag over those of interest, and so on. In this way, users may quickly declare a query path through the information space, without any prior knowledge of the underlying data structures. Users may also initially drag across several object types. The structure of the underlying model is then used to automatically infer a reasonable path through the information space (this Point-to-point query feature is ignored for the remainder of this article).

Items to be displayed are indicated with a tick “✓”: these ticks may be toggled on/off as desired. The query path may be restricted in various ways by use of operators and conditions. As a simple example, consider the query: List each employee who lives in the city that is the location of branch 52. This may be set out in ConQuer thus:

```
Q1 ✓Employee
    +-lives in City
        +- is location of Branch 52
```

This implicit form of the query may be expanded to reveal the reference schemes (e.g. EmployeeNr, BranchNr), and an equals sign may be included before “52”. For most users, the meaning of a ConQuer query should be clear enough (*semantic clarity*). ActiveQuery also generates an English verbalization of the query in case there is any doubt.

Since ORM conceptual object types are semantic domains, they act as semantic “glue” to connect the schema. This facilitates not only strong typing but also query navigation through the information space, enabling joins to be visualized in the most natural way. Notice how City is used as a join object type for this query. If attributes were used instead, we would typically have to formulate this in a more cumbersome way. If composite attributes are allowed we might use: List Employee.employeenr where Employee.city = Branch.city and Branch.branchnr = 52. If not, we might resort to: List Employee.employeenr where Employee.cityname = Branch.cityname and Employee.statecode = Branch.statecode and Employee.country = Branch.country and Branch.branchnr = 52. Apart from awkwardness, both of these attribute-based approaches violate the principle of *semantic relevance*. Since the identification scheme of City is not relevant to the question, the user should not be forced to deal explicitly with these details.

Even if we had a tool that allowed us to formulate queries directly in ER or OO models, and this tool displayed the attributes of the current object type for possible assimilation into the query (similar to the way Active query displays the roles of the highlighted object type), this would not expose immediate connections in the way that ORM does. For example, inspecting Employee.city does not tell us that there is some connection to Branch.city. The only way to do this is to use the domains themselves as a basis for connectedness, and this is one of the distinguishing features of ORM.

To illustrate other features of the query technology, it will help to show some SQL code that can be automatically generated from ConQuer queries. Using the Rmap algorithm [5], our conceptual schema maps to the relational schema shown in Figure 2 (for simplicity, domains are omitted). SQL queries apply to this.

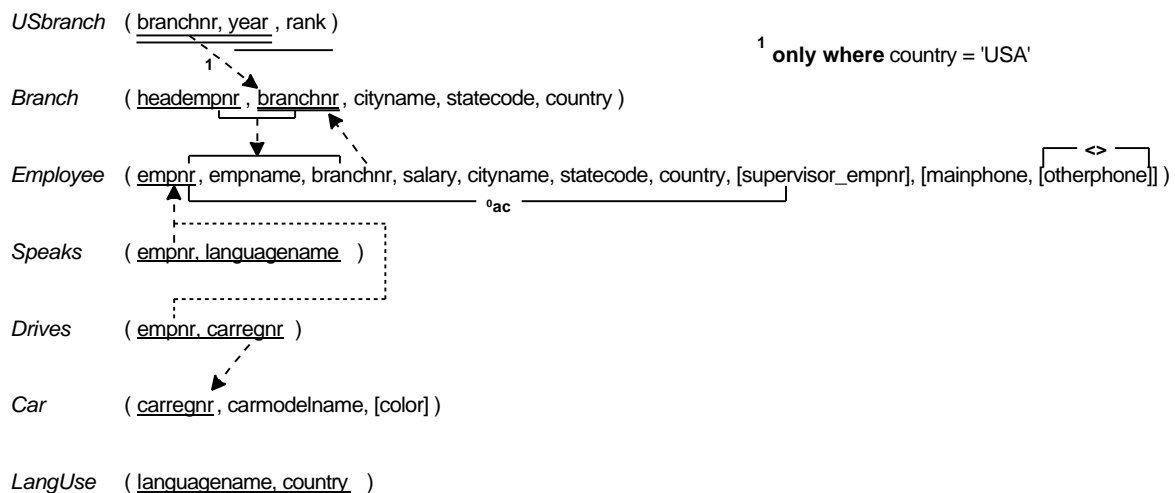


FIGURE 2: The relational schema mapped from the ORM schema of Figure 1.

In Figure 2, keys and uniqueness constraints are indicated by underlining (primary keys are doubly underlined where alternate keys exist). Optional columns are shown in square brackets. A subset constraint (e.g. foreign key constraint) is shown as a dotted

arrow. Here the numbered qualification 1 enforces the subtype definition. For more about subtyping in ORM, see [6].

ActiveQuery maps ConQuer queries to SQL for a variety of DBMSs, in the process performing semantic optimization where possible by accessing the constraints in the ORM schema. SQL code for query Q1 is shown below (S1). Notice how the conceptual query shields the user from details about City's composite reference scheme. Moving through an object type corresponds to a conceptual join. Here the relational join is a result of the same city playing two roles that map to separate tables, with no foreign key connection. In contrast to this semantic domain approach, some query tools require foreign keys to perform a join, and even force the user to specify what kind of join (e.g. inner or outer) is associated with a foreign key connection: the limitations of such an approach are obvious.

```
S1      select X1.empnr
        from Employee as X1, Branch as X2
        where X1.cityname = X2.cityname
           and X1.statecode = X2.statecode
           and X1.country = X2.country
           and X2.branchnr = 52
```

ORM makes no use of attributes in base models. This helps with natural verbalization, simplifies the framework, avoids arbitrary or temporary decisions about whether some feature should be modeled as an attribute, and lengthens the lifetime of conceptual queries since they are not impacted when a feature is remodeled as a relationship or attribute. This *semantic stability* of ORM models, and hence ORM queries, gives it a major advantage over ER, OO and lower level approaches.

For example, suppose that after storing the previous query, we change the schema to allow an employee to live in more than one city (e.g. a contractor might live in two cities). The uniqueness constraint on Employee lives in City is now weakened, so that this fact type is now many:many. With most versions of ER, this means the fact can no longer be modeled as an attribute of Employee.

Moreover, suppose that we now decide to record the population of cities. In ER or OO this would require that City be remodeled as an entity type instead of as an attribute. Hence an ER or OO based query would need to be reformulated. With ORM based queries however, the original query can still be used, since changing a constraint or adding a new fact type has no impact on it. Of course, the SQL generated by the ORM query may well differ with the new schema, but the meaning of the query is unchanged.

As an even simpler example, suppose we wanted to list employee drivers and their branches. In ConQuer we have:

```
Q2  ✓Employee
     +- drives Car
     +- works for ✓Branch
```

With our current schema, employees may drive many cars but work for at most one branch. So the information is spread over two relational tables, and the SQL code is:

```
S2a  select X1.empnr, X1.branchnr  
  
      from Employee as X1, Drives as X2  
  
      where X1.empnr = X2.empnr
```

However suppose that in an earlier version of our schema, employees could drive at most one car. In that case, all the information is in one table and the SQL code is:

```
S2b  select X1.empnr, X1.branchnr  
  
      from Employee as X1  
  
      where X1.carregnr is not null
```

Not only is this code sub-conceptual (null values are an implementation detail) but it is unstable, since a simple change to a conceptual constraint on the driving relationship requires the code to be changed as well. If we now relaxed our schema to allow employees to work for more than one branch (e.g. contract employees) the SQL code would need to be changed again since an extra table is needed to store the works relationship. In all these cases, the ConQuer query stays valid: all that changes is the SQL code that gets automatically generated from the query.

An OO query approach is often more problematic than an ER or relational approach, because there are many extra choices on how facts are grouped into structures, and the user is exposed to these structures in order to ask a question. Moreover these structures may change drastically to maintain performance as the business application evolves.

In the real world, changes often occur to an application, and work is required to cater for the changes in the database structures. Even more work is required to modify the code for stored queries. If we are working at the logical level, the maintenance effort can be very significant. We can minimize the impact of change to both models and queries by working in ORM at the conceptual level and letting a tool look after the lower level transformations.

The simple examples above illustrate how ConQuer achieves semantic clarity, relevance and stability. Let's look briefly at its semantic strength (expressibility). Further details on this may be found in [2]. The language supports the usual comparators (=, <, in, like etc.), logical operators (and, or, not), and bag functions (count, sum etc.), as well as a maybe operator for conceptual left outer joins. Subtype/supertype connections appear as "is" predicates.

Suppose we want to list the branch number of any USbranch that did not achieve the top rating before the year 1998, as well as the name, and cars (if any) of the branch's head. This may be formulated in ConQuer thus:

Q3 ✓USbranch
 +- **not** achieved Rank = 1 in Year < 1998
 +- **is** Branch
 +- is headed by Employee
 +- has ✓EmployeeName
 +- **maybe** drives ✓Car

Notice how easy this is, especially if the tool provides the predicates for each object type. As a minor point, ActiveQuery currently uses the syntactical variant “possibly” for “maybe”. You are invited to provide the lengthy SQL for this query. Although straightforward, notice how you need to locate the relevant four tables and then decide on what columns to join, what kinds of join to perform (inner or outer) and then add the intra-table restrictions. In other words, to do this in SQL you need to worry about low level implementation details.

The most powerful feature of ConQuer is its ability to perform *correlations* of arbitrary complexity. As a simple correlation example, consider the query: Who supervises an employee who lives in the same city as the supervisor but was born in a different country from the supervisor? Here is one way of expressing the query in ConQuer:

Q4 ✓Employee₁
 +- lives in City₁
 +- was born in Country₁
 +- supervises Employee₂
 +- lives in City₁
 +- was born in Country₂ <> Country₁

When an object type appears more than once, ActiveQuery automatically appends subscripts to distinguish the occurrences. You can assert that the instances are equal by equating the subscripts (e.g. City₁). More generally, you can use comparators to compare instances (e.g. Country₂ <> Country₁). Try this in SQL. It’s not that hard, but you have to admit it’s easier in ConQuer!

As a harder example that includes a function as well as nasty correlation, consider the query: Who owns a car, and does not drive more than one of those cars (that he/she owns)? In English, correlation is often achieved through pronouns. Here there is a correlation on cars (“those”) as well as employees (“he/she”). This query may be formulated as Q5. Recall that object variables with identical subscripts are correlated. This is used here to correlate cars (Car₁). The for-clause has only one instance of Employee in the query body to reference, so no subscripts are needed to perform the correlation for employees.

Q5 ✓Employee
 +- owns Car₁
 +- **not** drives Car₁
 +- **count**(Car₁) **for** Employee > 1

Equivalent SQL is shown below. Because the correlation stems from a function argument inside a negated function subquery, the correlation concerns membership in a set, not just equality with an outer instance (see italicized code). This is quite tricky, and even experienced SQL users might get it wrong.

```
S5 select X1.empnr
from Owns X1
where X1.empnr not in (
  select X2.empnr
  from Drives X2
  where X2.car in (select X3.car from Owns X3
    where X3.empnr = X1.empnr)
group by X2.empnr
having count(X2.car) > 1)
```

Last year I taught advanced SQL to a group of 4th year university students who already had years of experience with SQL. I then gave them a simple introduction to ConQuer, followed by a list of varied questions in English that they had to translate into both ConQuer and SQL. Even without tool support, they had little trouble with the ConQuer formulations, but they experienced great difficulty with the SQL. I admit the SQL questions were pretty nasty (lots of correlated subqueries and functions), but I set a wide range of questions without trying to bias them in favor of either language. At any rate, the relative performance was so dramatic that it reinforced my impression that ConQuer is much easier to learn than SQL. Of course, more extensive trials are needed for a reliable empirical evaluation of the language.

Business Rules

Although ORM's graphical notation can capture more constraints than popular ER notations such as IDEF1X and UML class diagrams, it still needs to be supplemented by a textual language to provide a complete coverage of the kinds of constraints and derivation rules found in business applications. Research is currently under way to adapt ConQuer for this purpose. This is analogous to the way that SQL is used for formulation of queries as well as declaration of constraints (e.g. check clauses) and derivation rules (e.g. view declarations). All of ORM's graphical constraints can be verbalized in FORML, an ORM language that was designed specifically for this purpose. The ConQuer language is more general, can be used to define other business rules, and can be mapped automatically to SQL. Hence it could be used as a very high level language for capturing business rules in general.

As a trivial example of a derivation rule, Figure 1 includes a definition of the derived fact type: Branch is in Country. Now that you have some familiarity with ConQuer, you

will recognize this definition as a ConQuer query. ActiveQuery allows you to define derived predicates, and store these definitions. These derived predicates (or “macros”) can then be used just like base predicates in other queries. Figure 1 also includes a subtype definition for USbranch. A subtype may be thought of as a derived object type, with its definition provided by a ConQuer query. Note that the subtype definition for USbranch made use of a derived predicate.

A constraint may be viewed as a check that a query searching for a violation of the constraint returns the null set. Hence constraints may also be expressed in terms of queries. Various high level constructs can be provided in the language to make it more natural than the not-exists-check-query form provided in SQL. Although there is no room here to go into detail, it should be clear that this approach is quite powerful.

Conclusion and Acknowledgement

This article outlined the benefits of lifting queries to the conceptual level, and argued that a truly conceptual query language should provide semantic strength, clarity, relevance and stability. It then indicated that ORM-based languages are especially suited for meeting these criteria, and gave examples of how queries can be formulated in one such language, ConQuer, and then mapped to SQL. Finally, it was noted that a conceptual query language can be used not just for queries but also for the declaration of business rules.

The ActiveQuery tool was constructed by a team of talented developers, now working at Visio Corporation. The fundamental research on the design of ConQuer and the associated mapping to SQL was performed jointly by Dr Anthony Bloesch and myself, and parts of this article are based on two of our papers [1, 2], in which a more formal discussion of the language’s semantics is provided.

References

1. Bloesch, A.C. & Halpin, T.A. 1996, ConQuer: a conceptual query language,’ *Proc. ER’96: 15th Int. Conf. on conceptual modeling*, Springer LNCS, no. 1157, pp. 121-33.
2. Bloesch, A.C. & Halpin, T.A. 1997, Conceptual queries using ConQuer-II,’ *Proc. ER’97: 16th Int. Conf. on conceptual modeling*, Springer LNCS 1131, pp. 113-26.
3. Cattell, R.G.G. & Barry, D. K. (eds) 1997, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, San Francisco CA (see ch. 4 for a definition of OQL).
4. Embley, D.W., Wu, H.A., Pinkston, J.S. & Czejdo, B. 1996, OSM-QL: a calculus-based graphical query language,’ Tech. Report, Dept of Comp. Science, Brigham Young Univ., Utah.
5. Halpin, T.A. 1995, *Conceptual Schema and Relational Database Design*, 2nd edn, Prentice Hall Australia, Sydney.
6. Halpin, T.A. 1995, Subtyping: conceptual and logical issues,’ *Data Base Newsletter*, ed. R.G. Ross, Database Research Group Inc., vol. 23, no. 6, pp. 3-9.

7. Halpin, T.A. 1996, Business rules and Object Role modeling,' *Database Programming and Design*, vol. 9, no. 10 (Oct. 1996), pp. 66-72.
8. Hofstede, A.H.M. ter, Proper, H.A. & Weide, th.P. van der 1996, Query formulation as an information retrieval problem,' *The Computer Journal*, vol. 39, no. 4, pp. 255-74.
9. Meersman, R. 1982, The RIDL conceptual language,' Research report, Int. Centre for Information Analysis Services, Control Data Belgium, Brussels.

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

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.

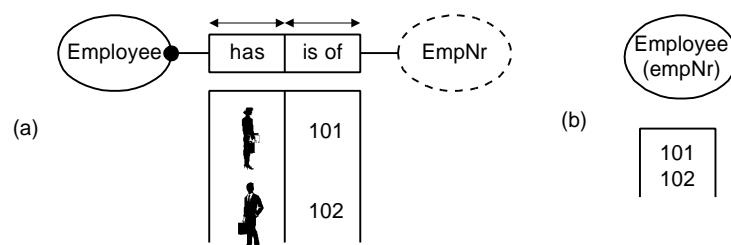


Figure 1: A simple reference scheme in ORM, shown (a) explicitly, (b) implicitly

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.

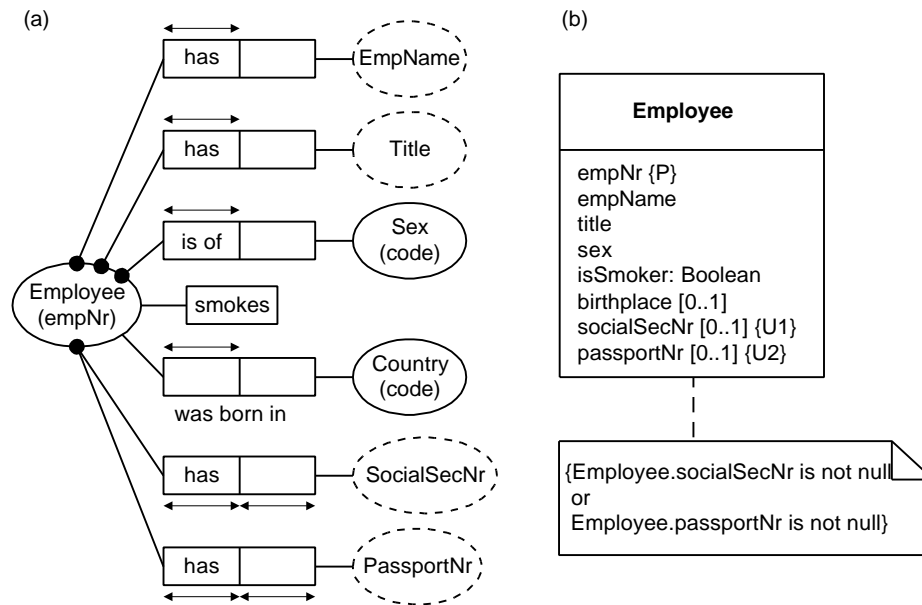


Figure 2: ORM relationship types (a) depicted as attributes in UML (b)

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 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** Person₁ has **a** Title **that** is restricted to Sex₁ **then** Person₁ is of Sex₁). 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.

UML Data Models From An ORM Perspective: Part 2

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

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

This paper is the second in a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 provided some historical background on both approaches, identified several design criteria for modeling languages, and discussed how object reference and single-valued attributes are modeled in both. In Part 2 we compare UML multi-valued attributes with ORM relationship types, including basic constraints on both. As part of this discussion, we also consider how these structures may be instantiated, using UML object diagrams or ORM fact tables.

Multi-valued attributes

Suppose that we are interested in recording 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). The mandatory role dot indicates that each employee has a name. The absence of 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 lack of a mandatory role dot on the roles of EmpName and Sport does not imply that these roles are optional. If in the global schema an object type has only one fact role, this is implied to be mandatory unless the object type has been declared independent. So if EmpName and Sport have no other roles in the complete application, their roles shown here are implicitly mandatory. This is of little importance, since implied constraints are automatically enforced with no additional overhead.

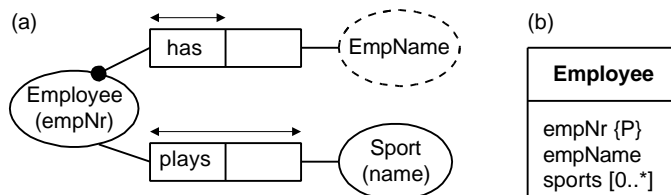


Figure 1: Plays depicted as an ORM $m:n$ fact type (a) and a UML multi-valued attribute (b)

Since an employee may play many sports, and a sport may be played by many employees, Plays is a *many-to-many* ($m:n$) relationship type. This is shown in ORM by

making the uniqueness constraint span both roles. Visually, this indicates that for each population of the fact type, only the combination of values for the two roles needs to be unique. In other words, each employee-sport pair can occur on at most one row of the associated fact table. Since it is understood that the population of any fact type is a *set* of rows (not a bag of rows), such a spanning uniqueness constraint always applies. We only show this constraint if no stronger one exists. For example, the uniqueness constraint on the empname fact type is stronger, since it spans just one role; so we don't bother adding the weaker, 2-role uniqueness constraint. Read from left to right, the empname relationship type is *many-to-one (n:1)*, since employees have at most one name, but the same name may refer to many employees.

One way of modeling the same situation in UML is shown in Figure 1(b). Here 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 queries. By restricting its base structures to elementary fact types, 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. If "*" is used without a lower bound, this is taken as an abbreviation for "0..*".

As mentioned in Part 1, an attribute with no explicit multiplicity constraint is assumed to be mandatory and single-valued (exactly one). This can be depicted explicitly by appending "[1]" to the relevant attribute. For example, to indicate explicitly that each employee has exactly one name, we would use "empName [1]". Although the UML standard [3] specifies that multi-valued attributes are allowed and that "[1]" is the default multiplicity of attributes, some authors of popular UML books appear to be unaware of this (e.g. [2], p. 63). Moreover, some of the principal authors of OMT (the Object Modeling Technique from which UML class diagrams are largely derivative) argue that the default is single-valued with nullability unspecified (i.e. either [1] or [0..1]); for example, see [1], p. 44.

ORM constraints are easily clarified by populating the fact types with sample populations. For example, see Figure 2. The inclusion of all the employees in the EmpName fact table, and the absence of employee 101 in the Plays fact table clearly shows that playing sport is optional. Notice also how the uniqueness constraints mark out which column or column-combination values can occur on at most one row. In the EmpName fact table, the first column values are unique, but the second column includes duplicates. In the Plays table, each column contains duplicates: only the whole rows are unique. Such populations are very useful for checking constraints with the subject matter experts. This validation-via-example feature of ORM holds for all its constraints, not just mandatory roles and uniqueness, since all its constraints are role-based, and each role corresponds to a fact table column.

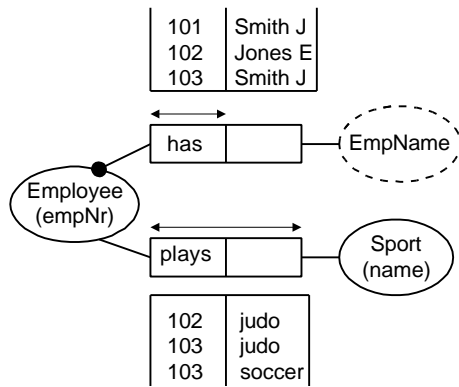


Figure 2: Fact tables with sample populations clarify the constraints

Instead of using fact tables for the purposes of instantiation, UML provides *object diagrams*. These are essentially class diagrams in which each object is shown as a separate class instance, with data values supplied for its attributes. As a simple example, the population of Figure 2 may be displayed in a UML object diagram as shown in Figure 3. For simple cases like this, object diagrams are useful. However, as we see later, they rapidly become extremely unwieldy if we wish to display multiple instances for more complex cases. In contrast, fact tables scale easily to handle large and complex cases.

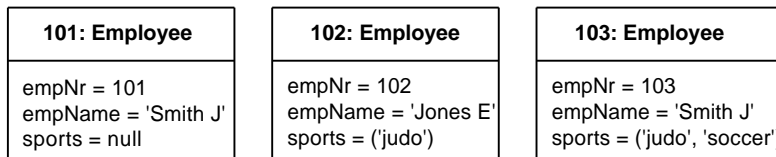


Figure 3: Object diagrams may be used in UML to show sample populations

Let's look at a couple more examples involving multi-valued attributes. At my former university, employees could apply for a permit to park on campus. The parking permit form required one to enter the license plate numbers of those cars (up to three) that one might want to park. A portable sticker was issued that could be transferred from one car to another. Over time, an employee may be issued different permits, and we want to maintain an historical record of this. Suppose that it is also a rule that an employee can be issued at most one parking permit on the same day. One way of modeling this situation in ORM is set out in Figure 4.

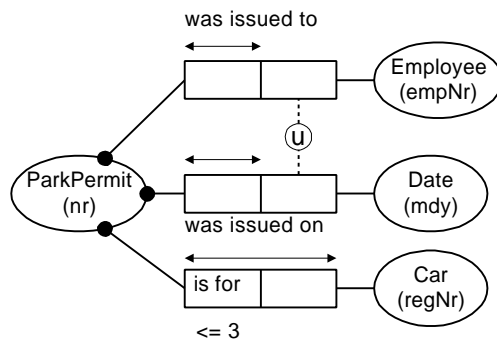


Figure 4: ORM diagram with external uniqueness constraint and frequency constraint

Here the circled “u” depicts an *external uniqueness constraint* (i.e. a uniqueness constraint that spans roles from different predicates). This captures the rule that any given employee on any given date may be issued at most one parking permit. An external uniqueness constraint is equivalent to an internal uniqueness constraint over the same roles in the conceptual join of the predicates. In this case, a join would create the compound fact type: ParkPermit was issued to Employee on Date. If this derived fact type were populated, the Employee-Date combination would be unique, and this is what the constraint means.

Notice also the “<=3” next to the first role of the fact type ParkPermit-is-for-Car. This is a *frequency constraint*, indicating that each permit in the fact column for that role appears there at most three times. In other words, each parking permit allows at most three cars to be parked on campus. In ORM, both uniqueness and frequency constraints may be applied to one or more roles, possibly from different predicates. Frequency constraints place restrictions on the number of times that instances of the role(s) may appear. The frequency might be a single number (e.g. 2), a number range (e.g. 2..5), a list of numbers (e.g. 2, 4) or a combination. The expression “<= n” means “at most n”, but since it applies to entries in the role column (rather than the object type) this is equivalent to “1.. n”, since any entry for the role has already appeared once. The expression “>= n” means “at least n”. A frequency constraint of 1 is simply a uniqueness constraint. However because uniqueness constraints are so common, they are given a special notation of their own.

One way of modeling the same application in UML is shown in Figure 5. In addition to the ParkPermit class, Employee and Car classes are included. The “...” shown here simply indicates that other attributes of these classes exist in the global schema (this use of “...” is not part of the UML notation). For discussion purposes, the attribute domains are displayed. In UML these domains are called “type expressions”. Instead of defining a standard syntax for type expressions, UML allows them to be written in any implementation language, assuming the latter provides the relevant parser. For example, one type expression might be a C++ function pointer. To keep our analysis model at least semi-conceptual, we restrict type expressions to simple data types and classes. Data types are sets of pure values (no oids), and include primitive types (e.g. Integer, String) as well as enumeration types (including Boolean and user-defined). In our example, the attribute domains include the data types Integer and Date, as well as the classes Employee and Car.

By using classes as domains in this way, we can at least understand when an attribute corresponds to a association between entities, even it is not displayed as such.

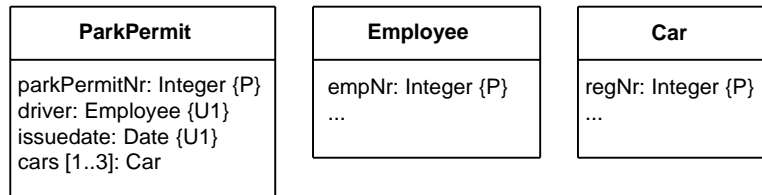


Figure 5: A UML alternative to the ORM model in Figure 4

As discussed in Part 1, constraints not included in the standard notation may be added in braces in some implementation language. In Figure 5 we use “{P}” for “primary identifier”, and “{U1}” on both driver and issuedate to indicate that this combination is unique. Taken together, the two “{U1}” annotations correspond to the ORM external uniqueness constraint in Figure 4. The “[1..3]” constraint on the cars attribute indicates that each parking permit is associated with at least one and at most three cars. The “at least one” part of this corresponds to an ORM mandatory role constraint; and the “at most three” corresponds to the “<= 3” ORM frequency constraint. Recall that mandatory role constraints are separated out in ORM, mainly because they have *global* impact (each population instance of that type must play all the roles of that object type in the global schema). In contrast, other ORM constraints (e.g. frequency and uniqueness) are local, applying only to the population of the associated role(s).

As a final example of multi-valued attributes, suppose that we wish to record the nicknames and colors of country flags. Let us agree to record at most two nicknames for any given flag, and that nicknames apply to only one flag. For example, “Old Glory” and perhaps “The Star-spangled Banner” might be used as nicknames for the USA flag. Flags have at least one color. Figure 6(a) shows one way to model this in ORM. For verbalization purposes we identify each flag by its country. Since country is an entity type, the reference scheme is shown explicitly (parenthesized reference modes may abbreviate reference schemes only when the referencing type is a value type). The uniqueness constraint on the role played by Country could be annotated with a “P” for primary reference, but this is implied if Flag has no other reference schemes. The “<= 2” frequency constraint indicates that each flag has at most two nicknames, and the uniqueness constraint on the role of NickName indicates that each nickname refers to at most one flag.

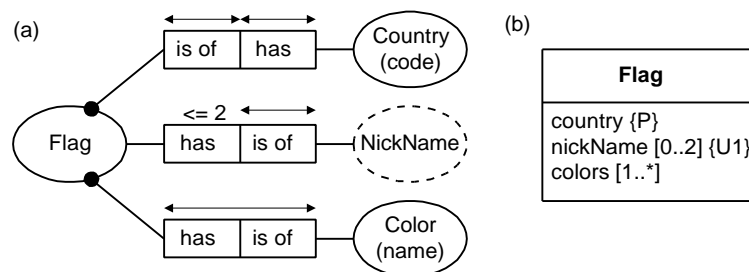


Figure 6

Figure 6(b) shows one way of modeling this in UML. The “[0..2]” indicates that each flag has at most two (from zero to two) nicknames, and we use “{U1}” to indicate that nicknames refer to at most one flag. The “[1..*]” declares that a flag has one or more colors. In this case we have omitted the display of attribute domains. NickName would typically have a data type for its domain (e.g. String). If we don't want to store any information about countries or colors, we might choose String as the domain for country and colors as well (although this is sub-conceptual, because in the real world countries and colors are not character strings). However since we might want to add information about these later, it's better to use classes for their domains (e.g. Country and Color). If we do this, we need to define the classes as well (cf. our previous example).

As we discuss in the next issue, UML gives us the choice of modeling a feature as an attribute or an association (similar to an ORM relationship type). At least for conceptual analysis and querying, explicit associations usually have many advantages over attributes, especially multi-valued attributes. 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 (as we did with our braced comments). This applies not only to simple uniqueness constraints (as discussed earlier) but also to other kinds of constraints (frequency, subset, exclusion etc.) over one or more roles that include the role played by the attribute's domain (in the implicit association corresponding to the attribute). For example, if the association Flag-is-of-Country is explicitly depicted in UML, the constraint that each country has at most one flag can be captured by adding a multiplicity constraint of “0..1” on the left role of this association. Although country and color are naturally conceived as classes, nickname would normally be construed as a data type (e.g. a subtype of String). Although associations in UML may include data types (not just classes), this is somewhat awkward; so in UML, nickname might best be left as a multi-valued attribute. Of course, we could model it cleanly in ORM first.

Another reason for favoring associations over attributes is stability. As we discuss later, if ever we want to talk about a relationship, it is possible in both ORM and UML to make an object out of it, and simply attach the new details to it. If instead we modeled the feature as an attribute, we would not be able to add the new details without first changing our original schema: in effect we would need to first replace the attribute by an association. 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. 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. The notion of objectified relationship types or association classes will be covered in a later issue.

Another problem with multi-valued attributes is that queries on them need some way of extracting the components, and hence complicate the query process for users. As a trivial example, compare queries Q1, Q2 expressed in ConQuer (the ORM query language

supported by Visio's ActiveQuery tool) with their counterparts in OQL (the Object Query language proposed by ODMG):

(Q1) **List each** Color **that** is of Flag USA'

(Q2) **List each** Flag **that** has Color red.'

(Q1a) **select** x.colors **from** x **in** Flag **where** x.country = 'USA'

(Q2a) **select** x.country **from** x **in** Flag **where** 'fed' **in** x.colors

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. For example, the ORM schema in Figure 6 can be prepared for mapping by annotating the roles played by NickName and Color to map as sets inside the mapped Flag structure. Such annotations are not a conceptual issue, and can be postponed till mapping. If you ever feel tempted to use multi-valued attributes in UML, you may be thinking about how you want the structure to be implemented rather than first trying to understand how things are related in the real world.

Later issues

The next issue focuses on a detailed comparison between ORM relationship types and UML associations, including related constraints. We then contrast ORM nesting with UML association classes, and ORM co-referencing with UML qualified associations. Later issues 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. Fowler, M. with Scott, K. 1997, *UML Distilled*, Addison-Wesley.
3. OMG UML Revision Task Force website, <http://uml.systemhouse.mci.com/>.

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

UML Data Models From An ORM Perspective: Part 3

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

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

This paper is the third in a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 provided some historical background on both approaches, identified design criteria for modeling languages, and discussed how object reference and single-valued attributes are modeled in both. Part 2 compared UML multi-valued attributes with ORM relationship types, and discussed basic constraints on both, as well as instantiation using UML object diagrams or ORM fact tables. This third issue compares UML associations and related multiplicity constraints with ORM relationship types and related uniqueness, mandatory role and frequency constraints. It also contrasts instantiation of associations using UML object diagrams and ORM fact tables.

Associations

Before discussing UML associations in detail, we review some ideas discussed previously. Attributes in UML are depicted as relationship types in ORM. 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). In both UML and ORM, a *role* is a part played in a relationship. The number of roles in a relationship is its *arity*.

ORM allows relationships of any arity. 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 natural verbalization of constraints and navigation paths in any direction. A predicate is an elementary sentence with holes in it for object terms. In ORM 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 sentence types and constraints may be specified either textually or graphically. Visio's InfoModeler can automatically transform between the graphical and textual representations. In an ORM diagram, roles appear as *boxes*, connected by a line to their object type. InfoModeler allows role names to be added on a properties sheet rather than on the diagram; in principle however, an ORM tool could display role names directly on the diagram (preferably with the display toggled by the user to avoid clutter). 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. Association names are normally shown in italics, starting with a capital letter. Binary associations are depicted as *lines* between classes. Association lines may include elbows to assist with layout or when needed (e.g. for ring relationships). Association roles appear simply as line ends instead of boxes, but may optionally be given role names. Once added, role names may not be suppressed. 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.

Figure 1 depicts two binary associations in both UML and ORM. On the UML diagram we have chosen to display the association names, their directional markers and all the role names: all of these could have been omitted. To avoid ambiguity however, either the directed association name or the role names should be shown. In the ORM diagram, both forward and inverse predicate names have been shown: at most one of these may be omitted. Role names are not displayed on the ORM diagram but may be added (as discussed above).

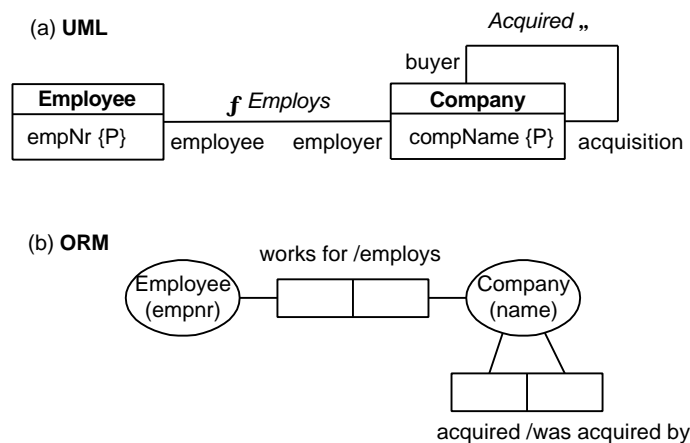


Figure 1: Binary associations in (a) UML and (b) ORM

Ternary and higher arity associations in UML depicted as a *diamond* connected by lines to the classes. Because many lines are used to denote the association, directional verbalization is ruled out, so the diagram can't be used to communicate in terms of sentences. This non-linear layout also often 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 of little use for population checks.

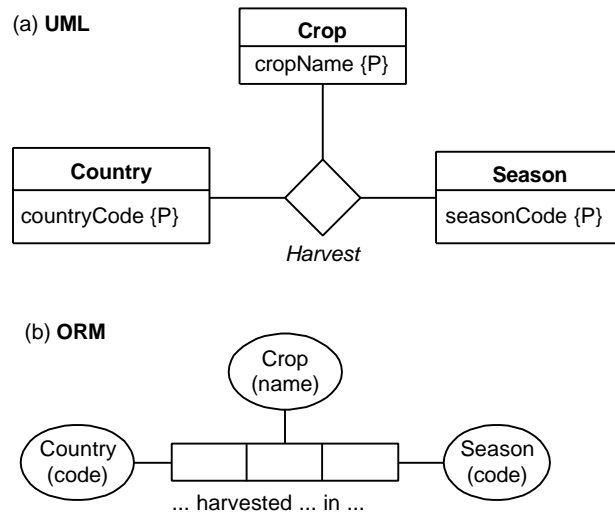


Figure 2: A ternary association in (a) UML and (b) ORM

As discussed in the previous issue, 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. [0], p. 31). Hence, as noted in the UML Notation Guide, “the use of object diagrams is fairly limited”.

Multiplicity constraints on associations

The previous issues discussed how UML depicts multiplicity constraints on attributes. A similar notation is used for associations, where the relevant multiplicities are written beside the relevant roles. Figure 3(a) adds the relevant multiplicity constraints to Figure 1(a). A “*” abbreviates “0..*”, meaning “zero or more”, “1” abbreviates “1..1”, meaning “exactly one”, and “0..1” means “at most one”. Unlike some ER notations, UML places each multiplicity constraint on the “far role”, in the direction in which the association is read. Hence the constraints in this example mean: each company employs zero or more employees; each employee is employed by exactly one company; each company acquired zero or more companies; and each company was acquired by at most one company.

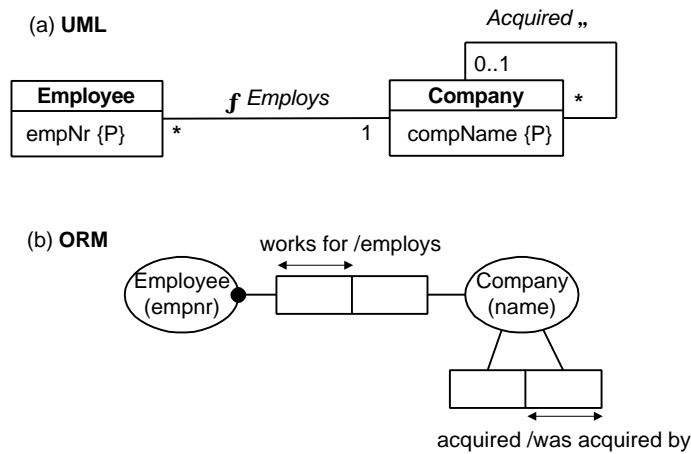


Figure 3: UML multiplicity constraints captured in ORM by uniqueness and mandatory role constraints.

The corresponding ORM constraints are depicted in Figure 3(b). Recall that multiplicity covers both cardinality (frequency) and optionality. Here the mandatory role constraint indicates that each employee works for at least one company, and the uniqueness constraints indicate that each employee works for at most one company, and each company was acquired by at most one company. As usual, the ORM notation facilitates verbalization and population. InfoModeler allows these constraints to be entered graphically, or by answering multiplicity questions, or by induction from sample populations, and can automatically verbalize the constraints.

For binary associations, there are four possible uniqueness constraint patterns ($n:1$, $1:n$; $1:1$, $m:n$) and four possible mandatory role patterns (only the left role mandatory, only the right role mandatory, both roles mandatory, both roles optional). Hence if we restrict ourselves to a maximum frequency of one, there are 16 possible multiplicity combinations for binary associations. The first four of these are shown in Figure 4, covering the cases where both roles are optional.

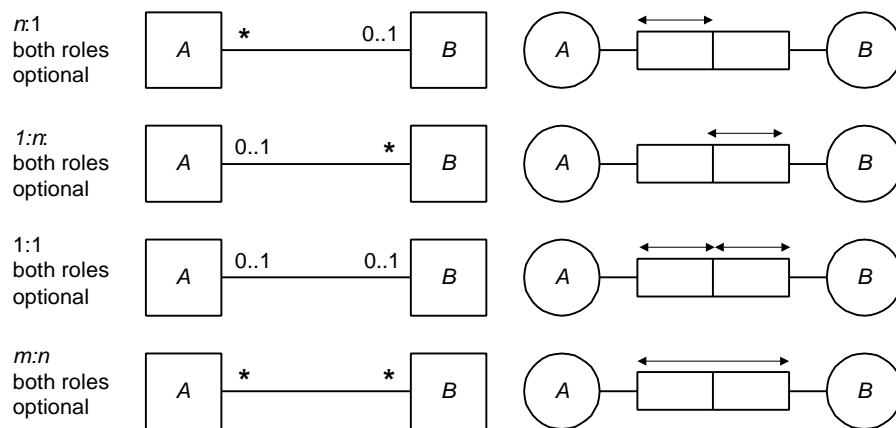


Figure 4: Equivalent constraint patterns in UML and ORM, where both roles are optional

The next four cases, shown in Figure 5, cover the situation where the first role is mandatory and the second role is optional.

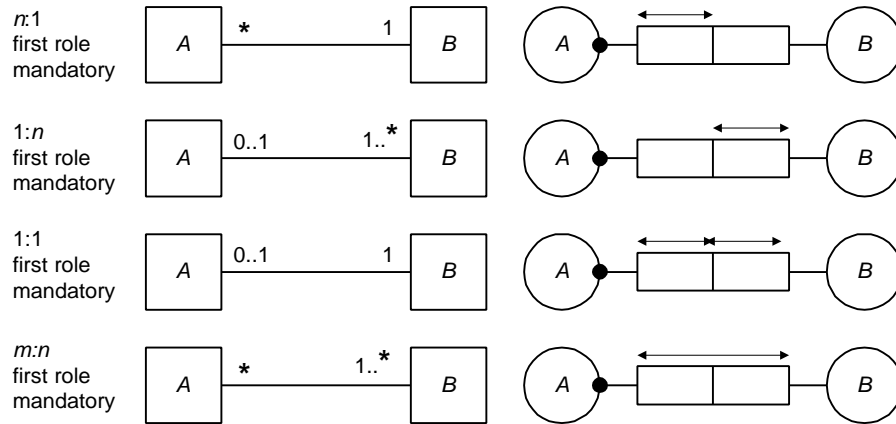


Figure 5: Equivalent constraint patterns in UML and ORM, where first role is mandatory

The next four cases, shown in Figure 6, cover the situation where the second role is mandatory and the first role is optional. Finally, Figure 7 covers the four cases where both roles are mandatory.

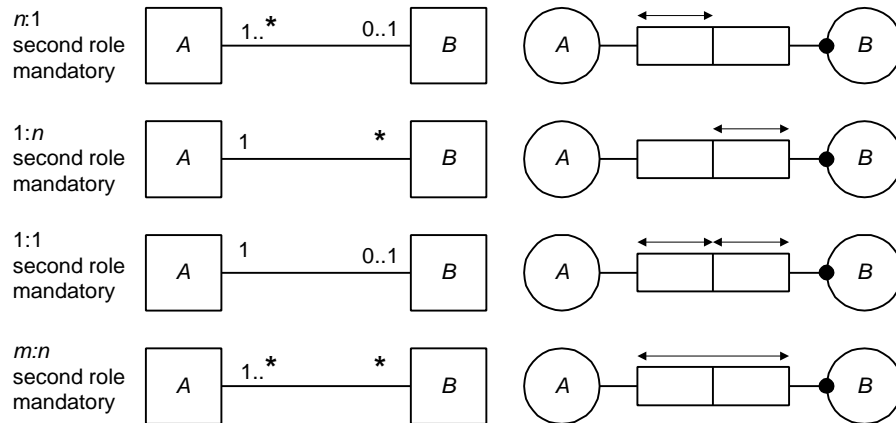


Figure 6: Equivalent constraint patterns in UML and ORM, where second role is mandatory

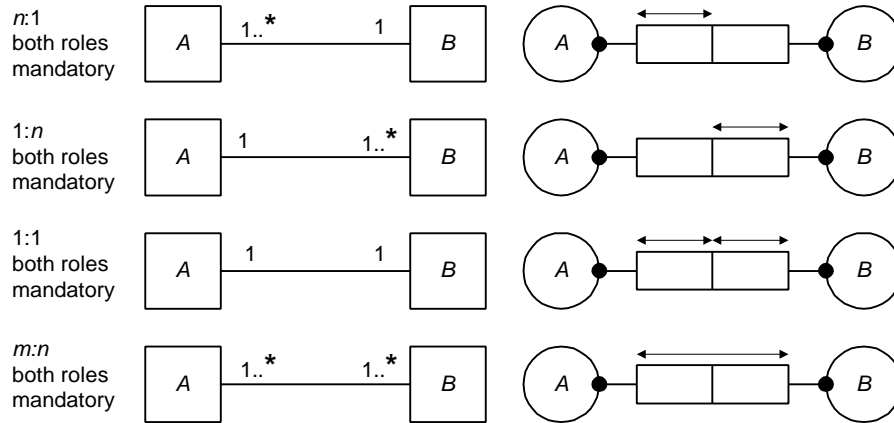


Figure 7: Equivalent constraint patterns in UML and ORM, where both roles are mandatory

In the previous issue, we discussed attribute multiplicity constraints involving occurrence frequency lists and/or ranges containing frequencies other than zero or one (e.g. “1..7, 10”). For such cases, ORM uses general frequency constraints instead of uniqueness constraints. In a similar way, both UML and ORM cater for multiplicity constraints of arbitrary complexity on single roles. As discussed in a later issue, ORM is actually more expressive in this regard since it can apply such constraints to arbitrary collections of roles.

An *internal constraint* applies to roles in a single association. For an elementary 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. In UML, a multiplicity constraint on a role of an n -ary association effectively constrains the population of the other roles combined. For example, Figure 8 is a UML diagram for a ternary association in which both Room-Time and Time-Activity pairs are unique. For simplicity, we have omitted the conceptual reference schemes for the classes.

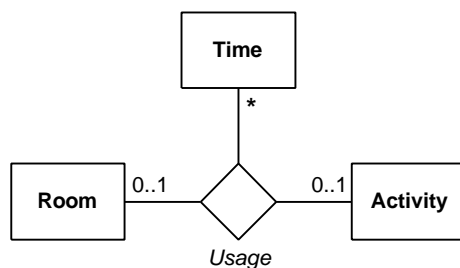


Figure 8: Multiplicity constraints on a ternary in UML

An ORM depiction of the same association is shown in Figure 9, including the reference schemes and sample population. The left-hand uniqueness constraint indicates that Room-Time is unique (i.e. for any given room and time, there is at most one activity). The right-hand uniqueness constraint indicates that Time-Activity is unique (i.e. for any

given time and activity, at most one room is used). 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 and asking the client whether this is possible.

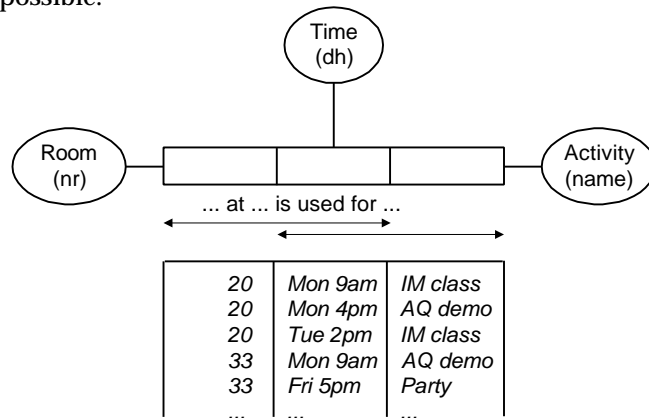


Figure 9: An ORM ternary with a sample population

Later issues

The next issue looks at associations in more detail, covering some advanced constraints, and then contrasts ORM nesting with UML association classes, and ORM co-referencing with UML qualified associations. Later issues discuss more advanced constraints, aggregation, subtyping, derivation rules and queries.

References

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

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

UML data models from an ORM perspective: Part 4

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

This article first appeared in the August 1998 issue of the Journal of Conceptual Modeling, published by InConcept.

This paper is the fourth in a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 provided historical background and design criteria for modeling languages, and discussed object reference and single-valued attributes. Part 2 discussed multi-valued attributes, basic constraints, and instantiation using UML object diagrams or ORM fact tables. Part 3 compared UML associations and related multiplicity constraints with ORM relationship types and related uniqueness, mandatory role and frequency constraints; it also contrasted instantiation of associations using UML object diagrams and ORM fact tables. In Part 4 we look at associations in more detail, contrasting ORM nesting with UML association classes, and ORM co-referencing with UML qualified associations, then discuss exclusion constraints, and summarize how the two methods compare with respect to terms and notations for data structures and instances.

Association classes

Unlike many ER versions, both UML and ORM allow associations to be objectified as first class object types, called *association classes* in UML and *nested object types* (or *objectified relationship types*) in ORM. UML requires the same name to be used for the original association and the association class, impeding natural verbalization of at least one of these constructs. In contrast, ORM nesting is based on linguistic *nominalization* (a verb phrase is objectified by a noun phrase), thus allowing both to be verbalized naturally, with different names for each. When an association is objectified, VisioModeler automatically creates a name for the nested object type, which you are free to edit. UML allows the association class name to be displayed on the association or the association class, or both.

In spite of identifying association classes with their underlying association, UML displays them separately, making the connection by a dashed line (see Figure 1). Each person may write many papers, and each paper is written by at least one person. In the UML depiction, we have used “{P}” to indicate the primary reference attributes used for

human communication about persons and papers. Since authorship is *m:n*, the association class Writing has a primary reference scheme based on the combination of person and paper (e.g. the writing by person 'Norma Jones' of paper 33). The optional period attribute stores how long that person took to write that paper. Instead of distancing the objectified association from its underlying association, ORM intuitively envelops the association with an object type frame. Writing is marked independent (displayed with "!") to indicate that a writing object may exist, independently of whether we record its period. ORM displays Period as an object type, not an attribute, and includes its unit.

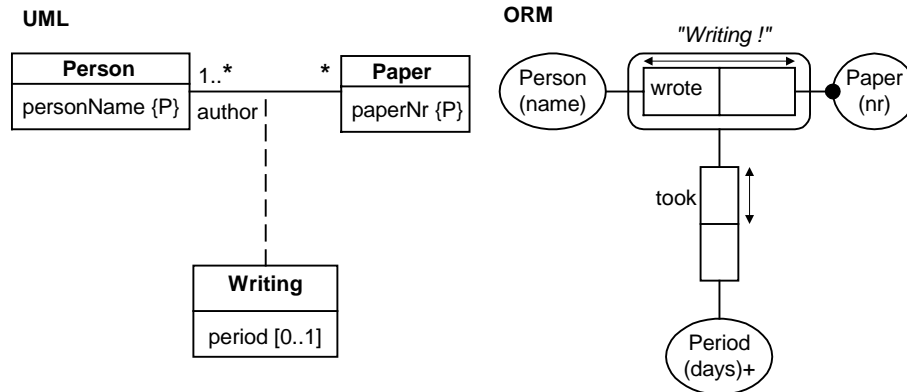


Figure 1: Writing is depicted as an objectified association in UML and ORM

Objectified relationships in standard ORM must have at least two roles, and must either have a single, spanning uniqueness constraint or be a 1:1 binary. A Dutch variant of ORM known as FCO-IM allows unaries to be objectified, but this adds no extra expressibility and is not supported in Visio technology. UML allows any association (binary and above) to be objectified into a class, regardless of its multiplicity constraints. In particular UML allows objectification of *n:1* associations, unlike ORM (see Figure 2).

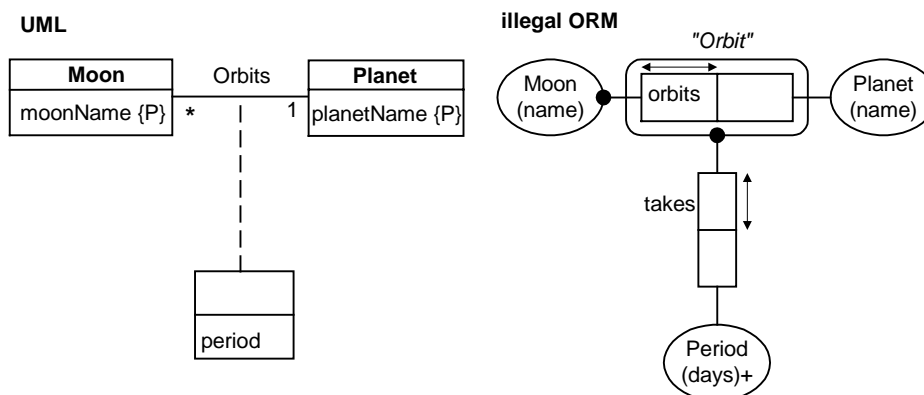


Figure 2: Objectification of *n:1* associations is allowed in UML but not ORM

ORM currently forbids such cases, mainly to encourage modelers to conceptualize facts in elementary rather than compound form. For example, since each moon orbits only one planet, we can specify its orbital period without having to mention its planet. Hence

ORM requires this case to be modeled using two separate fact types, as shown in Figure 3. This also facilitates removal/addition of mandatory role constraints on the fact types independently (e.g. the nested version has to be completely remodeled if we now decide to keep period facts mandatory but make planet facts optional). However, if an experienced modeler aware of the implications still finds it easier to think about a situation as a nested $n:1$ association, there may be some argument for relaxing ORM's restriction, just as we relaxed it for 1:1 cases to avoid arbitrary decisions about relative importance. If enough people feel this way, ORM could be relaxed to downgrade this error to a warning, and mapping algorithms would add a pre-processing step to re-attach roles and adjust constraints internally.

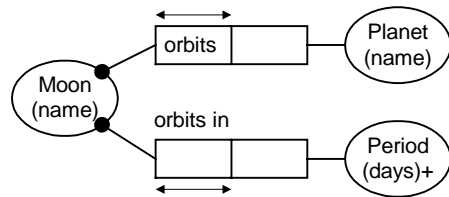


Figure 3: ORM models $n:1$ association classes instead as separate, elementary fact types

Qualified associations

In Part 2 of this series, we saw that UML has no graphic notation to capture ORM external uniqueness constraints across roles that are remodeled as attributes in UML. Hence we introduced our own $\{Un\}$ notation to append as textual constraints to the constrained attributes (see Part 2, Figures 4 and 5). Simple cases where ORM uses an external uniqueness constraint for *co-referencing* can also be modeled in UML using *qualified associations*. Here, instead of depicting the relevant ORM roles or object types as attributes, UML uses a class, adjacent to a *qualifier*, through which connection is made to the relevant association role. A qualifier in UML is a set of one or more attributes, whose values can be used to partition the class, and is depicted as a rectangular box enclosing its attributes. Figure 4 is based on an example from the UML standard document [4], along with the ORM counterpart.

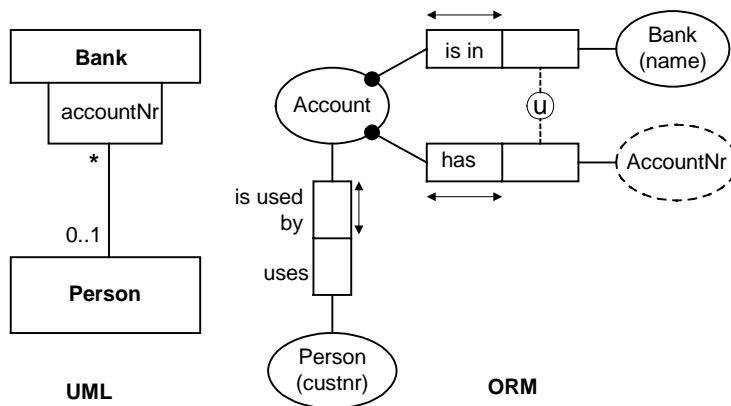


Figure 4: Qualified association in UML, and co-referenced object type in ORM

Here each bank account is used by at most one person, and each person may use many accounts. In the UML model, the attribute `accountNr` is used as a qualifier on the association, effectively partitioning each bank into different accounts. In the ORM model, an `Account` object type is explicitly introduced, and is referenced by combining its bank with its (local) account number. The circled “u” may be replaced by a “P” to indicate primary reference.

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. For a similar example, see [2] (p. 92, Fig. 5.10), where `product` is used with `Order` to qualify an order line association: again, this is unfortunate, since we would normally introduce a `Product` class to record data about products, and relevant connections are then lost. As a complicated example of this deficiency, see [1] (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. The use of qualified associations in UML is hard to motivate, but may be partly explained by their ability to capture some compound uniqueness constraints in the standard graphic notation, rather than relying on non-standard textual notations (such as our $\{Un\}$ notation).

ORM’s concept of an external uniqueness constraint that may be applied to a set of roles in one or more predicates provides a simple, uniform way to capture all of UML’s qualified associations and unique attribute combinations, as well as other cases not expressible in UML graphical notation (e.g. cases with $m:n$ predicates or long join paths). As always, the ORM notation has the further advantage of facilitating validation through verbalization and multiple instantiation.

Or-associations

UML uses the term *or-association* for one of many associations stemming from a class, where at any given time each class member may participate in at most one of these associations. To indicate this, UML uses what it calls an *or-constraint* between the associations, attaching the constraint string “{or}” to a dotted line connecting the relevant associations. Figure 5 is based on an example from the UML standard. For simplicity, reference schemes and other constraints are omitted.

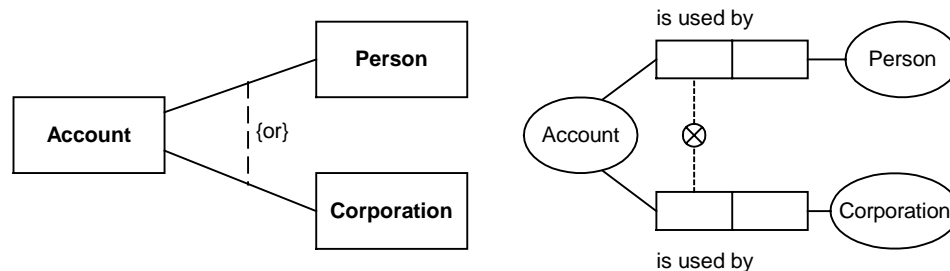


Figure 5: No account is used by both a person and a corporation

UML’s use of “or” for this constraint is confusing because it is used in an *exclusive* instead of inclusive sense (in contrast to virtually all computer languages). An alternative such as “xor” would be less ambiguous, and hence safer, even if artificial.¹ There is another possible confusion arising from the standard document itself. A literal reading of the latest version (1.2) of the UML standard indicates that the constraint simply means that an account is used by *at most one* of the two choices (person or corporation). However, some authors argue that its use in OMT (a precursor of UML) means each account must be used by *exactly one* of these choices ([1], p. 50). If this is the case, the constraint means that the disjunction is both exclusive and mandatory. Given that the lengthy UML standard currently contains a number of ambiguities and inconsistencies, I’m not sure which reading is actually correct. For now, I’ll assume that the weaker reading (exclusive) is correct. In this case, the constraint is captured in ORM by an *exclusion constraint*, shown by connecting “⊗” by dotted lines to the relevant roles (see above figure). If the stronger reading is correct², a disjunctive mandatory role constraint needs to be added as well (see Part 1).

UML or-constraints apply between single roles. The standard seems to imply that these roles must belong to different associations. If so, UML cannot use an or-constraint between roles of a ring fact type (e.g. between the husband and wife roles of a marriage association). ORM exclusion constraints cover this case, as well as many other cases not expressible in UML graphic notation. ORM exclusion constraints may apply to any set of compatible *role-sequences*, by connecting “⊗” by dotted lines to the relevant role-sequences. As a trivial example, consider the difference between the following two constraints: no person both wrote and reviewed a book; no person wrote and reviewed the same book. ORM clearly distinguishes these by noting the precise arguments of the constraint (see Figure 6).

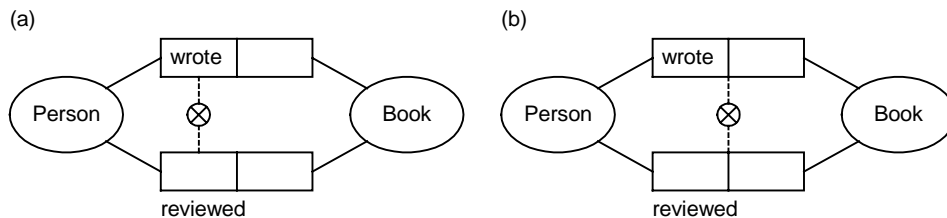


Figure 6: (a) no person wrote and reviewed; (b) no person wrote and reviewed *the same* book

The pair-exclusion constraint in Figure 6(b) can be expressed in UML by adding a comment box that includes a textual constraint written in some language (e.g. OCL), and connecting this by dotted lines to the two associations. However this notation is both cluttered and non-standard (since UML allows users to pick their own language to write textual constraints).

UML has no graphic notation for exclusion between attributes, or between attributes and associations. In Figure 7(a), the unary predicate must be modeled in UML as a

¹ After the original publication of this article, UML 1.3 replaced the “or” constraint notation by “xor”

² In UML 1.3, the xor constraint was clarified to mean the stronger reading, i.e. “exactly one”

Boolean attribute, and the contract predicate would probably be modeled as a `contractDate` attribute. In Figure 7(b), the completion predicate would be modeled in UML as a `completionDate` attribute of the `Project` class, while resource usage would normally be modeled as an association between `Project` and `Resource` classes. If we made these modeling choices in UML, we must resort to non-standard notations or textual constraints to add exclusion constraints between attributes (a) or between an attribute and association (b). There are alternative ways to model these cases in UML (e.g. using subtypes) that offer more chance to capture the constraints graphically, but it is clear that UML's or-constraint is far less expressive than ORM's exclusion constraint.

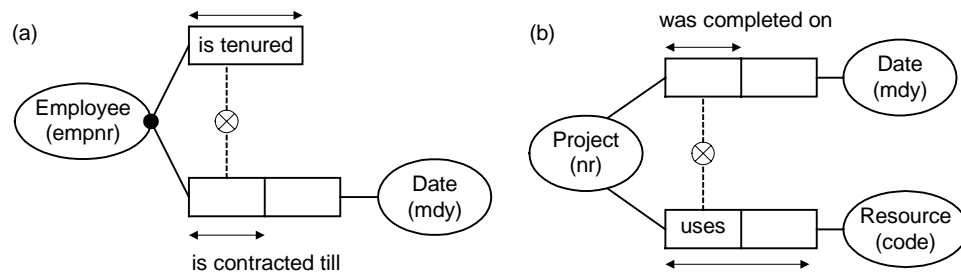


Figure 7: A comparative summary of data structure concepts

We've now covered essentially all the high level data structures that can be specified in graphic notation on ORM data models and UML class diagrams. As we discuss in a later issue, collection types may also be specified in both ORM and UML via textual annotations. Table 1 summarizes the differences between the two modeling methods with respect to terms and graphic (not textual) notations for data instances and structures. We still have several constraints to discuss, so will delay provision of a summary table about constraints till a later issue.

Table 1: Basic correspondence between ORM and UML conceptual data concepts

<i>Data instances/structures</i>	
<i>ORM</i>	<i>UML</i>
Entity	Object
Value	Data value
Object	Object or Data value
Entity type	Class
Value type	Data type
Object type	Class or Data type
— { use relationship type }	Attribute
Unary relationship type	— { use Boolean attribute }
2 ⁺ -ary relationship type	Association
2 ⁺ -ary relationship instance	Link
Nested object type	Association class
Co-reference	Qualified association §

§ = incomplete coverage of corresponding concept

Later issues

Later issues will discuss more advanced graphic constraints in both ORM and UML (subset, equality, aggregation, ring, join etc.), 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. Fowler, M. with Scott, K. 1997, UML Distilled, Addison-Wesley.
3. Halpin, T. 1995, Conceptual Schema and Relational Database Design, 2nd edn, Prentice Hall Australia.
4. OMG-UML v1.2, OMG UML Revision Task Force website, <http://uml.systemhouse.mci.com/>.

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

UML data models from an ORM perspective: Part 5

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

This paper first appeared in the October 1998 issue of the Journal of Conceptual Modeling, published by InConcept.

This paper is the fifth in a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 discussed historical background, design criteria for modeling languages, object reference and single-valued attributes. Part 2 covered multi-valued attributes, basic constraints, and instantiation using UML object diagrams or ORM fact tables. Part 3 compared UML associations and related multiplicity constraints with ORM relationship types and related uniqueness, mandatory role and frequency constraints, as well as how associations may be instantiated. Part 4 contrasted ORM nesting with UML association classes, ORM co-referencing with UML qualified associations, and ORM exclusion constraints with UML or-constraints. Part 5 discusses ORM subset and equality constraints, and how these may be specified in UML.

Subset constraints

ORM allows a *subset constraint* to be graphically specified between any pair of compatible role-sequences by connecting them with a dashed arrow. This declares that the population of the source role sequence must always be a subset of the target role sequence (the one hit by the arrow-head). Each sequence may comprise one or more roles. These constraints have corresponding verbalizations. For example, in Figure 1 the subset constraint between single roles indicates that students have second names only if they have first names. The other subset constraint is between Student-Course role-pairs, and declares that students may pass tests in a course only if they have enrolled in that course. Since the role of having a surname is mandatory for Student, subset constraints to it from all the other student roles are implied (and hence not shown).

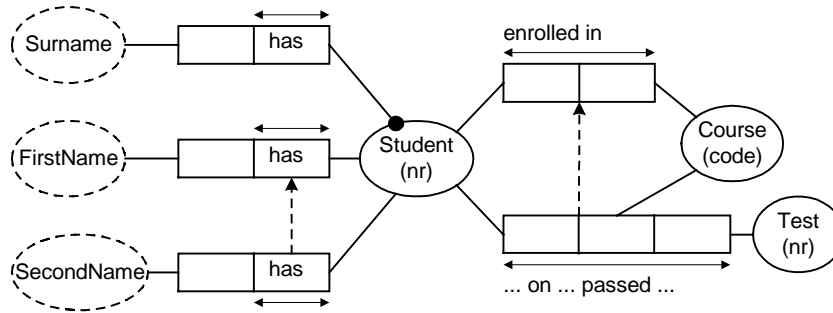


Figure 1: Subset constraints in ORM

As an extension mechanism, UML allows subset constraints to be specified between *whole associations* by attaching the constraint label “{subset}” next to a dashed arrow between the associations. For example, the subset constraint in Figure 2 indicates that any person who chairs a committee must be a member of that committee.

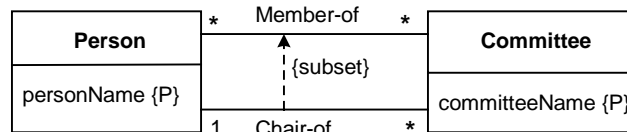


Figure 2: A subset constraint in UML

However UML does not provide a graphic notation for subset constraints between single roles or between parts of associations. Hence if a UML diagram depicted the relationship types in Figure 1 as associations, it would not be able to capture the subset constraints graphically. Of course, other options are available in UML. For instance, if we model surname, firstName and secondName as attributes of Person we can express the single-role subset constraint by attaching a comment including the following textual constraint (see Figure 3):

Student.firstName **is not null** or Student.secondName **is null**

Although this does capture the subset constraint, it is at a lower level than ORM’s graphic or verbalized form, and is basically the same as the check clause generated by VisioModeler when mapping the constraint down to a relational implementation.

One way in UML to capture the pair-subset constraint from Figure 1 is to transform the ternary into a binary association with a subset constraint to the enrollment association, and with a binary association to Test. A better solution is to use ORM’s overlap algorithm [2, p. 349] to objectify the enrollment association and associate this with Test. As discussed in Part 4, the equivalent UML action is to make a class out of enrollment (see Figure 3). Although in this situation an association class provides a good way to cater for a compound subset constraint, sometimes this nesting transformation leads to a very

unnatural view of the world. Ideally the modeler should be able to view the world naturally, while having optimization transformations that lessen the clarity of the conceptual schema performed under the covers.

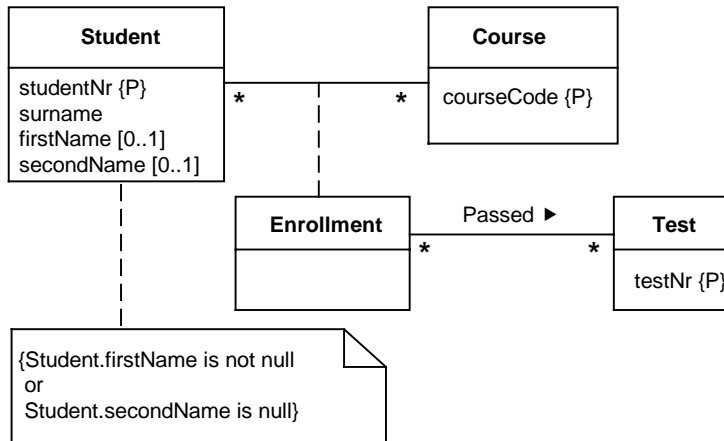


Figure 3: A UML version of the ORM schema in Figure 1

ORM has a mature formalization, including a rigorous theory of such topics as schema consistency, equivalence and implication. UML was only recently standardized, and is undergoing revisions. Since formal guidelines for working with UML are somewhat immature, extra care is needed to avoid logical problems. As a simple example, look back at Figure 2, which comes from the current draft of the UML 1.2 standard [3], with reference schemes added. Do you spot anything confusing about the constraints?

You probably noticed the problem. The multiplicity constraint of 1 on the chair association indicates that each committee must have at least one chair. The subset constraint tells us that a chair of a committee must also be a member of that committee. Taken together, these constraints imply that each committee must have a member. Hence one would expect to see a multiplicity constraint of “1..*” (one or more) on the Person end of the membership association. However we see a constraint of “*” (zero or more) instead, which at best is very misleading.

An ORM schema equivalent to Figure 2 is shown in Figure 4(a). The implied mandatory role constraint (each Committee includes at least one Person) is added explicitly in Figure 4(b). Which representation do you prefer?

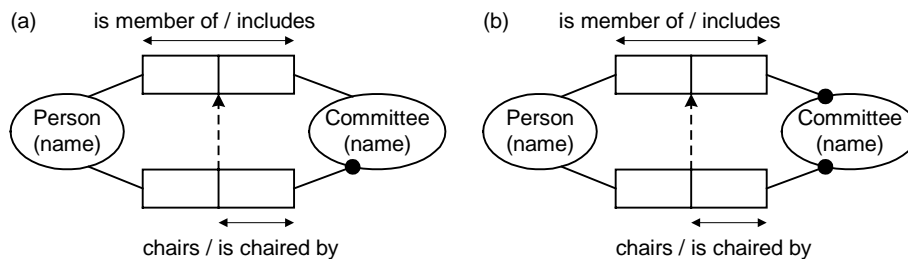


Figure 4: Schema (a) has an implied mandatory role constraint, shown explicitly in (b)

Although display options for implied constraints may sometimes be a matter of taste, practical experience has shown that in cases like this is better to show implied constraints explicitly rather than expect modelers or domain experts participating in the modeling process to figure them out for themselves. If you enter the schema of Figure 4(a) in VisioModeler, and attempt to build the logical dictionary, the tool will detect the misleading nature of the constraint pattern and ask you to resolve the problem. Human interaction is the best policy here, since there is more than one possible mistake (e.g. is the subset constraint correct or is the optional role correct?). Clicking on the error message throws you back into the conceptual schema with a red arrow highlighting the problem for you to fix (e.g. add the mandatory role constraint).

Note that if the schema of Figure 4(b) is mapped to a relational database it generates a referential cycle, since the mandatory fact types for Committee map to different tables (so each committee must appear in both tables). Referential cycles can be messy to work with, so VisioModeler warns you about this, but still generates the code to cope with it. The relational schema diagram generated by VisioModeler is shown in Figure 5 (the arrows show the foreign key references, one simple and one composite, that correspond to the subset constraints).

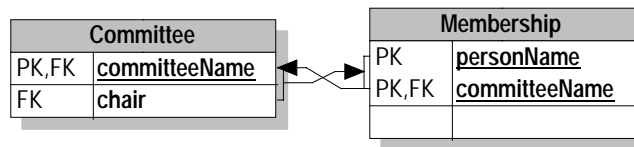


Figure 5: The relational schema mapped from the schema of Figure 4(b)

As another constraint example in UML, consider Figure 6, which is the UML version of an OMT diagram used in [1, p. 68] to illustrate a subset constraint between associations. See if you can spot any problems with the constraints.

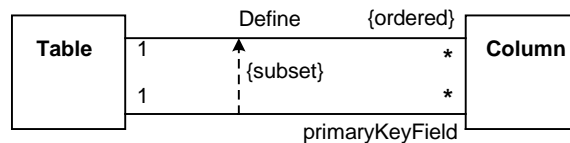


Figure 6: Spot anything wrong?

There are some fairly 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 to have 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 PK association and subset constraint could be replaced by a boolean isaPKfield attribute on Column.

From an ORM perspective, heuristics lead us to initially model the situation using natural reference schemes as shown in Figure 7. 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, 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).

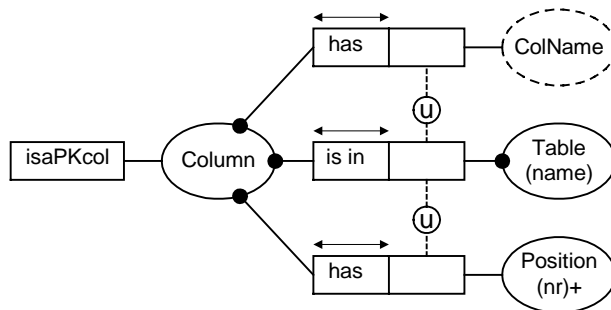


Figure 7

What is interesting about this example is not that the authors of the earlier model may have made some trivial errors with constraints (I’ve made slips of the pen like that in some of my book examples too), but rather the difference in modeling approaches. Most OMT and 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.

One other feature of the example is worth mentioning. The UML solution in Figure 6 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 debate has been going on for several years on the best way to deal with constructors (e.g. set, bag, sequence, unique sequence) at the conceptual level. My view (and that of several other ORM researchers) is that such constructors should not appear in the base conceptual model. Hence the use of Position in Figure 7 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. I may have more to say about this in a later article.

Equality constraints

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 8. 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 very 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).

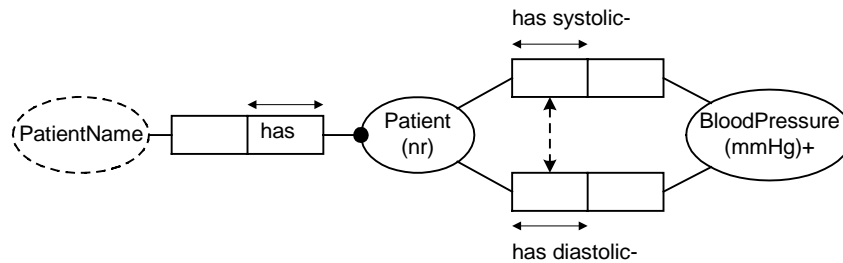


Figure 8: A simple equality constraint

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 normally be modeled as attributes of patient, and hence a textual constraint is attached to the Patient class as shown in Figure 9. Like UML textual subset constraints, this is awkward compared to the corresponding ORM constraint (graphic or verbalized).

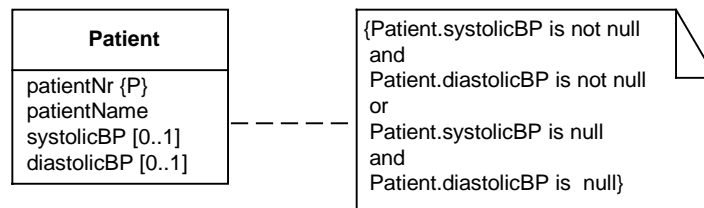


Figure 9: A simple subset constraint in UML

Subset and equality constraints enable various classes of schema transformations to be stated in their most general form, and ORM's more general support for these constraints allows more transformations to be easily visualized. For example, Figure 10 depicts equivalence PSG2 [2, p. 331].

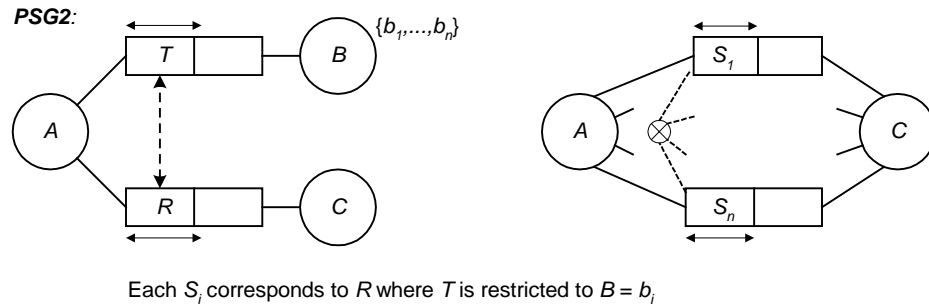


Figure 10: A basic schema equivalence in ORM

Later issues

Later issues will discuss other advanced graphic constraints in ORM and UML (join, ring, 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. Halpin, T. 1995, *Conceptual Schema and Relational Database Design, 2nd edn*, Prentice Hall Australia.
3. OMG-UML v1.2, OMG UML Revision Task Force website, <http://uml.systemhouse.mci.com/>.

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

UML data models from an ORM perspective: Part 6

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

This paper first appeared in the December 1998 issue of the Journal of Conceptual Modeling, published by InConcept.

This paper is the sixth in a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 discussed historical background, design criteria for modeling languages, object reference and single-valued attributes. Part 2 covered multi-valued attributes, basic constraints, and instantiation using UML object diagrams or ORM fact tables. Part 3 compared UML associations and related multiplicity constraints with ORM relationship types and related uniqueness, mandatory role and frequency constraints, as well as instantiation of associations. Part 4 contrasted ORM nesting with UML association classes, ORM co-referencing with UML qualified associations, and ORM exclusion constraints with UML or-constraints. Part 5 discussed ORM subset and equality constraints, and how to specify them in UML. Part 6 examines subtyping in ORM and in UML.

Subtyping: semantics and motivation

Both UML and ORM support *subtyping*, using substitutability (“is-a”) semantics, where each instance of a subtype is also an instance of its supertype(s). For example, declaring Woman to be a subtype of Person entails that each woman is a person, and hence Woman inherits all the properties of Person. Given two object types, *A* and *B*, we say that *A* is a *subtype* of *B* if, for each state of the database, the population of *A* is included in the population of *B*. For data modeling, the only subtypes of interest are *proper* subtypes. We say that *A* is a proper subtype of *B* if and only if (i) *A* is a subtype of *B*, and (ii) there is a possible state where the population of *B* includes an instance that is not in *A*. We could have a database state in which all people are women, and another in which some people are men, but we never have a state in which a woman is not also a person. From now on, we use “subtype” as short for “proper subtype”.

In both ORM and UML, *specialization* is the process of introducing subtypes, and *generalization* is the inverse procedure of introducing a supertype. Both ORM and UML allow single *inheritance* as well as multiple inheritance (where a subtype has more than one direct supertype). For example, AsianWoman may be a subtype of both AsianPerson

and Woman. In UML, “subclass” and “superclass” are synonyms of “subtype” and “supertype” respectively, and generalization may also be applied to things other than classes (e.g. interfaces, use case actors and packages). We confine our attention here to subtyping between object types (classes).

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). An operation implements a service and has a signature (name and formal parameters) and visibility, but may be realized in different ways. A method is an implementation of an operation, and hence includes both a signature and a body detailing an executable algorithm to perform the operation. In an inheritance graph, there may be many methods for the same operation (*polymorphism*), and scoping rules are used to determine which method is actually used for a given class. If a subclass has a method with the same signature as a method of one of its supertypes, this is used instead for that subclass (*overriding*). For example, if Rectangle and Triangle are subclasses of Shape, all three classes may have different methods for display(). Since the focus of this series of articles is on data modeling, not behavior modeling, we restrict our attention to inheritance of static properties (attributes and associations), typically ignoring operations or methods.

Subtypes are used in data modeling to do at least one of the following:

- assert typing constraints
- encourage reuse of model components
- show a classification scheme (taxonomy)

In this context, typing constraints ensure that subtype-specific roles are played only by the relevant subtype. As we will see, ORM has a stronger approach to typing constraints than UML. Both approaches use subtyping for reuse. Since a subtype inherits the properties of its supertype(s), only its specific roles need to be declared when it is introduced. Apart from reducing code duplication, the more generic supertypes are likely to find reuse in other applications. At the coding level, inheritance of operations/methods augments the reuse gained by inheritance of roles/attributes/associations. 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.

Display of subtypes

Data modeling approaches typically depict subtyping graphically using either Euler diagrams or Directed Acyclic Graphs. *Euler diagrams* depict relationships between subtypes spatially (unlike Venn diagrams, with which they are sometimes confused). For example, Figure 1 depicts the following information: *B*, *C* and *D* are subtypes of *A*, and *E* is a subtype of both *C* and *D*. Moreover, *B* overlaps with *C* (i.e. they may have a common instance) and *C* overlaps with *D*, but *B* and *D* are *mutually exclusive* (cannot have a

common instance). For example: $A = \text{Person}$; $B = \text{Asian}$; $C = \text{Consultant}$; $D = \text{American}$; $E = \text{TexanConsultant}$.

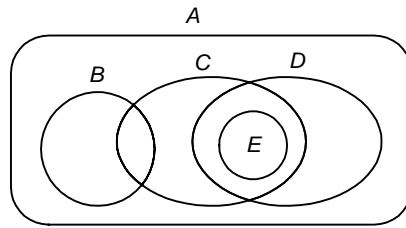


Figure 1: An Euler diagram

Euler diagrams provide intuitive displays for simple cases, and are used in some ER modeling tools (e.g. Designer/2000). However Euler diagrams are too cumbersome for complex subtype patterns often found in real applications, where an object type might have a large number of subtypes, possibly overlapping. Moreover, individual subtypes may have many specific details recorded for them, and there is simply no room to attach these details if the subtype nodes are crowded inside their supertype nodes.

For such reasons, Euler diagrams are eschewed for non-trivial subtyping. Instead *directed acyclic graphs* (DAGs) are often used, as is the case for both ORM and UML. A directed graph is simply a graph of nodes with directed connections, and “acyclic” means there are no cycles (a consequence of proper subtyping). The subtype pattern in Figure 1 is represented in DAG form in Figure 2, with (a) ORM and (b) UML versions shown. Here an arrow from one node to another shows that the first is a subtype of the second. ORM uses a solid shaft and arrowhead, while UML uses a thin arrow shaft with an open arrowhead. As an alternative notation, UML also allows separate shafts to merge into one, with one arrowhead acting for all (see Figure 3 later). Since subtyping is transitive, indirect connections are omitted (e.g. since E is a subtype of C , and C is a subtype of A , it follows that E is a subtype of A , so there is no need to display this implied connection).

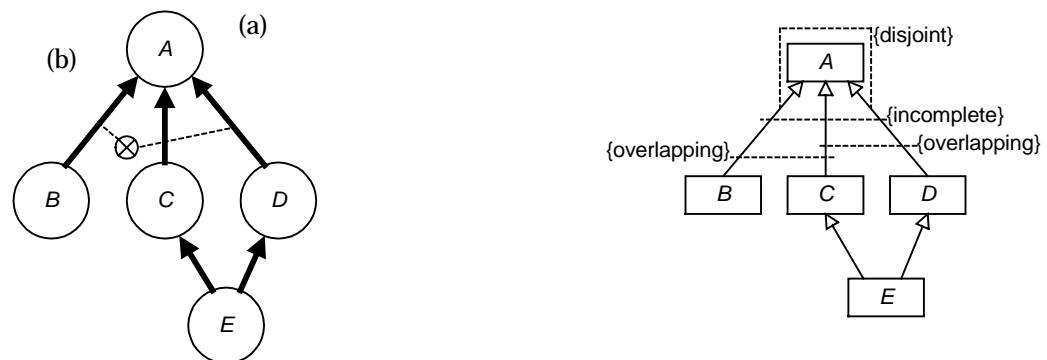


Figure 2: Previous Euler subtype diagram depicted as directed acyclic graphs in (a) ORM and (b) UML

As shown, ORM and UML both show subtypes outside, connected by arrows to their supertype(s). Although less intuitive than Euler diagrams, this is preferable since it allows us to express subtype patterns of any complexity, with enough space at each subtype node to add specific details. By default, ORM subtypes may overlap, and subtypes need not collectively exhaust their supertype. However ORM allows graphic constraints to be added to indicate that subtypes are mutually exclusive (a circled “X” connected to the relevant subtypes via dotted lines, as in Figure 2, collectively exhaustive (a circled dot) or both (a circled, crossed dot). Exhaustion constraints are also called “totality constraints”. Although exclusion and totality constraints for subtypes are part of ORM, VisioModeler does not yet support them. As we will see presently, explicit depiction of such constraints is not a necessity, since other constraints in conjunction with formal subtype definitions typically imply the relevant exclusion and totality constraints.

In UML the only default for generalization appears to be “incomplete”, i.e. not all subtypes have been specified. The opposite of “incomplete” is “complete”—this probably means the same as totality (collective exhaustion) in ORM, i.e. the supertype equals the union of its subtypes. However the wording in the UML standard [7] does not make this clear. In UML, constraint keywords may be added in braces besides dotted lines connecting the relevant subtypes, as shown in Figure 2. The following four keywords are predefined in UML for this purpose: “overlapping” (the subtypes overlap), “disjoint” (the subtypes are mutually exclusive), “complete” (all subtypes have been declared), and “incomplete” (some more subtypes may be introduced later). Other keywords may be added by users. When more than one arrowhead is involved, UML requires the keyword to be written beside a single dotted line that connects the relevant subtypes. I have assumed that this line may include elbows (as shown in Figure 2 for the disjoint constraint); without elbows or a similar device, some cases can’t be specified.

As Figure 2 shows, ORM’s depiction of inter-subtype constraints is less cluttered than UML’s. ORM’s approach is that exclusion and totality constraints are enforced on populations, not types. For example, an overlapping “constraint” does not mean that the populations must overlap, just that they may overlap. Hence from an ORM viewpoint, this is not really a constraint at all, so there is no need to depict it.

For any subtype graph, the topmost supertype is called the *root*, and the bottom subtypes (those with no descendants) are called *leaves*. In UML this can be made explicit by adding “{root}” or “{leaf}” below the class name. If we know the whole subtype graph is shown, there is little point in doing this; but if we were to display only part of a subtype graph, this notation makes it clear whether or not the local top and bottom nodes are also like that in the global schema. For example, from Figure 3, we know that globally Customer has no supertype, and MalePerson and FemalePerson have no subtypes. However, since Organization has not been marked as a leaf node, it may have other subtypes not shown here. Currently ORM does not include such a root/leaf notation (apart from adding a text box with this information), but it would be simple to add it. UML also allows an ellipsis “...” in place of a subclass to indicate that at least one subclass of the parent exists in the global schema, but its display has been suppressed on the diagram.

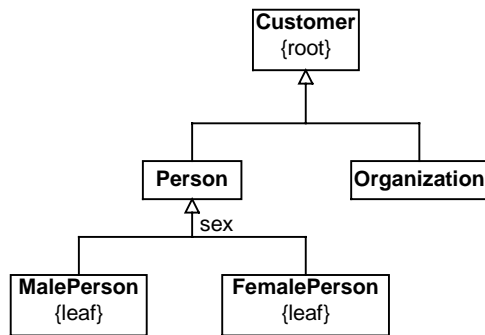


Figure 3: An incomplete UML subtype graph: Organization may have other subtypes not shown here

UML also distinguishes between *abstract* and *concrete* classes. An abstract class cannot have any direct instances, and is shown by writing its name in italics or by adding “{abstract}” below the class name. Abstract classes are realized only through their descendants. Concrete classes may be directly instantiated. This distinction seems to have little relevance at the conceptual level, and is not depicted explicitly in ORM. For code design however, the distinction is important (e.g. abstract classes provide one way of declaring interfaces, and in C++ abstract operations correspond to pure virtual operations, while leaf operations map to nonvirtual operations). For further discussion, see [3, pp. 85-8] and [1, pp. 125-6].

Subtype definitions

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 3 includes a “sex” discriminator to specialize Person into MalePerson and FemalePerson. The UML standard [7] says that the discriminator names a “a partition of the subtypes of the superclass”. In formal work, the term “partition” usually implies the division is both exclusive and exhaustive. In UML, the use of a discriminator does not imply that the subtypes are exhaustive or complete, but it does seem that they must be exclusive (e.g. [3], p. 78). If that is the case, there does not appear to be any way in UML of declaring a classification scheme for a set of overlapping subtypes. The same discriminator name may be repeated for multiple subclass arrows to show that each of these subclasses belong to the same classification scheme. This repetition can be avoided by merging the arrow shafts to end in a single arrowhead, as in Figure 3.

The UML standard states that “the discriminator must be unique among the attributes and association roles of the given superclass” but I’m unsure what this means. If it implies that an attribute or association role can’t be used as a discriminator, then that would seem to be bizarre. If it doesn’t imply this, then the notation is open to inconsistency. As a trivial example, consider the Patient subtyping in Figure 4, where the sex attribute is used as a discriminator. This attribute is based on the enumerated type Sexcode, which has been defined using the stereotype «enumeration», and listing its values as attributes.

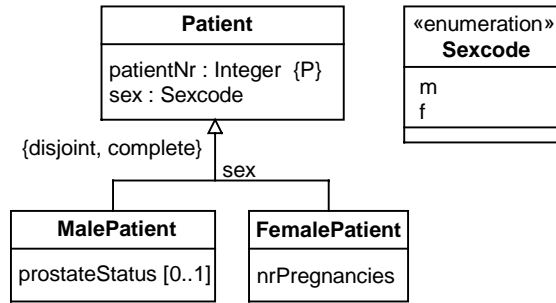


Figure 4

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. This problem is best explained using ORM, where it's easy to display populations. Figure 5 shows an equivalent ORM schema, with prostate status being measured for a female, and pregnancies being recorded for one of the males. This kind of nonsense is allowed because the model hasn't formally related the subtypes back to their precise sex.

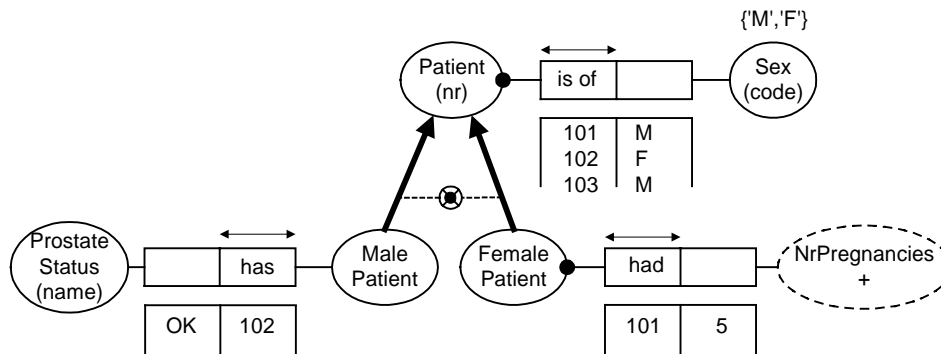
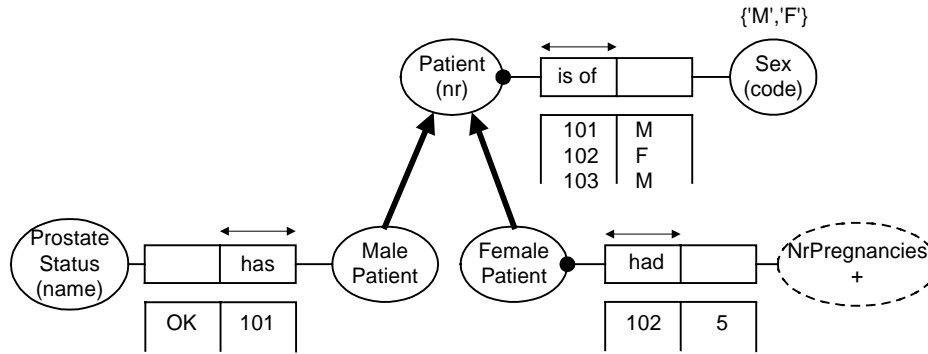


Figure 5: What is the problem here?

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). The correct schema is shown in Figure 6, together with a satisfying population. Note that the ORM partition (exclusion and totality) constraint has been removed from the diagram since it is now implied by the combination of the subtype definitions and the three constraints on the fact type Patient-is-of-Sex. Though long part of ORM, formal subtype definitions are not yet supported by VisioModeler, which allows them to be entered only as comments. However the conceptual query technology underlying ActiveQuery potentially provides one way of formally defining and mapping subtypes, and the related formal theory is mature.



each MalePatient is a Patient who is of Sex 'M'
each FemalePatient is a Patient who is of Sex 'F'

Figure 6: Formal subtype definitions are needed, and subtype partition constraints are implied

While the subtype definitions in Figure 6 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*, and now define *LargeUSCity* as follows:

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.

This article has ignored various subtyping issues such as mapping and context-dependent reference. For an ORM perspective on these and related issues see [4, 5, 6].

Later issues

Later issues will discuss other advanced graphic constraints in ORM and UML (ring, join, aggregation etc.), derivation rules and queries.

References

1. Booch, G., Rumbaugh, J. & Jacobson, I. 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading MA, USA.
2. Blaha, M. & Premerlani, W. 1998, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, New Jersey.

3. Fowler, M. with Scott, K. 1997, UML Distilled, Addison-Wesley.
4. Halpin, T. 1995, Conceptual Schema and Relational Database Design, 2nd edn, Prentice Hall Australia.
5. Halpin, T.A. 1995, 'Subtyping: conceptual and logical issues', Database Newsletter, ed. R.G. Ross, Database Research Group Inc., vol. 23, no. 6, pp. 3-9, reproduced by permission on www.orm.net.
6. Halpin, T. & Proper, H. 1995, 'Subtyping and polymorphism in object-role modelling', Data & Knowledge Engineering 15, 3 (June), 251-281, reproduced by permission on www.orm.net.
7. OMG-UML v1.2, OMG UML Revision Task Force website, <http://uml.systemhouse.mci.com/>.

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

UML data models from an ORM perspective: Part 7

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

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

This paper is the seventh in a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 discussed historical background, design criteria for modeling languages, object reference and single-valued attributes. Part 2 covered multi-valued attributes, basic constraints, and instantiation using UML object diagrams or ORM fact tables. Part 3 compared UML associations and related multiplicity constraints with ORM relationship types and related uniqueness, mandatory role and frequency constraints, as well as how associations may be instantiated. Part 4 contrasted ORM nesting with UML association classes, ORM co-referencing with UML qualified associations, and ORM exclusion constraints with UML or-constraints. Part 5 discussed ORM subset and equality constraints, and how to specify these in UML. Part 6 discussed subtyping. Part 7 discusses some other graphic constraints (value, ring and join constraints).

Value constraints

An ORM *value constraint* restricts the population of a value type to a finite set of values specified either in full (*enumeration*), by start and end values (*range*), or some combination of both (*mixture*). The values themselves are primitive data values, typically character strings or numbers. The constraint is shown by declaring the possible values in braces besides either the value type, or an entity type with a reference mode. In the latter case, the constraint is understood to apply to the implicit value type. For example, in Figure 1 the constraints apply to Sexcode, RatingNr and SQLchar.

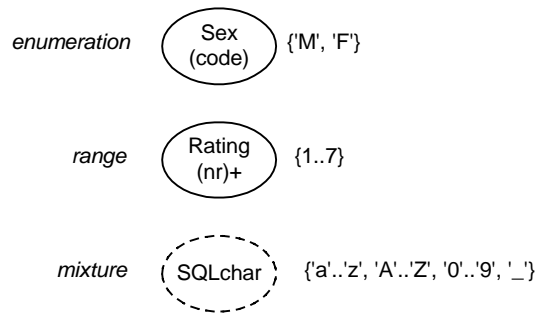


Figure 1: Value constraints in ORM

In UML, enumeration types may be modeled as classes, stereotyped as enumerations, with their values listed (somewhat unintuitively) as attributes. Ranges and mixtures may be specified by declaring a textual constraint in braces, using any formal or informal language. For example, see Figure 2.

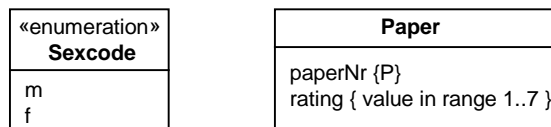


Figure 2: Data value restrictions declared as enumerations or textual constraints in UML

Value constraints other than enumeration, range and mixture may be declared in either ORM or UML as textual constraints, e.g. {committeeSize must be an odd number}. For further UML examples, see [5, pp. 236, 268].

Ring constraints

A ring fact type has at least two roles played by the same object type (either directly, or indirectly via a supertype). ORM allows various *ring constraints* to be applied to such role-pairs. For example, in Figure 3, the isParentOf association is declared to be acyclic (^oac) and intransitive (^oit). Here “acyclic” means nobody can be one of his/her own descendants. A satisfying population is shown in the fact table below the schema. In the population graph shown at the right of the figure, people are denoted by circular nodes containing the first letter of their name, and the directed arrows denote the “is parent of” relationship. The acyclic constraint means there can’t be any cycles or loops in the graph.

In this example, “intransitive” means nobody is a parent of any of his/her grandchildren. In terms of the graph, it means we can’t add any arrows that jump over one node to provide an alternate path to the target node. By default, ORM constraints are “*hard*”, meaning no violation is permitted. If we did accept that incest might occur in the UoD, and wanted to record any cases of it, this intransitive constraint should be downgraded to a “*soft* constraint”, where violations are accepted but other action is taken to minimize their occurrence (e.g. send message to police).

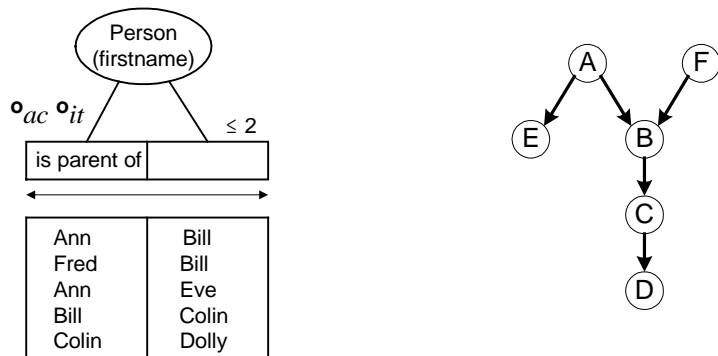


Figure 3: Some ring constraints in ORM

ORM provides six built-in ring constraints: antisymmetric ($^{\circ}ans$), asymmetric ($^{\circ}as$), acyclic ($^{\circ}ac$), irreflexive ($^{\circ}ir$), intransitive ($^{\circ}it$), and symmetric ($^{\circ}sym$). Because of their underlying logic, various implications exist between the constraints, and some combinations are impossible. To save you having to worry about these complexities, I designed the ring-constraint interface for VisioModeler so that you can't enter a ring constraint that is implied by, or incompatible with, one that you have already chosen (see Figure 4).

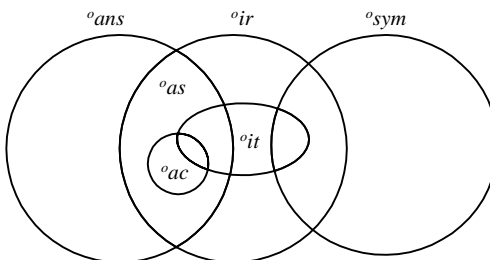


Figure 4: Ring constraint interface in VisioModeler

If you are mapping your model to a relational database, some ring constraints are very efficiently enforced. For example, irreflexivity typically maps to a simple check clause like “**check** (parent <> child)”. On the other hand, some ring constraints can be very expensive (e.g. acyclicity). In this case, a conscious decision needs to be taken as to whether to have the constraint enforced at all by the system (e.g. in batch mode overnight) or to have users instructed that they are responsible for enforcing the constraint.

UML does not provide ring constraints built-in, so the modeler needs to specify these as a textual constraint in some chosen language. In UML, if a textual constraint applies to just one model element (e.g. an association path), it may be added in braces beside that element. For example, the ORM parenthood schema might be recast in UML as shown in Figure 5(a). It is the responsibility of the software tool (used to work with the diagram) to ensure the constraint is linked internally to the relevant model element, and to interpret any textual constraint expressions. If the tool cannot interpret the constraint, it should be placed inside a note, without braces, showing that it is merely a comment, and explicitly linked to the relevant model element, as shown in Figure 5(b).

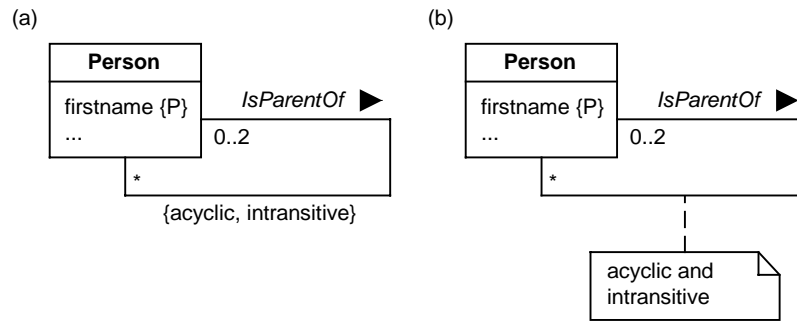


Figure 5: Ring constraints expressed in UML as (a) textual constraints and (b) comments

Join constraints

In ORM, a *join* constraint applies to one or more role sequences, at least one of which is projected from a *path* from one predicate through an object type to another predicate. The act of passing from one role through an object type to another role invokes a *conceptual join*, since the same object instance is asserted to play both the roles. The external uniqueness constraint (discussed in an earlier article) is actually a very simple case of this, in which there is just one argument. For example, in Figure 6 suppose we start at Employee, then follow the path to Date. This gives us the path: Employee was issued a ParkPermit that was issued on a Date. The “that” in this path expression asserts that the parking permit issued to the employee is the *same* one issued on the date. This identity claim is a conceptual join—like an equi-join in relational theory, except that it is over objects, not attribute values. In a later issue, we briefly discuss how such path expressions are used in ConQuer, an ORM conceptual query language. Now that the path is known, we project on the first and last roles (those played by Employee and Date) and assert uniqueness over this combination. In other words, a given employee on a given date can be issued at most one parking permit. This is the most fundamental way to understand external uniqueness constraints.

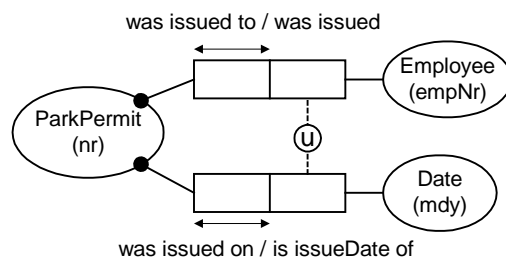


Figure 6: An external uniqueness constraint is a simple join constraint over one path

Role sequences featuring as arguments in set comparison constraints (subset, equality, exclusion) may also arise from projections over a join path. For example, in Figure 7, the subset constraint runs from the (Room, Facility) role-pair projected from the path: Room at a Time is used for an Activity that requires a Facility. This path includes a conceptual join on Activity. Since the subset constraint involves at least one join path, it is called a *join-subset constraint*. The constraint may be verbalized as: **if a Room at a Time is used for an Activity that requires a Facility then that Room provides that Facility**.

This example is based on a room scheduling application at a university with built-in facilities in various lecture and tutorial rooms (PA = Personal Address system, DP = Data Projection facility, INT = Internet access). Figure 7 includes a satisfying population for the three fact types.

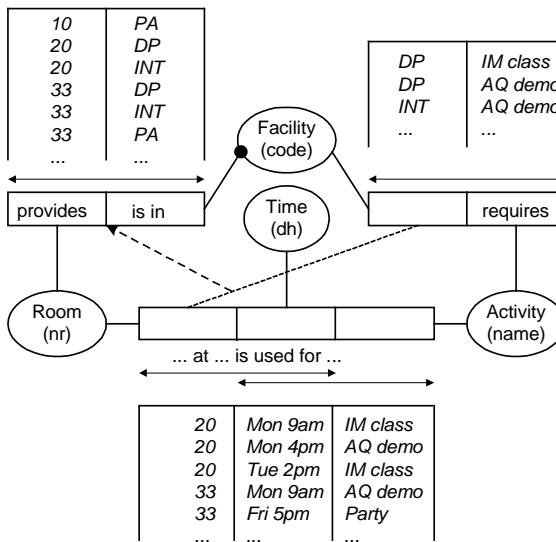


Figure 7: A join-subset constraint in ORM

Although join constraints arise frequently in real applications, UML has no graphic symbol for them. Nevertheless, they may be declared on UML diagrams by writing a textual constraint or comment in a note (dog-eared rectangle), attached by a dashed line to the model elements involved (here, three associations). Figure 8 uses a comment.

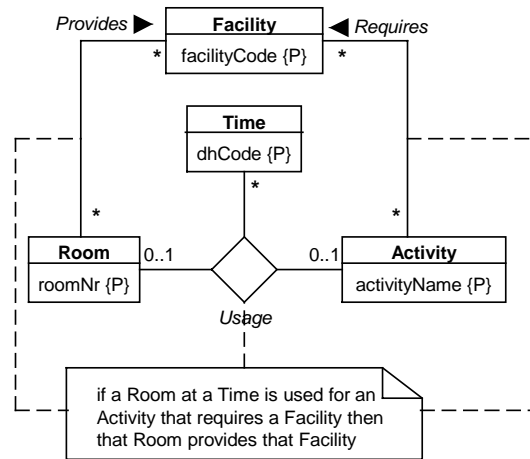


Figure 8: Join constraint specified as a comment in UML

Figure 7 again illustrates how ORM facilitates validation constraints via sample populations. The UML associations in Figure 8 are not so easily populated on the diagram. When attributes are used, the situation worsens considerably. As another example, consider the UML Employee class shown in Figure 9. This is nice and compact, but it makes it hard to express the common business rule that certain titles apply to only one sex (e.g. Lady applies only to females). In ORM this can be captured by a populated join-subset constraint as shown in the right hand side of the figure. In ConQuer, this constraint verbalizes as: if Person1 has a Title that applies only to Sex1 then Person1 is of Sex1. Step 5b of ORM’s conceptual schema design procedure prompts the modeler to add the extra association between Title and Sex, and in this case the population becomes part of the rule. It is unclear as to how to approach this problem in UML, other than by converting title and sex to classes and writing down a population somewhere in a note.

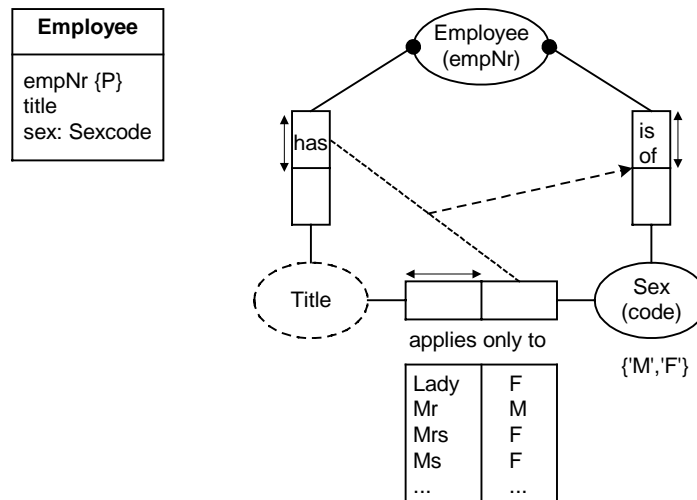


Figure 9: ORM makes it easy to capture the constraint between title and sex

As an example of a join-exclusion constraint, consider the following rule from a conference paper review application: no Person who works at an Institute that employs a Person who wrote a Paper may review that paper. As discussed in a later article, subset and equality constraints also provide one way of specifying derivation rules. In the absence of further marks on the schema diagram, ORM join constraint paths may sometimes be ambiguous. This problem may easily be resolved by having the modeler indicate the path in some way (e.g. by shift-clicking the predicates on the path) and then displaying this path in some way when the constraint is inspected (e.g. by shading).

Later issues

Later issues will discuss aggregation, initial value declarations, derivation rules and changeability settings in ORM and UML.

References

1. Booch, G., Rumbaugh, J. & Jacobson, I. 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading MA, USA.
2. Blaha, M. & Premerlani, W. 1998, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, New Jersey.
3. Halpin, T. 1995, *Conceptual Schema and Relational Database Design, 2nd edn (revised 1999)*, WytLytPub, Bellevue WA, USA.
4. OMG-UML v1.2, OMG UML Revision Task Force website, <http://uml.systemhouse.mci.com/>.
5. Rumbaugh, J., Jacobson, I. & Booch, G. 1999, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading MA, USA.

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

UML data models from an ORM perspective: Part 8

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

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

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

Updates to the UML standard

Recently Visio became a member of the Object Management Group (OMG), and began participating in the ongoing work to refine the UML standard. Within the OMG, the UML standard is the responsibility of the Analysis and Design Task Force (ADTF, formerly OOA&DTF), chaired by Jim Odell and Cris Kobryn. Minor changes to the UML standard that lead to point releases (e.g. 1.1, 1.2, 1.3) are managed by a subgroup of the ADTF known as the UML RTF (Revision Task Force), chaired by Cris Kobryn. The latest release of UML (version 1.2) is fully supported by Visio Enterprise, including all nine diagram types. Currently, the UML RTF is working on a draft of version 1.3, and some further point releases might be considered later (e.g. version 1.4). The next major release (2.0) is not expected to be forthcoming from the ADTF for quite some time (e.g. late 2001). As a result of recent email discussions and meetings of the UML RTF team, several revisions to the UML 1.3 draft have been made, including two that I will comment on here, since they relate to issues discussed in earlier articles in this series.

The “{or}” constraint discussed in Part 4 of this series has been renamed “{xor}” (short for “exclusive or”), and has been redefined to mean *exactly one* of the association roles is chosen. This means it is equivalent to ORM’s exclusive-or constraint, which is a combination of a disjunctive mandatory role constraint and an exclusion constraint. For example, consider the following constraint

- (1) each Vehicle is either leased from a Company or was purchased from a Company, but not both.

In ORM, this may be expressed by the following two constraints:

- (2) **each** Vehicle is leased from **a** Company **or** was purchased from **a** Company.
- (3) **no** Vehicle is leased from **a** Company **and** was purchased from **a** Company.

Constraint (2) is a disjunctive mandatory role constraint, shown as a black dot on the object type connected to the two roles, or by a circled black dot “⊙” connected to the roles. Constraint (3) is an exclusion constraint, shown as a circled ex “⊗” connecting the two roles. The constraints are orthogonal, and may be shown either separately as in Figure 1(a) or by combining the two symbols as in Figure 1 (b).

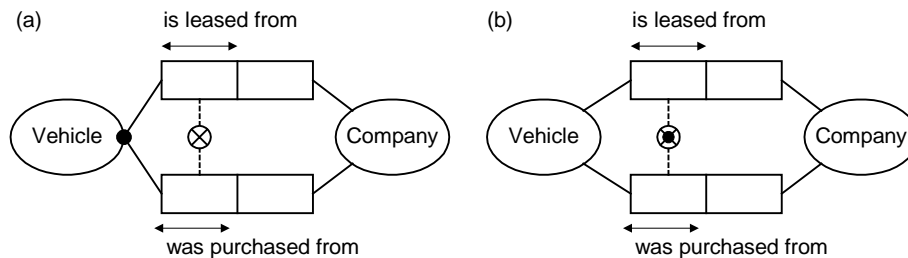


Figure 1: Exclusive-or constraint depicted in ORM using (a) separate or (b) combined symbols

In UML, the constraint is displayed by connecting the relevant association-ends (roles in ORM) by a dashed line, labeled “{xor}” (see Figure 2). Although the current wording of the UML standard describes the constraint as applying to a set of associations, we need to apply the constraint to a set of association-ends to avoid ambiguity in cases like this with multiple common classes. Visually this could be shown by attaching the dashed lines near the relevant ends of the associations, as we have done here.

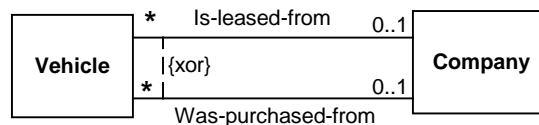


Figure 2: Exclusive-or constraint depicted in UML

As discussed in previous issues, UML has no symbols for exclusion or disjunctive mandatory role constraints. If ever UML symbols for these constraints are considered, then “{x}” and “{or}” respectively seem appropriate—this choice also exposes the composite nature of “{xor}”. Even if such a proposal were accepted as a UML extension, this would capture only a fragment of ORM’s expressive power in this area—recall that ORM’s exclusion constraint applies not just to a set of roles, but a set of role-sequences, and hence is far more general than the kind of case considered here. Moreover, ORM roles include unary predicates, and ORM needs no additional notations to constrain attributes.

The second proposed revision to UML concerns the semantics of the “{complete}” constraint for subtyping. This constraint, discussed in Part 6, was formerly described as indicating that the modeler intended to add no more subtypes. This weak notion of completeness does not entail that the constrained subtypes collectively exhaust the supertype, but this latter notion is far more useful in practice and is called a totality constraint in ORM. Although typically implied by other constraints, a totality constraint may be explicitly depicted in ORM by connecting the mandatory symbol “ \odot ” to the relevant subtype links (it is mandatory for each instance of the supertype to be an instance of at least one of the subtypes). Hence the supertype equals the union of the constrained subtypes. Recall that a type is the set of all possible instances, while a population is the set of current instances. The practical way to enforce the constraint is to check that for each state of the database, the population of the supertype equals the union of the populations of the constrained subtypes. At the UML RTF meeting in March it was agreed that the UML notion of subtype completeness would be redefined as this set-theoretic notion, thus making it equivalent to ORM’s subtype totality constraint. With this understanding, the ORM and UML schemas in Figure 3 are equivalent.

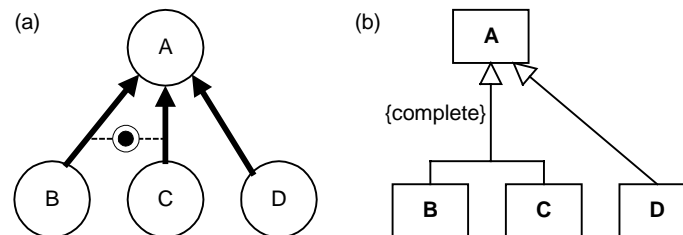


Figure 3: The subtype totality constraint $A = B \cup C$ expressed in (a) ORM and (b) UML.

Aggregation

In UML, the term “aggregation” is used to describe a whole/part relationship. For example, a team of people is an aggregate of its members, so this membership may be modeled as an aggregation association between Team and Person. Several different forms of aggregation might be distinguished in real world cases. For example, Jim Odell and Conrad Bock discuss the following six varieties of aggregation: component-integral; material-object; portion-object; place-area; member-bunch; and member-partnership [4, 5]. Currently, UML associations are classified into one of three kinds: ordinary association

(no aggregation); shared (or simple) aggregation; composite (or strong) aggregation. Hence UML version 1.x recognizes only two varieties of aggregation: shared and composite. Although early planning for UML version 2.x foreshadows further kinds of aggregation being introduced, we confine our attention here to shared and composite aggregation. Some versions of ER include an aggregation symbol (typically only one kind). ORM, as well as many versions of ER, includes no special symbols for aggregation.

These different stances with respect to aggregation are somewhat reminiscent of the different modeling positions with respect to null values. Although over twenty kinds of null have been distinguished in the literature, the relational model recognizes only one kind of null, Codd's version 2 of the relational model proposes two kinds of null, and ORM argues that nulls have no place in base conceptual models (because all its base facts are elementary). But let's return to the topic at hand.

Shared aggregation is denoted in UML as a binary association, with a hollow diamond at the "whole" or "aggregate" end of the association. Composition (composite aggregation) is depicted with a filled diamond. For example, Figure 4 depicts a composition association from Club to Team, and a shared aggregation association from Team to Person.

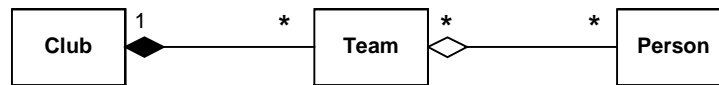


Figure 4: Composition (composite aggregation) and shared aggregation in UML

In ORM, this situation would be modeled as shown in Figure 5. As we see, ORM has no special notation for aggregation. Does Figure 4 convey any extra semantics, not captured in Figure 5? At the conceptual level, it is doubtful whether there is any additional useful semantics. At the implementation level however, there is additional semantics. Let's discuss this in more detail.

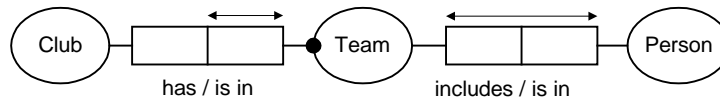


Figure 5: The Figure 4 example modeled in ORM

The UML standard declares that "both kinds of aggregation define a transitive ... relationship" [6]. The use of "transitive" here is somewhat misleading, since it refers to indirect aggregation associations rather than base aggregation associations. For example, if Club is an aggregate of Team, and Team is an aggregate of Person, it follows that Club is an aggregate of Person. However if we wanted to discuss this result, it should be exposed as a derived association. In UML, derived associations are indicated by prefixing their names with "/". The derivation rule can be expressed as a constraint, either connected to the association by a dependency arrow, or simply placed beside the association as in Figure 6.

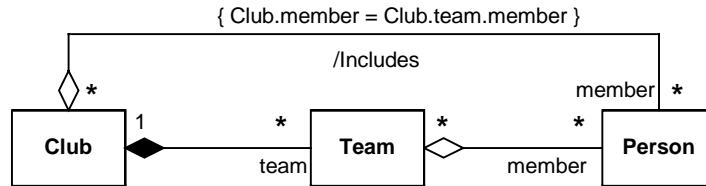


Figure 6: A derived aggregation in UML

In ORM, derived fact types may be diagrammed by marking them with an asterisk, and derivation rules may be specified in an ORM textual language such as ConQuer (see Figure 7). In many cases, derivation rules may also be diagrammed as a join-subset or join-equality constraint. As this example illustrates, the derived transitivity of aggregations can be captured in ORM without needing a special notation for aggregation.

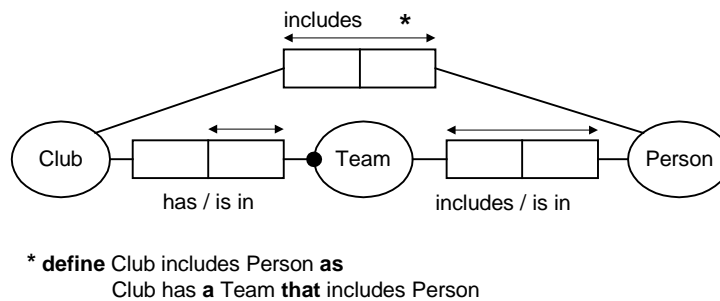


Figure 7: The derived aggregation of Figure 6 modeled in ORM

More fully, the UML standard declares that “both kinds of aggregation define a transitive, antisymmetric relationship (i.e. the instances from a directed, non-cyclic graph)” [6]. Recall that a relation R is antisymmetric if and only if, for all x and y , if x is not equal to y then xRy implies that yRx . It would have been better to simply state that paths of aggregations must be acyclic. At any rate, this rule is designed to stop errors such as that shown in Figure 8. If a person is part of a team, and a team is part of a club, it doesn’t make sense to say that a club is part of a person.

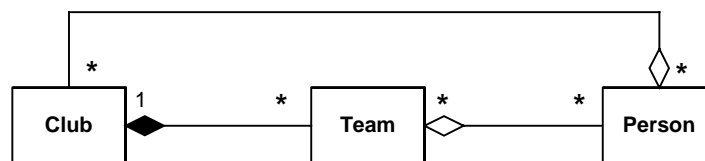


Figure 8: Illegal UML model. Aggregations should not form a cycle.

Since ORM does not specify whether an association is an aggregation, illegal diagrams like this can’t occur in ORM. Of course, it is possible for an ORM modeler to make a silly mistake by adding an association such as Club is part of Person, where “is part

of” was informally understood in the aggregation sense, and this would not be formally detectable. But avoidance of such a bizarre occurrence doesn’t seem to be a compelling reason to add aggregation to ORM’s formal notation. There are plenty of associations between Club and Person that do make sense, and plenty that don’t. In some cases however, it is important to assert constraints such as acyclicity, and this is handled in ORM by ring constraints (see Part 7).

Composition does add some important semantics to shared aggregation. To begin with, it requires that each part belongs to at most one whole at a time. In ORM, this is captured by adding a uniqueness constraint to the role played by the part (e.g. see the role played by Team in Figure 5). In UML, the multiplicity at the whole end of the association must be 1 or 0..1. If the multiplicity is 1 (as in Figure 4), the role played by the part is both unique and mandatory (as in Figure 5). As an example where the multiplicity is 0..1 (i.e. where a part optionally belongs to a whole), consider the ring fact type of Figure 9: Package contains Package. Here “contains” is used in the sense of “directly contains”. The UML standard notes that “composition instances form a strict tree (or rather a forest)” [6]. This strengthening from directed acyclic graph to tree is an immediate consequence of the functional nature of the association (each part belongs to at most one whole), and hence ORM requires no additional notation for this. In this example, the ORM model explicitly includes an acyclic constraint. Note that this direct containment association is intransitive by implication (acyclicity implies irreflexivity, and any functional, irreflexive association is intransitive).

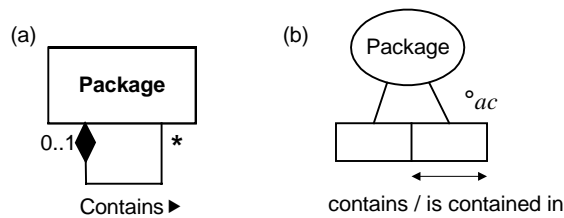


Figure 9: Direct containment modeled in (a) UML and (b) ORM

UML allows some alternative notations for aggregation. If a class is an aggregate of more than one class, the association lines may be shown joined to a single diamond (see Figure 10(a)). For composition, the part classes may be shown nested inside the whole by using role names, and multiplicities of components may be shown in the top right hand corners (see Figure 10(b)).

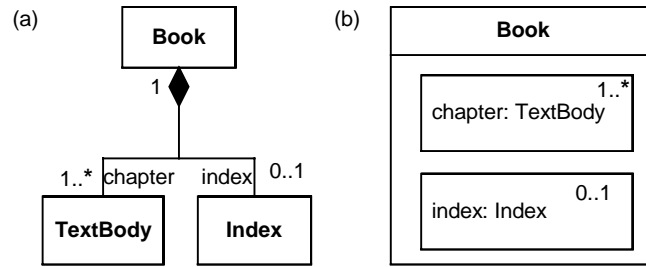


Figure 10: Alternative UML notations for aggregation

Some authors list kinds of association that are easily confused with aggregation but should not be modeled as such (e.g. topological inclusion, classification inclusion, attribution, attachment and ownership [4, 5]). For example, Finger belongs to Hand is an aggregation, but Ring belongs to Finger is not. There is some disagreement among authors about what should be included on this list. For example, attribution is treated by some as a special case of aggregation (a composition between a class and the classes of its attributes) [7]. My own viewpoint is that for conceptual modeling purposes, agonizing over such distinctions doesn't seem to be worth the trouble. This position seems to be taken by some other authors. For example, [7, p. 148] argues that "Aggregation conveys the thought that the aggregate is inherently the sum of its parts. In fact, the only real semantics that it adds to association is the constraint that chains of aggregate links may not form cycles ... Some authors have distinguished several kinds of aggregation, but the distinctions are fairly subtle and probably unnecessary for general modeling".

Indeed there seems little justification for introducing the notion of aggregation at all as a separate concept at the conceptual level. There are plenty of other distinctions (apart from aggregation) we could make about associations, but we don't feel compelled to do so. At the implementation level however, composite aggregation does add important semantics. "A composite implies propagation semantics ... For example, if the whole is copied or deleted, then so are the parts as well" [6]. Clearly this dynamic semantics has nothing to do with a conceptual view of the domain area, and it would be unreasonable to introduce this notion when validating the business model with the subject matter expert. However, once a decision is made to implement the conceptual model in an object-oriented system, it is important to capture this semantics. One way of doing this would be to convert a conceptual ORM model to a UML model, and then add aggregation at that stage.

Obviously there are different stances one could take about how, if at all, aggregation should be included in the conceptual modeling phase. My position is one of many. You can decide what's best for you.

Later issues

Later issues will discuss default values, changeability settings, derived data, derivation rules and queries in ORM and UML.

References

1. Booch, G., Rumbaugh, J. & Jacobson, I. 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading MA, USA.
2. Blaha, M. & Premerlani, W. 1998, *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall, New Jersey.
3. Halpin, T. 1995, *Conceptual Schema and Relational Database Design, 2nd edn (revised 1999)*, WytLytPub, Bellevue WA, USA.
4. Martin, J. & Odell, J. 1998, *Object-Oriented Methods: a Foundation, UML edn*, Prentice Hall, Upper Saddle River, New Jersey. { Ch. 18 discusses aggregation }
5. Odell, J. 1998, *Advanced Object-Oriented Analysis & Design using UML*, Cambridge University Press, & SIGS Books, New York. { Part V (pp. 137-65) discusses aggregation }
6. OMG-UML 1.3 draft, OMG UML Revision Task Force website, <http://uml.systemhouse.mci.com/>.
7. Rumbaugh, J., Jacobson, I. & Booch, G. 1999, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading MA, USA.

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

UML data models from an ORM perspective: Part 9

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

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

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

Initial values

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

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

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

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

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

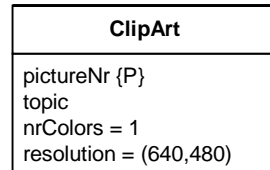


Figure 1: Attributes may be assigned initial values in UML

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

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

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

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

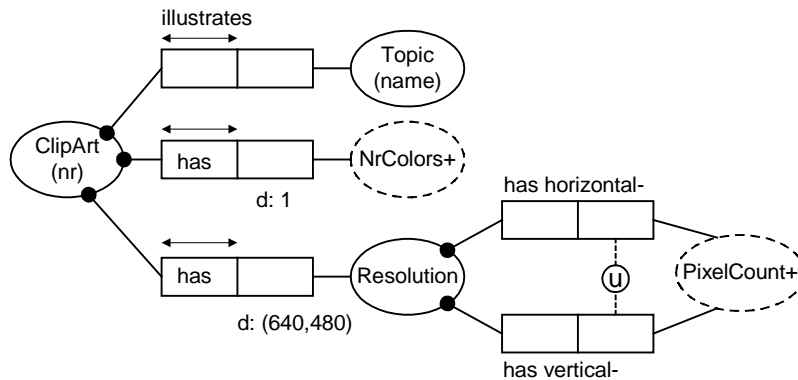


Figure 2: A possible extension to ORM to capture simple and composite default values

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

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

Derived data

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

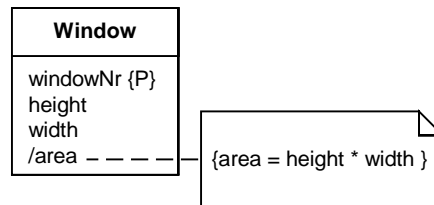


Figure 3: Area depicted as a derived attribute in UML, with derivation rule declared in a note

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

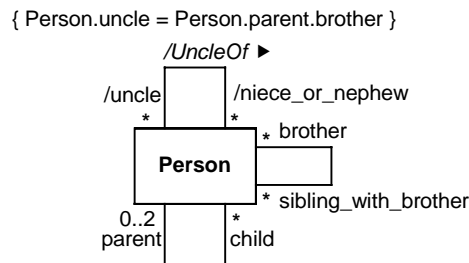


Figure 4: Derived uncle association (and roles) in UML, with derivation rule declared as a constraint

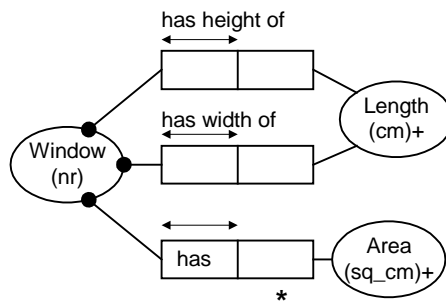
In ORM, a fact type that is primitive (i.e. not defined in terms of others) is said to be a *base* fact type. *Derived* fact types are defined in terms of other fact types (base or derived). If displayed on a diagram, derived fact types are marked with an asterisk. Constraints on derived fact types are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

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

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

```
Window has Area iff      Window has height of Length1 and
                          Window has width of Length2 and
                          Area = Length1 * Length2.
```

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



```
* { area = height x width }
define Window has Area(sq_cm)
as
  Window has height of Length1 and
  Window has width of Length2 and
  Area = Length1 * Length2
```

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

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

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

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

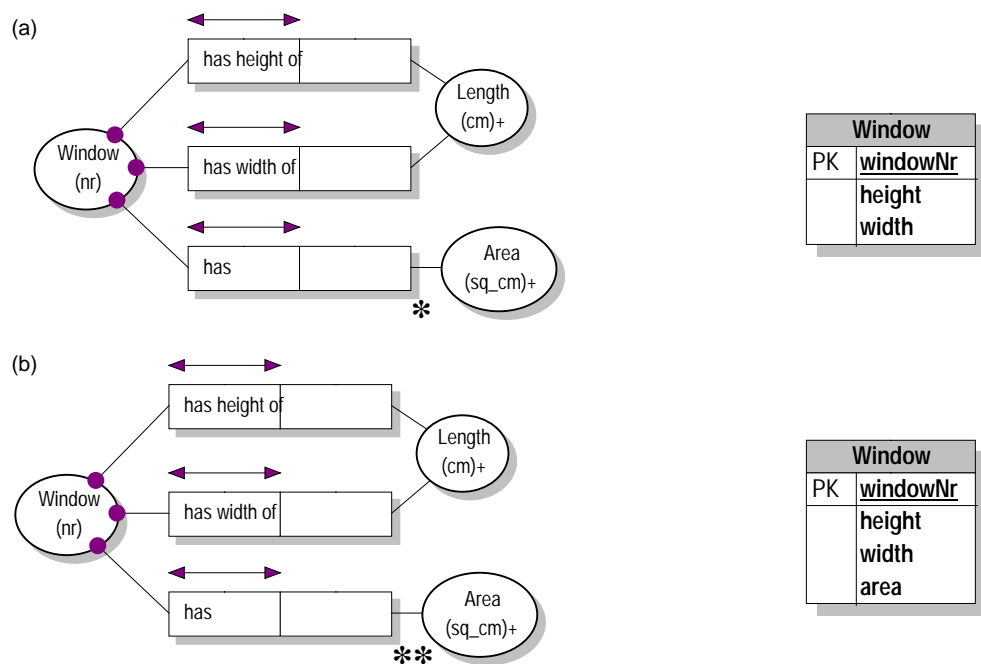


Figure 6 As an implementation issue, derived fact types may be evaluated lazily (a) or eagerly (b)

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

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

Next issue

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

References

- Booch, G., Rumbaugh, J. & Jacobson, I. 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading MA, USA.
- Halpin, T. 1995, *Conceptual Schema and Relational Database Design, 2nd edn (revised 1999)*, WytLytPub, Bellevue WA, USA.
- Halpin, T.A. & Vermeir, D. 1997, 'Default reasoning in information systems', *Database Applications Semantics*, Chapman & Hall, London, pp. 423-41.
- Martin, J. & Odell, J. 1998, *Object-Oriented Methods: a Foundation, UML edn*, Prentice Hall, Upper Saddle River, New Jersey.
- OMG-UML Specification v. 1.3 beta R6 draft, OMG UML Revision Task Force website, <http://uml.systemhouse.mci.com/artifacts.htm>.
- Rumbaugh, J., Jacobson, I. & Booch, G. 1999, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading MA, USA.
- Warmer, J. & Kleppe, A. 1999, *The Object Constraint Language: precise modeling with UML*, Addison-Wesley, Reading MA, USA.

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

UML data models from an ORM perspective: Part 10

by Dr. Terry Halpin

Director of Database Strategy, Visio Corporation

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

This paper is the tenth in a series of articles examining data modeling in the Unified Modeling Language (UML) from the perspective of Object Role Modeling (ORM). Part 1 discussed historical background, language design criteria, object reference and single-valued attributes. Part 2 covered multi-valued attributes, basic constraints, and instantiation using UML object diagrams or ORM fact tables. Part 3 compared UML associations and related multiplicity constraints with ORM relationship types and related uniqueness, mandatory role and frequency constraints, as well as how associations may be instantiated. Part 4 contrasted ORM nesting, co-referencing and exclusion constraints with UML association classes, qualified associations, and xor-constraints respectively. Part 5 discussed subset and equality constraints. Part 6 discussed subtyping. Part 7 discussed value, ring and join constraints. Part 8 listed some recent updates to the UML standard, then discussed aggregation. Part 9 examined initial values and derived data in ORM and UML. Part 10 discusses changeability and collection types in UML and ORM.

Changeability properties

In UML, restrictions may be placed on the *changeability* of attributes, as well as the roles (ends) of binary associations. It is unclear whether changeability may be applied to the ends of n-ary associations, but my guess is that this is currently forbidden. The following three values for changeability are recognized, only one of which can apply at a given time:

- changeable
- frozen
- addOnly

The value “changeable” was previously called “none”. Although the new term “changeable” was approved for UML 1.3 [0], some instances of “none” still occur in the standard; this oversight should be remedied in a later version. The default changeability is “changeable” (any change is permitted). Although the UML standard [0, p. 2-25] and

some authors [0, p. 166] indicate that “changeable” is a value, the standard also says “there is no symbol for whether an attribute is changeable”, so it appears that this default cannot be explicitly declared. However it makes sense to allow explicit declaration of this default, and it would not be surprising to see the standard revised to permit it. The other settings (frozen and addOnly) may be explicitly declared in braces. For an attribute, the braces are placed at the end of the attribute declaration. For an association, the braces are placed at the opposite end of the association from the object instance to which the constraint applies.

Recall that a “link” is an instance of an association. The term “frozen” means that once an attribute value or link has been inserted, it cannot be updated or deleted, and no additional values/links may be added to the attribute/association (for the constrained object instance). The term “addOnly” means that although the original value/link cannot be deleted or updated, others values/links may be added to the attribute/association (for the constrained object instance). Clearly, addOnly is only meaningful if the maximum multiplicity of the attribute/association-role exceeds its minimum multiplicity.

As a simple if unrealistic example, see Figure 1. Here empNr, birthDate and country of birth are frozen for Employee, so they cannot be changed from their original value. For instance, if we assign an employee the empNr 007, and enter his/her birthdate as 02/15/1946 and birth country as ‘Australia’, then we can never make any changes or additions to that.

Notice also that for a given employee, the set of languages and the set of countries visited are addOnly. Suppose that when facts about employee 007 are initially entered, we set his/her languages to {Latin, Japanese} and countries visited to {Japan}. So long as employee 007 is referenced in the database, these facts may never be deleted. However we may add to these (e.g. later we might add the facts that employee 007 speaks German and visited India).

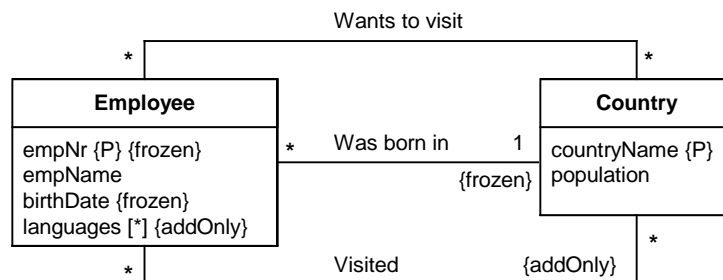


Figure 1 Changeability of attributes and association roles may be specified in UML

By default, the other properties are changeable. For example, employee 007 might change his name by deed poll from ‘Terry Hagar’ to ‘Hari Seldon’, and the set of countries he wants to visit might change, after some traveling, from {Ireland, Italy, USA} to {Greece, Ireland,}.

Some traditional data modeling approaches also note some restrictions on changeability. For example, Oracle's ER notation includes a diamond to mark a relationship as non-transferable (once an instance of an entity type plays a role with an object, it cannot ever play this role with another object). Although changeability restrictions may at first appear very useful, in practice their application in database settings is limited. One reason for this is that we almost always want to allow facts entered into a database to be changed. With snapshot data, this is the norm, but even with historical data, changes can occur. The most common occurrence of this is to allow for corrections of mistakes, which might be because we were told the wrong information originally or because we carelessly made a misspelling or typo when entering the data.

In exceptional cases, we might require that mistakes of a certain kind be retained in the database (e.g. for auditing purposes) but be corrected by entering later facts to compensate for the error. This kind of approach makes sense for bank transactions (see Figure 2). For example, if a deposit transaction for \$100 was mistakenly entered as \$1000, the record of this error is kept, but once the error is detected it can be compensated for by a bank withdrawal of \$900. As a minor point, the balance is both derived and stored, and its frozen status is typically implied by the frozen settings on the base attributes, together with a rule for deriving balance.

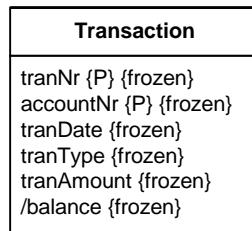


Figure 2 All attributes of Transaction are frozen

Although not stated in UML 1.3, some authors allow changeability to be specified for a class, as an abbreviation for declaring this for all its attributes and opposite association ends [0, p. 184]. For instance, all the {frozen} constraints in Figure 2 might be replaced by a single {frozen} constraint below the name "Transaction". While this notation is neater, it would be rarely used. Even in this example, we would probably want to allow for the possibility of adding non-frozen information later (e.g. a transaction might be audited by zero or more auditors).

Changeability settings may have more use in the design of program code than in conceptual modeling (e.g. {frozen} corresponds to const in C++). Although changeability settings are not supported in ORM, which focuses on static constraints, such features could easily be added as role properties if desired. In the wider picture, being able to completely model security issues (e.g. who has the authority to change what) would provide greater value. This view is nicely captured by the following comment of John Harris, in a recent thread on the InConcept website: "Rather than talk of "immutable" data I think it is better to talk of a privilege requirement. For instance, you can't change your recorded salary but your boss can, whether it's because you've had a pay rise or because

there's been a typing error. Privileges can be as complicated or as simple as they need to be, whereas "immutable" can only be on or off. Also, privileges can be applied to the insertion of new data and removal of old data, not just to updates”.

Collection types

Though collection types (e.g. sets, bags, sequences and arrays) are commonly used in programming, their use as record components in database schemas largely disappeared with the widespread acceptance of relational databases, where each table column is based on an atomic domain. However, the recent emergence of object-relational and object databases has once again allowed collection types to be embedded as database fields. Although a number of collection types were slated for inclusion in the object-relational database standard SQL3, the only one that made it was array (a one dimensional array with a maximum number of elements). It is anticipated that three further collection types will be added in SQL4: set (unordered collection with no duplicates); multiset (bag, i.e. unordered collection that allows duplicates); and list (sequence, i.e. an ordered bag). Some commercial systems already support these. Experience with these systems indicates that little performance gain is actually achieved by use of collection types; but this may change as the technology matures. Array, set, bag and list are also included as collection types in the object database standard ODMG 2.0 [0].

UML includes none of these as standard notations, but does include the {ordered} constraint to indicate mapping to an ordered set (i.e. a sequence with no duplicates); and its associated textual language OCL (Object Constraint Language) includes set, bag and sequence types as well as collection as their abstract supertype [0, pp. 38-49]. While UML allows collection types to be specified as stereotypes of classes, and realized as implementation classes [0, pp. 485-6], this usage seems geared toward code design so will not be elaborated here.

Different approaches have arisen as to how collection types should be specified within the conceptual analysis and logical design of data. Some proposals use collections directly within the conceptual schema, some introduce them only at the logical schema level, while some specify them as annotations to the conceptual schema to guide the mapping to the logical level. As a simple example, consider Figure 3. The ORM schema (a) and UML schema (b) depict driving as a many-to-many association. The employee name information is modeled as a functional fact type in ORM and as an attribute in UML. If this is mapped to a relational database system, then by default the m:n association maps to a separate table, resulting in a 2-table schema (c).

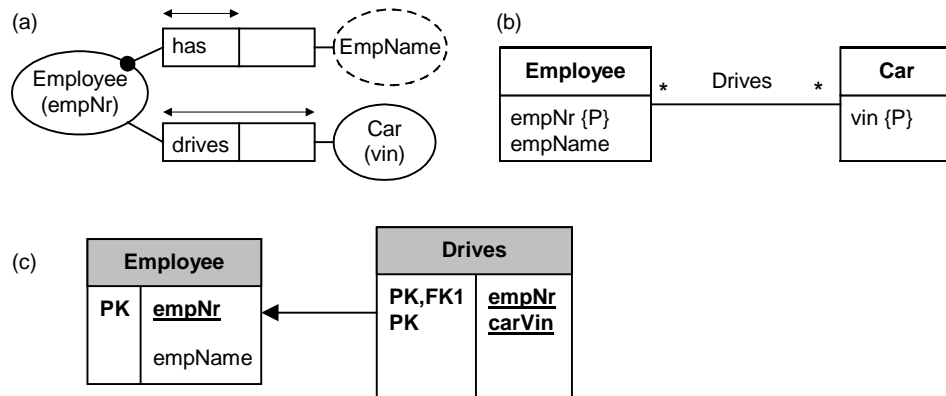


Figure 3 ORM schema (a) and UML schema (b) map by default to relational schema (c)

Now suppose that for some reason we wish to map both fact types into the same table, as shown in Figure 4(d). Some object-relational databases support this option. Clearly this mapping decision is an implementation, not a conceptual, issue, but how do we specify it? Visio Enterprise 5 lets you do this at the logical level (d), and VisioModeler lets you specify it either at the logical level or as an annotation to the ORM schema. The annotation shown in Figure 4(a) differs from that of VisioModeler (which uses a box between the role and its object type), but the idea is the same (indicating that this role maps to a set field of the co-role's table). The display of such annotations should be hidden during conceptual analysis, and toggled on only when we wish to discuss overrides to the default logical mapping. In UML we could invent a similar annotation, as in Figure 4(b), or instead use a multi-valued attribute, as in Figure 4(c), with this display being used only for discussing the logical mapping. As discussed in an earlier article, multi-valued attributes should never be used in conceptual analysis.

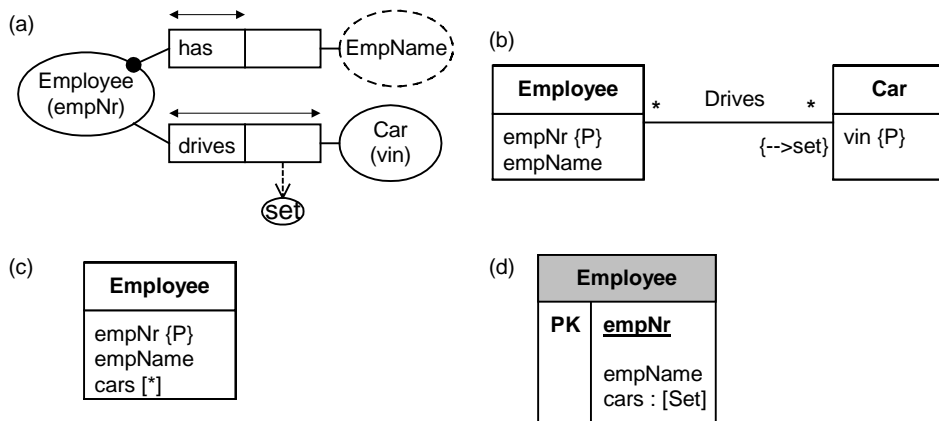


Figure 4 Some possible ways of indicating that driving should map to a set-valued column

Some extensions of ORM (e.g. PSM [0]) allow collection types (e.g. set, bag, sequence and schema) to be modeled as first class object types, using constructors often shown as a shape around the member object type. A sequence is an ordered bag, and in extended ORM its collection type may be marked “seq”. If the sequence cannot have duplicates, it is a “unique sequence” (or ordered set) and is marked “seq”. As an example of the unique sequence (or ordered set) constructor, see Figure 5(a). Here an author list is a sequence of authors, each of whom may appear at most once on the list. This may be modeled in flat ORM by introducing a Position object type to store the sequential position of any author on the list, as shown in Figure 5(b).

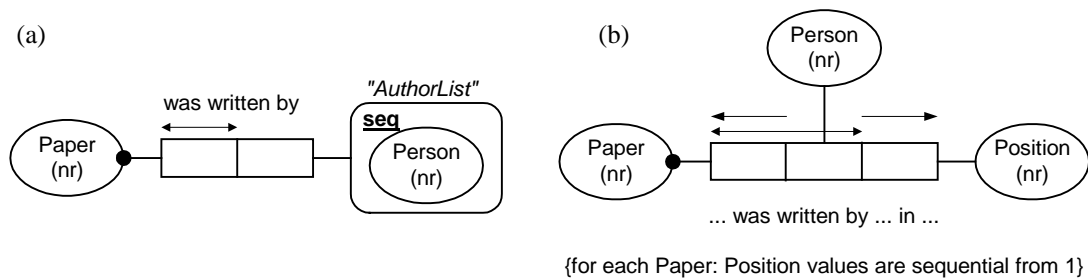


Figure 5 Unique sequence modeled in ORM with a constructor (a) or by introducing Position (b)

The uniqueness constraint on the first two roles declares that for each paper an author occupies at most one position; the constraint covering the first and third roles indicates that for any paper, each position is occupied by at most one author. The textual constraint below the graphic indicates that the positions in any list are numbered sequentially from 1. Although this ternary representation may appear awkward, it is easy to populate and it facilitates any discussion involving position (e.g. who is the second author for paper 21?). From an implementation perspective, a sequence structure could still be chosen: this can simplify updates by localizing their impact. However the update overhead of the positional structure is not onerous anyway, given set-at-a-time processing (e.g. to delete author n, simply set position to position-1 for position > n).

Though not shown here, the ternary solution can also be modeled in UML. If the ternary model is chosen as the base model, it would be useful to support the annotated binary shown in Figure 6(a) or Figure 6(b) as a view of the base model. In ORM we have shown a unique sequence annotation connected to the relevant role. This representation is equivalent to the {ordered} constraint in UML, as shown in Figure 6(b), indicating that the authors are to be stored as a unique sequence. The unique sequence annotation is not yet supported by Visio. UML does include “{ordered}” as a standard notation, but it does not yet include notations for other collections, although obvious ones suggest themselves (e.g. {sequence}).

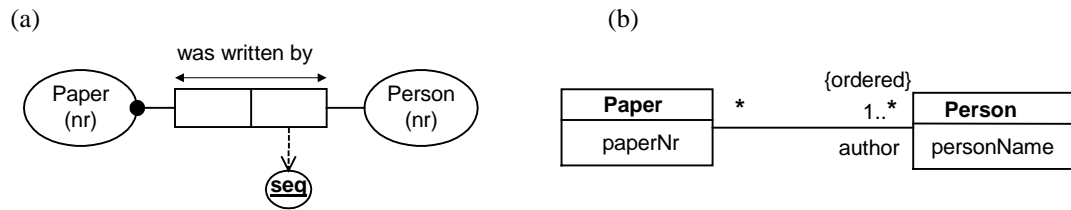


Figure 6 Unique sequence modeled with an annotation in ORM (a) and UML (b)

Flat models (no constructors) substantially simplify the declaration of constraints (which typically apply to members, not collections), derivation rules (and hence queries), and avoid arbitrary or non-conceptual decisions about how to store (and possibly duplicate) fact types and constraints. For example, in Figure 7 the ORM exclusion constraint may be verbalized: no Person wrote and reviewed the same Book. Although one could use collection types here (e.g. sets of books for an author, or sets of authors of a book) this would be extremely unwise, since it would complicate verbalization, validation, fact expression (possibly duplicated) and constraint expression (possibly duplicated). In conceptual modeling, we should not have to concern ourselves about how individual fact types might be stored in structures, or where the constraint code will reside. Such concerns are implementation details, and should be delayed until a clear conceptual picture of the world is obtained.

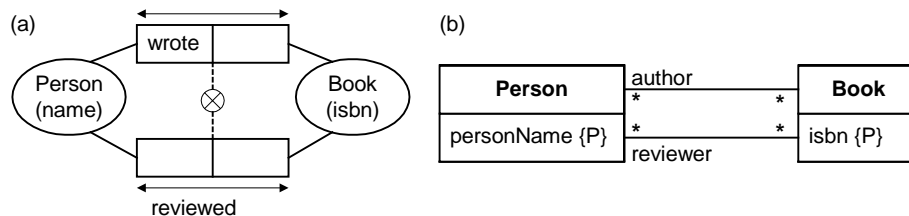


Figure 7 Pair-exclusion constraint in ORM (a) needs to be captured textually in UML

Since UML does not provide a graphical notation for such an exclusion constraint, it should be specified either informally as a note, or formally using a language of choice. Since OCL includes collection types with predefined operations, and the population of the association roles author and reviewer are sets, this constraint can be expressed in OCL as follows:

```

Book
self.author -> intersection(selfreviewer) -> isEmpty

```

Although this constraint expression is clear enough to somebody with a formal background, it is of little use for validating the rule with the subject matter expert

(typically a business person with little formal training). For such purposes, ORM's ConQuer language is far more suitable.

Next issue

The ten articles in this series have covered UML data modeling issues from an ORM perspective. My next couple of articles will consider other data modeling notations (flavors of ER, as well as IDEF1X) from an ORM viewpoint. Later on, I may return to UML to discuss its behavioral side.

References

- Booch, G., Rumbaugh, J. & Jacobson, I. 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading MA, USA.
- Cattell, R.G.G. (ed.) 1997, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publishers, San Francisco.
- Halpin, T. 1995, *Conceptual Schema and Relational Database Design, 2nd edn (revised 1999)*, WytLytPub, Bellevue WA, USA.
- ter Hofstede, A.H.M., Proper, H.A. & Weide, th.P. van der 1993, 'Formal definition of a conceptual language for the description and manipulation of information models', *Information Systems*, vol. 18, no. 7, pp. 489-523.
- Martin, J. & Odell, J. 1998, *Object-Oriented Methods: a Foundation, UML edn*, Prentice Hall, Upper Saddle River, New Jersey.
- OMG, *UML Specification v. 1.3 final draft*, OMG UML Revision Task Force website, <http://uml.systemhouse.mci.com/artifacts.htm>.
- OMG, *UML 1.3 Revisions and Recommendations*, Appendix A, issues 35-6, document ad/99-06-11, <http://uml.systemhouse.mci.com/artifacts.htm>.
- Rumbaugh, J., Jacobson, I. & Booch, G. 1999, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading MA, USA.
- Warmer, J. & Kleppe, A. 1999, *The Object Constraint Language: precise modeling with UML*, Addison-Wesley, Reading MA, USA.

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

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 WG8.1 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

<i>Data instances/structures</i>		<i>Constraints</i>	
<i>ORM</i>	<i>UML</i>	<i>ORM</i>	<i>UML</i>
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). *Whenever 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** Person₁ has a Title **that** is restricted to Sex₁ **then** Person₁ is of Sex₁). 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).

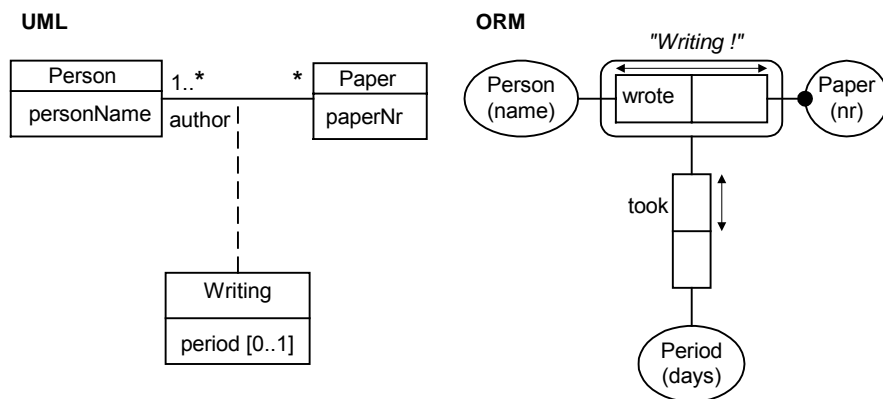


Figure 1: Writing is depicted as an objectified association in UML and ORM

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.

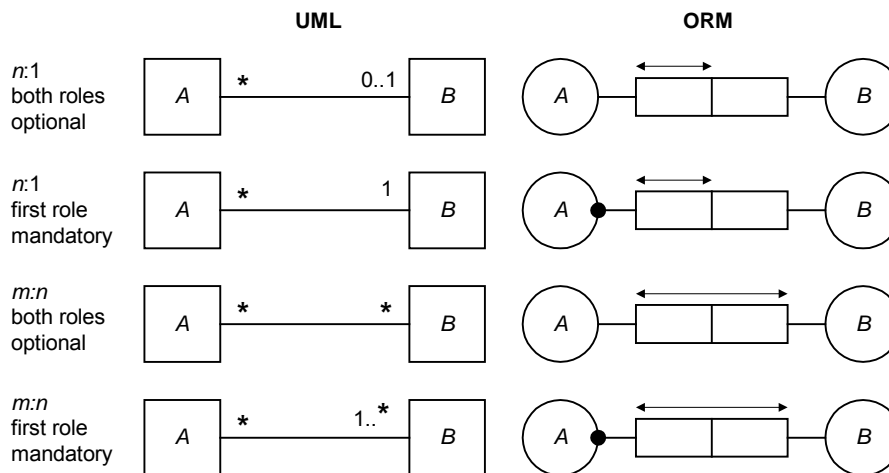


Figure 2: Some equivalent representations in UML and ORM

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.

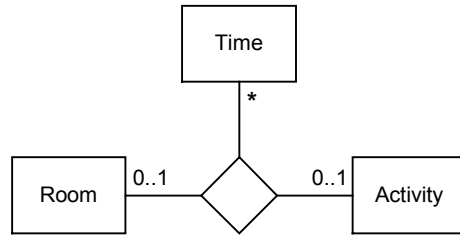


Figure 3: Multiplicity constraints on a ternary in UML

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.

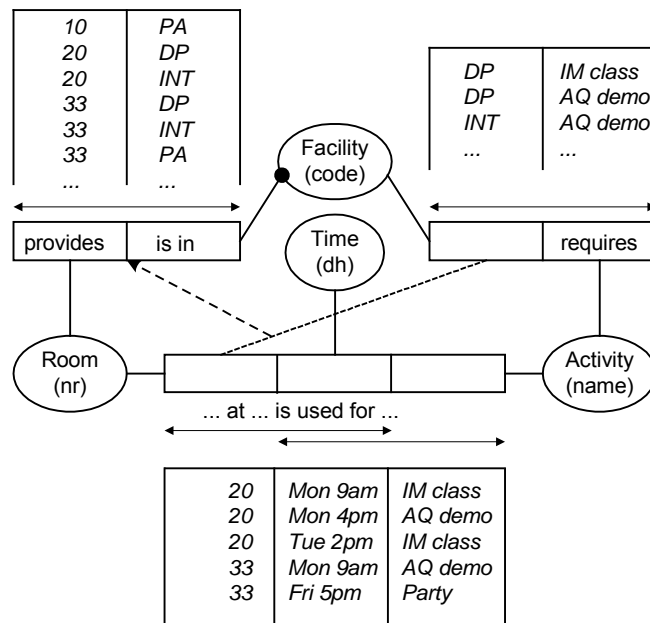


Figure 4: An ORM diagram with sample populations

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.

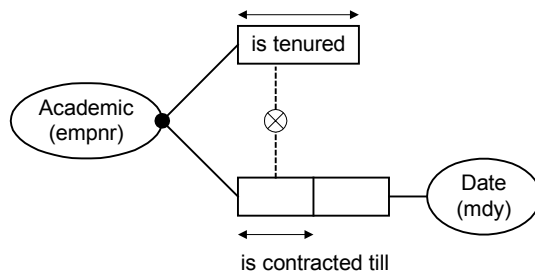


Figure 5: Disjunctive mandatory role constraint and exclusion constraint.

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).

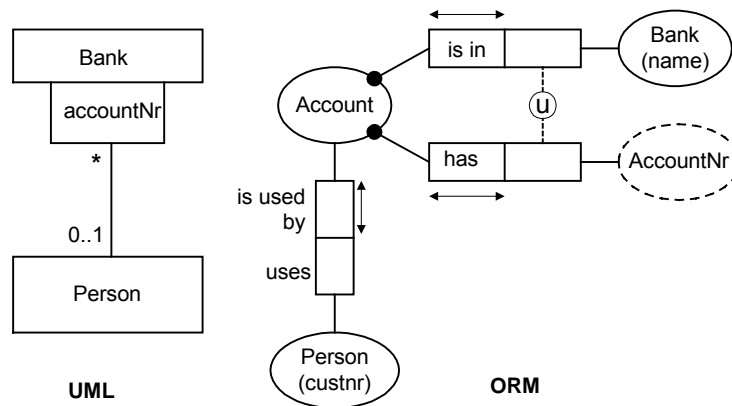


Figure 6: Qualified association in UML, and co-referenced object type in ORM

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 ActiveQuery².

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:Product; m:MoneyAmount | ∃ c: MoneyAmount; w: MoneyAmount •
    p has cost of c ∧ p has wholesale price of w ∧ m = w - c }
```

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

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

Or mathematically, as:

```
¬ ∃ p:Product • ∃ c: MoneyAmount; w: MoneyAmount •
    p has cost of c ∧ p has wholesale price of w ∧ 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.

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

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 WG8.1 International 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).

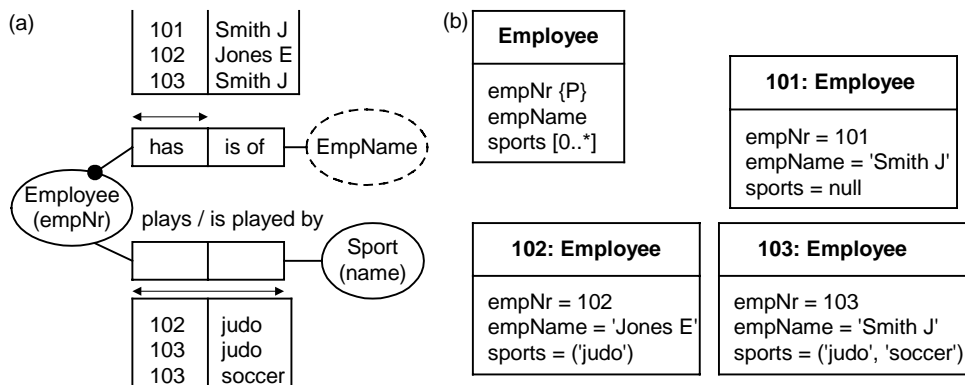


Figure 1: Employee plays Sport depicted as (a) ORM fact type and (b) UML multi-valued attribute.

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.

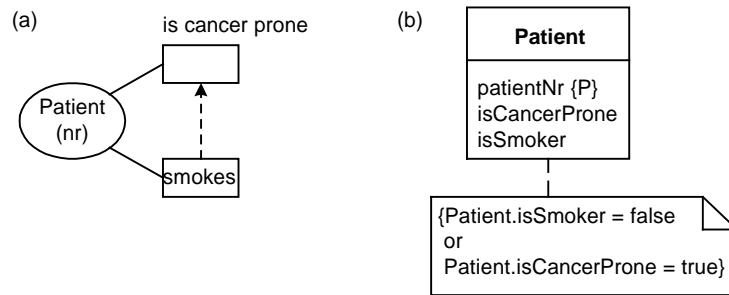


Figure 2: Patients who smoke are cancer prone, modeled in (a) ORM and (b) UML

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`.

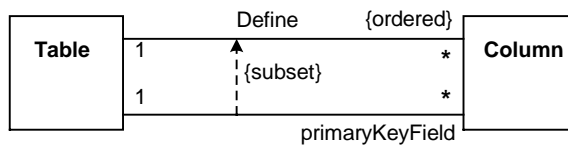


Figure 3: Spot anything wrong?

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).

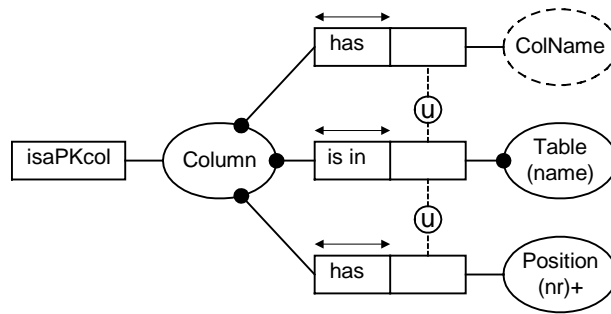


Figure 4: Alternative ORM model for schema shown in Figure 3

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.

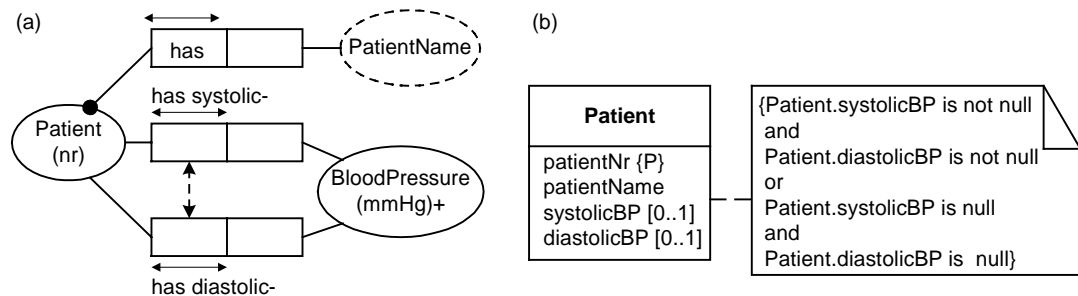


Figure 5: A simple equality constraint modeled in (a) ORM and (b) UML

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].

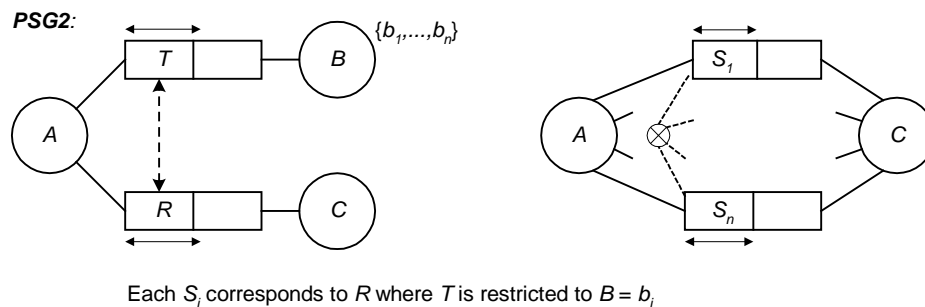


Figure 6: A basic schema equivalence in ORM

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.

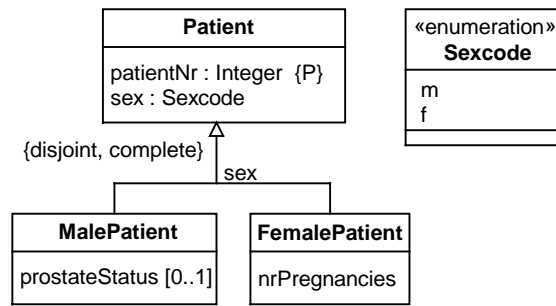


Figure 7: Subtyping in UML

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.

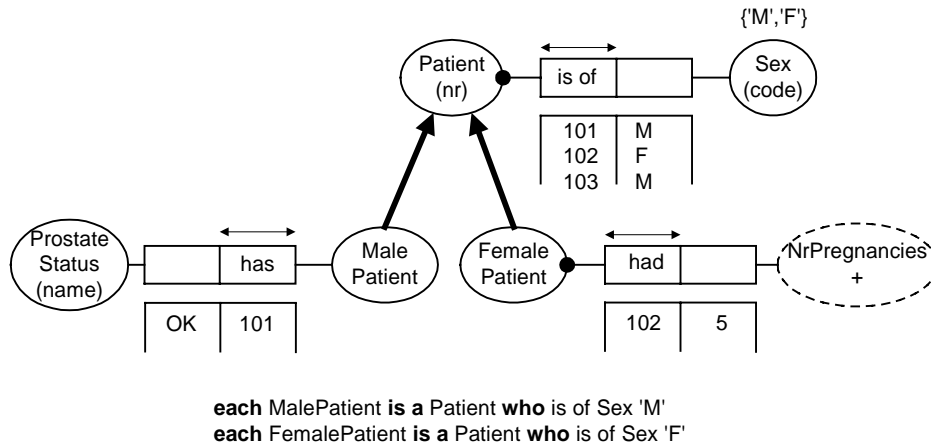


Figure 8: Formal subtype definitions are needed, and subtype partition constraints are implied

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.

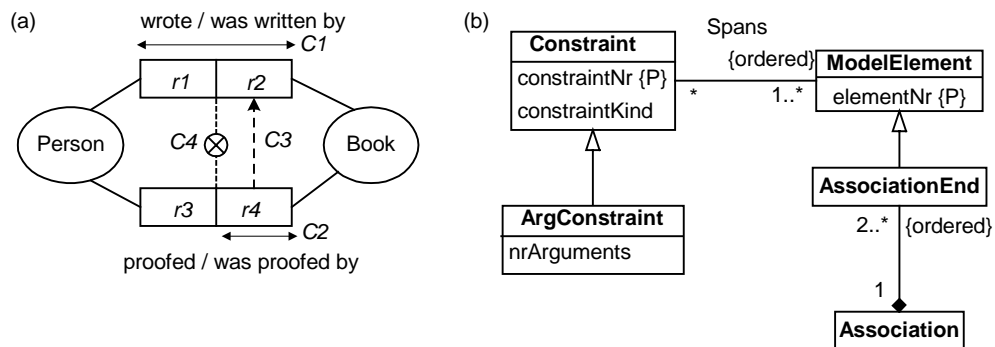


Figure 9: These ORM constraints (a) may be stored in an extended UML metamodel fragment (b)

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:

<i>constraintNr</i>	<i>constraintKind</i>	<i>nrArguments</i>	<i>argumentsSpanned</i>
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.

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

Augmenting UML with Fact-orientation

Terry Halpin

Microsoft Corporation, USA

TerryHa@microsoft.com

Abstract

The Unified Modeling Language (UML) is more useful for object-oriented code design than conceptual information analysis. Its process-centric use-cases provide an inadequate basis for specifying class diagrams, and its graphical language is incomplete, inconsistent and unnecessarily complex. For example, multiplicity constraints on n-ary associations are problematic, the constraint primitives are weak and unorthogonal, and the graphical language impedes verbalization and multiple instantiation for model validation. This paper shows how to compensate for these defects by augmenting UML with concepts and techniques from the Object Role Modeling (ORM) approach. It exploits "data use cases" to seed the data model, using verbalization of facts and rules with positive and negative examples to facilitate validation of business rules, and compares rule visualizations in UML and ORM. Three possible approaches are suggested: use ORM for conceptual analysis then map to UML; supplement UML with population diagrams and user-defined constraints; enhance the UML metamodel.

1. Introduction

The Unified Modeling Language (UML) was adopted in 1997 by the Object Management Group (OMG) as a language for object oriented (OO) analysis and design. This paper is concerned with UML version 1.3, the latest approved version at the time of writing. A minor revision (1.4) should be approved around December 2000, and a major revision (2.0) should be completed a few years later. Though not yet a standard, UML has been proposed for standardization by the International Standards Organization, with approval likely around 2001 [28].

The UML notation includes the following kinds of diagram for modeling different perspectives of an application: use case diagrams, class diagrams, object diagrams, statecharts, activity diagrams, sequence diagrams, collaboration diagrams, component diagrams and deployment diagrams. This paper focuses on conceptual data modeling, so considers only the static structure (class and object) diagrams. Class diagrams are used for the data model, and object diagrams for data

populations. Although not yet widely used for designing database applications, UML class diagrams effectively provide an extended Entity-Relationship (ER) notation that can be annotated with database constructs (e.g. key declarations). Background on UML may be found in its specification [31], a simple introduction [13] or a detailed treatment [6, 32]. In-depth discussions of UML for database design may be found in [30] and (with a slightly different notation) [3].

UML has become popular for designing OO program code. It is well suited for this purpose, covering data, behavior, and OO-implementation details (e.g. attribute visibility and directional navigation across associations). However, UML is less suitable for developing and validating a conceptual data model with domain experts. Its use-cases are process-centric, and in practice the move from use cases to class diagrams is often little more than a black art. Moreover, the UML notation prevents many common business rules from being diagrammed.

We believe these defects are best avoided by using fact-oriented modeling as a precursor to object-oriented modeling in UML. Object-Role Modeling (ORM) is the main exemplar of the fact-oriented approach, and is supported by CASE tools such as Microsoft Visio Enterprise [34]. For data modeling, ORM's graphical notation is more expressive and orthogonal than UML's, its models and queries are semantically stabler, and its design procedures fully exploit data examples using both verbalization and multiple instantiation to help capture and validate business rules with domain experts.

This paper identifies several weaknesses in the UML graphical language and discusses how fact-orientation can augment the object-oriented approach of UML. It shows how verbalization of facts and rules, with positive and negative examples, facilitates validation of business rules, and compares rule visualizations in UML and ORM on the basis of specified modeling language criteria. The following three approaches are suggested as possible ways to exploit the benefits of fact-orientation: (1) use ORM for conceptual information analysis and map the ORM model to UML; (2) use UML in its current form, supplemented by informal population diagrams and user-defined constraints; (3) correct and extend the UML metamodel to better support business rules.

The rest of this paper is structured as follows. Section 2 provides a brief comparative overview of UML and ORM, based on linguistic design criteria. Section 3 discusses verbalization issues related to multiplicity constraints on binary associations. Section 4 illustrates how “data use cases” help guide the data modeling process as a joint activity between modeler and domain expert. It also exposes problems with UML multiplicity constraints on n-ary associations, and highlights the need for a richer graphical constraint notation. Section 5 summarizes how the lessons learned from fact-orientation can be used to augment UML, identifies areas of future research, and lists references for further reading.

2. ORM, UML and language criteria

Object-Role Modeling is a conceptual modeling method that views the world as a set of objects (entities or values) that play roles (parts in relationships). For example, you are now playing the role of breathing (a unary relationship involving just you), and also the role of reading this paper (a binary relationship between you and this paper). An entity in ORM corresponds to a UML object, and a value to a UML data value. A role in ORM corresponds to an association-end in UML, except that ORM also allows unaries. The main structural difference between ORM and UML is that ORM excludes attributes as a base construct, treating them instead as a derived concept. For example, `Person.birthdate` is modeled in ORM as the fact type: `Person was born on Date`. Overviews of ORM may be found in [15, 16] and a detailed treatment in [14]. The ORM notation uses only a handful of symbols, readily mastered by UML modelers. Although various ORM-based proposals for process/behavioral modeling exist [e.g. 24], they are ignored here.

The ORM language was designed from the ground up to meet the following criteria: expressibility; clarity; learnability (hence orthogonality, parsimony and convenience); semantic stability (minimize the impact of change); semantic relevance (scope views to just the currently relevant task); validation mechanisms; abstraction mechanisms; and formal foundation. Background on these principles may be found in [1, 4, 25, 26]. Practical trade-offs between design criteria can arise, e.g. expressibility-tractability [29] and parsimony-convenience [18]. In this paper our focus is on validation mechanisms, expressibility and orthogonality.

The most debatable feature of ORM is its avoidance of attributes in the base model. This omission was originally made to avoid fuzzy and unstable distinctions about whether a feature should be modeled as an attribute or association [12]. Although this advantage is enjoyed by some other semantic modeling approaches, such as OSM [10], a disadvantage is that attribute-free diagrams often

take up more space. A detailed argument that this price is worth paying can be found in [19]. The main advantages are that all facts and rules can be easily verbalized as sentences, all data structures can be easily populated with multiple instances, the metamodel is simplified, and models and queries are stabler since they are immune to changes that reshape attributes as associations. Finally the compactness of attribute-based models can still be achieved by deriving them as views (this is automatable).

Table 1 summarizes the main correspondences between conceptual data constructs in ORM and UML. Some examples are given later, and complementary discussions can be found in the references [14, 18, 19, 20, 21]. An uncommented “—” indicates no predefined support for the corresponding concept, and “†” indicates incomplete support. Clearly, ORM’s built-in symbols provide greater expressibility for conceptual constraints on data.

Table 1 Conceptual data constructs in ORM and UML

<i>ORM</i>	<i>UML</i>
Data structures: object type: entity type; value type — { use association } <i>unary association</i> 2 ⁺ -ary association objectified association co-reference	Data structures: object class data type <i>attribute</i> — { use Boolean attribute } 2 ⁺ -ary association association class qualified association †
Predefined Constraints: internal uniqueness external uniqueness simple mandatory role disjunctive mandatory role frequency: internal; external value subset and equality exclusion subtype link and definition ring constraints join constraints object cardinality —{use unique and ring} †	Predefined Constraints: multiplicity of ..1 * — {use qualified assoc. } † multiplicity of 1 ⁺ .. † — multiplicity †; — enumeration, and textual subset † xor † subclass discriminator etc. † — — class multiplicity aggregation/composition
Textual constraints	Textual constraints

Because of its orthogonality and avoidance of attributes, ORM achieves this greater expressibility without adding complexity. For example, ORM includes a disjunctive mandatory role (inclusive-or) constraint to constrain instances of an object type to play at least one of a set of roles (e.g. each Applicant must have a Qualification or a JobReference or both). ORM also includes an exclusion constraint that may apply between compatible role sequences (e.g. no Person who writes a Paper may referee that Paper). In ORM an exclusion

constraint between single roles may be orthogonally combined with an inclusive-or constraint to form an exclusive-or constraint (e.g. no Person may get a BusPass and a ParkingPermit). In contrast, UML supports an exclusive-or constraint as a primitive, but no inclusive-or and no general exclusion constraint.

Unlike UML, ORM allows constraints to be applied wherever they makes sense. For example, subset constraints may apply between compatible role sequences, not just associations (e.g. if a Person drives a Car then that Person has a DriverLicence). Ring constraints are logical constraints on ring associations (e.g. “no Person reports to himself/herself” is an irreflexive ring constraint). Join constraints apply to roles from connected predicates, e.g. each Employee who works in a Country also speaks a Language that is spoken in that Country).

Although the additional constraints in ORM often arise in practice, UML models often omit them unless the modeler is very experienced. Both UML and ORM allow the user to add constraints and derivation rules in a textual language of their choice. UML suggests OCL (Object Constraint Language) [33] for this purpose, but does not mandate its use. ORM’s conceptual query language, ConQuer [4, 5, 21], provides a formal but higher level alternative to OCL. Although textual languages are needed for completeness, it is easier for a modeler to think of a rule if it is part of his/her graphical rule language.

3. Binary associations

Since the domain expert is the person who understands the universe of discourse (UoD) or application domain, it is critical to promote good communication between the modeler and the domain expert in the conceptual analysis phase. Subject matter experts are often not technically skilled in modeling notations, so any business rules should be verbalized in their natural language for model validation. This section discusses verbalization of binary associations and their associated multiplicity constraints.

Consider a UoD in which employees must occupy a room, possibly shared with another employee, and some rooms may be unoccupied. For a given state of the database, the *population* of a type is the set of instances of that type that are present in the database. For this UoD, each population of the occupancy association is a total function (mandatory n:1 relation) from the population of Employee to the population of Room. A significant sample population is included in the instance diagram at the top of Figure 1.

Figure 1(a) depicts this binary association in UML. Classes are denoted by named rectangles, and *binary associations* by connecting lines. The association ends correspond to roles in ORM, and may be given a role name (e.g. “office”). The association itself may be given a

name (e.g. “Occupies”) as well as a marker “▶” to indicate the direction in which the association should be read. So long as an association name is supplied, the association can be verbalized as a sentence type (e.g. Employee occupies Room).

The association roles (ends) may be adorned with *multiplicity constraints* that specify the possible multiplicities. For example, “1..*” means one or more (at least one), “0..1” means zero or one (at most one), “1” abbreviates “1..1” (exactly one) and “*” abbreviates “0..*” (zero or more). Like ORM, UML allows multiplicities to include combinations of numbers and number ranges (e.g. “2, 4, 6, 10..20”), even if these would be rarely used.

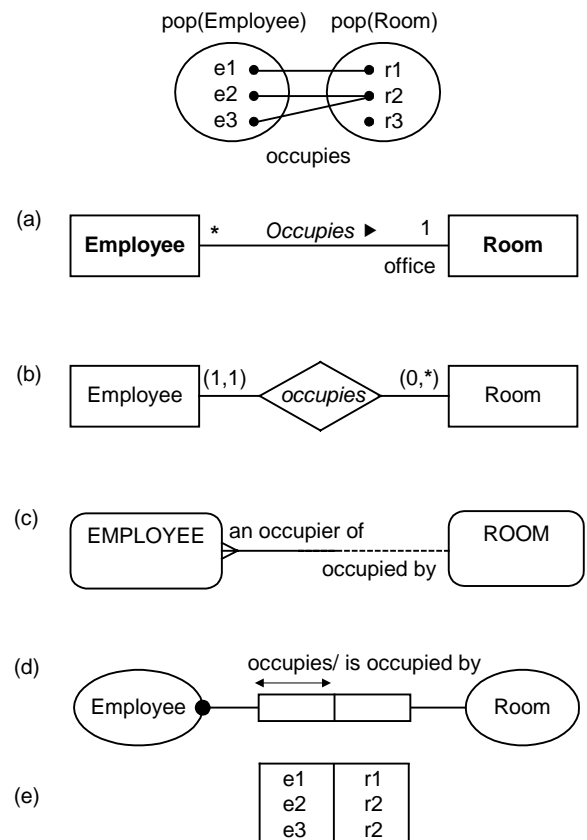


Figure 1 Mandatory n:1 association in (a) UML (b) DSB-ER (c) Barker-ER (d) ORM

UML places each multiplicity constraint on the “*far role*”, in the direction in which the association is read. Hence the multiplicity constraint on the Room role may be *verbalized* thus: **each** Employee occupies **exactly one** Room. The “*” constraint on the Employee role may be verbalized: **it is possible that more than one** Employee occupies **the same** Room. The “*” (zero or more) is the default multiplicity for a role, and may be regarded as the absence of a constraint rather than a constraint. Hence we could omit its

verbalization, but it is normally safer to provide it to clarify its impact.

These verbalizations, which we developed for use in ORM, rely on singular terms being used for class names (e.g. “Employee” not “Employees”) for natural phrasing. Words shown in bold type have formal meaning, allowing an ORM tool to automatically generate an ORM diagram from the textual formulation of the association and its constraints. Although UML does not have any formal verbalization, a request for proposal has been issued by the UML committee for a “Human Readable Textual Notation”, so something like this could eventually be added to UML. ORM’s verbalization patterns could provide a good basis for extending UML in this way.

Figure 1(b) shows the same association in an ER notation recently proposed by Dey, Storey and Barron for work with binary and n-ary ($n > 2$) associations [9]. Let’s call this DSB-ER notation after its proponents. Here, entity types are depicted as named rectangles and binary relationships are depicted as named diamonds, as in Chen’s original ER [8]. The constraints are called *participation constraints*. The association and its constraints may be verbalized as before. As with some other versions of ER, this notation places the constraint on the “near role”, to indicate the minimum and maximum number of times each instance of the role player must participate in that role. Hence the “(1, 1)” and “(0,*)” on the left and right roles correspond to UML’s “1” and “*” placed on the right and left roles respectively (the opposite).

Figure 1(c) shows the same example in the Barker-ER notation popularized by Richard Barker [1] and Oracle Corporation. Unlike UML and DSB-ER, but like ORM, the Barker notation supports *forward and inverse readings* of binary relationships. This is useful practice facilitates navigation in different directions around a schema, and often leads to improved verbalization of rules. Some UML users have added their own notations in this regard, such as appending reverse readings in parentheses to the association name [11]. However the UML specification has no formal support for this. We recommend that UML be extended by adding a slot in its metamodel to store reverse readings, and provide a standard syntax for their display.

Unlike the two previous notations, Barker-ER uses *separate notations for minimum and maximum cardinalities*. Minimum cardinalities of 0 (optional) or at least 1 (mandatory) are specified as optional and mandatory roles. A role that is *optional* for its entity type is designated by a dashed line-half, and a role that is *mandatory* is depicted by a solid line-half: these are specified on the near role. A maximum cardinality of 1 is the default (no explicit mark), and a maximum cardinality of many is depicted as a crow’s-foot: these are shown on the far role as in UML.

Barker [1] suggests a relationship naming scheme that, while awkward for verbalizing relationship types or instances, does allow a structured means of verbalizing the cardinality constraints. Let $A R B$ denote an infix relationship R from entity type A to entity type B . Name R in such a way that each of the following four patterns results in an English sentence: **each A (must | may) be R (one and only one B | one or more B-plural-form)**. Use “must” or “may” when the first role is mandatory or optional respectively. Use “one and only one” or “one or more” when the cardinality on the second role is one or many respectively. For example, the constraints in Figure 1(c) verbalize as: **each Employee must be an occupier of one and only one Room; each Room may be occupied by one or more Employees**. This verbalization convention is good for basic multiplicity constraints on infix binaries. However it is less general than ORM’s approach, which applies to instances as well as types, for predicates of any arity, with no need for pluralization.

Figure 1(d) shows the same association in ORM. Entity types are depicted as named, solid ellipses, and relationships as named sequences of one or more roles, with each role depicted as a box connected by a line to its object type. A relationship is called a *fact type* unless it is used simply to provide a primary reference scheme. For binary associations, forward and inverse readings may be provided, separated by a slash. As in UML, each role may also be named, although ORM tools typically store role names on property sheets rather than display them on the diagram.

A black dot “•” on a role connector indicates the role is *mandatory* (must be played by each instance in the population of the object type). By default, a role is optional (no black dot). ORM constraints were designed to facilitate validation using sample populations. An arrow-tipped bar over one or more roles is a *uniqueness constraint* declaring that each entry in the population of that role sequence is unique (occurs there exactly once). Any relationship may be populated with a table where each column corresponds to the role in that position. So the constraint over the left role of Figure 1(d) indicates that entries in the left column of Figure 1(e) must be unique, unlike the right column. If the association were instead many-to-many, the constraint would span both roles and only the entry-pairs making up the table rows must be unique.

Of the four notations, only UML depicts a mandatory role by a minimum multiplicity > 0 on the far role. As we’ll see in the next section, this leads to problems for n-ary associations. As it turns out, of all the notations discussed, only the ORM notation generalizes properly for n-ary associations.

4. Data use cases and n-ary associations

Use cases in UML illustrate ways in which the required information system may be used, so they are useful in requirements analysis. However because they focus on *behavioral* modeling, they can only go so far in helping the modeler arrive at a *data* model. They should be supplemented by examples of information that the system is expected to manage. In ORM these examples have traditionally been referred to as “information samples familiar to the domain expert”. By analogy with the UML term, we call them *data use cases*. They can be output reports or input screens, and since they exist at the external level they can present information in many different ways (e.g. tables, forms, graphs, diagrams).

Whatever the appearance of a data use case, a subject matter expert should be able to verbalize its information in The modeler then transforms that informal verbalization into a formal yet natural verbalization that is clearly understood by the domain expert. These two verbalizations, one by the domain expert transformed into one by the modeler, comprise step 1 of ORM’s conceptual analysis procedure. Here we use verbalization of populations to arrive at the fact instances that are then abstracted to fact types. Constraints and derivation rules are meta-facts (facts about the object facts), which are then added and themselves validated by verbalization and population. This approach is very effective in practice, and we believe it is an ideal precursor to the specification of the data model in UML or any other language.

Suppose that our system is required to output reports like that shown in Figure 2. We ask the domain expert to read off the information contained in the tables and then rephrase this in formal English. For example, the subject matter expert might read off the facts on the top row of the first table as follows: Archery is new (it’s the first year it’s been included in the rankings); the US ranks first in archery, and scored 10 points for that. As modelers, we note that Rank functionally determines Points in the population, so ask: Does the Rank (e.g. 1) determine the Score (e.g. 10)? The domain expert replies in the affirmative (if he/she gets this wrong, ORM’s arity-check can detect it later [14]).

We now rephrase the information into elementary sentences: the Sport named ‘Archery’ is new; the Country coded ‘US’ has the Rank numbered 1 in the Sport named ‘Archery’; the Rank numbered 1 earns the Score 10 points. Similarly, the top row of the second table may be verbalized as: the Country coded ‘AD’ has the CountryName ‘Andorra’. If reference schemes are agreed to up front, these long-winded verbalizations can be abbreviated. Once the domain expert agrees with the verbalization, we proceed to abstract from the fact instances to the fact types.

(a) * new

Sport	Rank	Country	Points
Archery *	1	US	10
Baseball	1	US	10
	2	JP	5
Cricket	1	AU	10
	1	GB	10
...

(b)

Country	
Code	Name
AD	Andorra
AE	United Arab Emirates
...	...
ZW	Zimbabwe

Figure 2 Two sample output reports for a data use case

We may now draw the conceptual schema and populate it with sample facts. For discussion purposes, we consider the ORM solution (Figure 3) before the UML solution. Simple reference schemes may be abbreviated in parenthesis (e.g. “Country(code)” abbreviates the injective association Country has Countrycode). Value types need no reference scheme, and appear as named, dashed ellipses (e.g. CountryName). Here we have one unary fact type, Sport is new, two binary associations Country has CountryName, Ranks earns Score, and one ternary association Country has Rank in Sport.

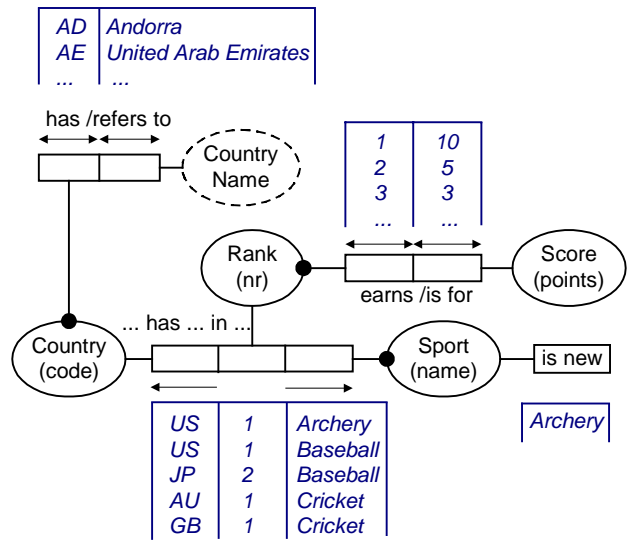


Figure 3 ORM schema for Figure 2, with sample data

Unlike other approaches, ORM allows mixfix predicates, which are sentences with object holes (denoted by "...") that may appear anywhere in the sentence. In this example, the ternary predicate is "... has ... in ...". This allows verbalization of sentences of any arity in any natural language, along with their associated constraints and derivation rules. Other approaches use a simple name for the verb phrase or assume binary infix predicates, that support only SVO (Subject Verb Object) languages, not SOV languages (e.g. Japanese) or VSO languages (e.g. Tongan). In principle, mixfix predicates could be used in UML, by extending its metamodel with positional information to provide a role order for predicate readings.

For each fact type in Figure 3, a sample fact table has been added to help validate the constraints. ORM schemas can be represented in diagrammatic or textual form, and tools such as Visio Enterprise can automatically transform between the two representations. Models are *validated* with domain experts in two ways: *verbalization*; and *population*. For example, the uniqueness constraints on the rank association in Figure 3 verbalize as: **each Rank earns exactly one Score**; **each Score refers to at most one Rank**. The 1:1 nature of this association is illustrated by the population, where each column is unique. A sample row for rank 3 has been added to illustrate the mandatory and optional nature of the roles played by Rank (a rank's score must be recorded even if no country achieves this rank).

The uniqueness constraint on the first and last roles of the ternary has a *positive verbalization* of: **each Country has at most one Rank in each Sport**. This is illustrated by the population, where the Country-Sport value pairs are unique. To double check a constraint in ORM, a *negative verbalization* of the constraint may be given, as well as a *counter-example* to test whether the constraint may be violated. For example, the uniqueness constraint on the ternary may also be verbalized thus: **it is impossible that the same Country has more than one Rank in the same Sport**. Adding the counter-row (US, 2, Archery) to the sample population of the ternary gives the US two ranks in archery, and hence violates the uniqueness constraint. Concrete examples like this make it easier for domain experts to see whether the constraint being tested really is a rule.

Because all fact types are elementary, and no attributes are used, populations never contain null values. Although closed or open world semantics may be chosen, the default semantics is closed world. For example Baseball appears in the population of Sport but does not play the role "is new", so we know it is not new. This is less confusing to the domain expert than assigning False to a boolean attribute, as in UML. For this reason, and to support natural verbalization, we suggest that UML be extended to allow unaries. A trivial change to the metamodel would allow this (change the multiplicity on Association-end from "2..*" to "1..*"). However,

pragmatism may require an inelegant alternative that is easier for vendors to support.

Unlike other approaches, ORM allows n readings for any n -ary predicate ($n > 0$), one starting at each role. This facilitates constraint declaration, and navigation through the information model from any starting position using natural sentences [4, 5]. In principle, the UML metamodel could be extended to support this.

Figure 4 shows a UML schema for the same UoD. All the ORM binary fact types are modeled here as attributes. In the absence of a standard UML syntax for primary identification or uniqueness constraints on attributes, we use our own notations "{P}" and "{U1}" respectively. Such notations are needed if UML is to be used to completely model even simple database applications.

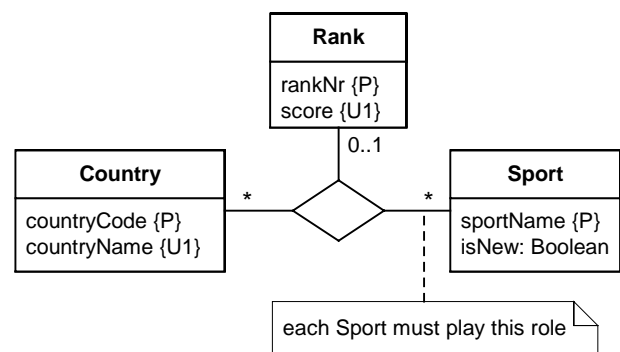


Figure 4 UML schema for Figure 2

The uniqueness constraint from the ORM ternary is modeled in UML using the 0..1 multiplicity constraint on the role played by Rank. The "*" multiplicities indicate the absence of any other uniqueness constraint. If an n -ary fact type is elementary, any internal uniqueness constraint must span either $n-1$ or n roles. The UML notation for multiplicity constraints can express these cases, but cannot express uniqueness or frequency constraints on fewer than $n-1$ roles. Hence unlike ORM it cannot be used to specify compound fact types that may be required for derivation or denormalization. The DSB-ER notation was developed to cater for cardinalities on n -aries, but is even worse than UML in this regard since it cannot express composite uniqueness and frequency constraints. The Barker-ER notation has the same problem if extended to n -aries.

Note that the *simple mandatory role constraint on Sport cannot be expressed by a multiplicity constraint in UML*. It might be thought that this constraint can be expressed by changing the multiplicity on the Country role to 1..*. But this would mean that each Sport-Rank pair formed from the populations of Sport and Rank must be associated with at least one country. But this is not true, since the role played by Rank is optional. For example, the pairs Archery-2 and Archery-3 have no associated country in the sample population. As discussed later, any

attempt to redefine the semantics of multiplicity constraints in terms other than the populations of its object types leads to other problems.

This exposes a fundamental problem with the scalability of UML's multiplicity notation. Although it caters adequately for binaries, it cannot express a simple mandatory constraint on at least 1 and at most $n-2$ roles within an n -ary association. If we are to use an n -ary in UML, the only thing we can do in such cases is to add a textual description of the constraint in a note, as in Figure 4. This problem is a direct consequence of choosing to attach minimum multiplicity to a far role instead of the near role. The DSB-ER and ORM notations can express mandatory constraints on roles of n -aries, and the Barker-ER notation could be extended to do so, since each attaches minimum multiplicity on the near role.

Sometimes, we can overcome this problem with UML by binarizing the n -ary. For example, Figure 5 expresses the fact type Country is ranked in Sport as a binary association, that is objectified as the class Ranking. The mandatory role for Sport is now catered for by the 1..* constraint on the role for Country. However, this approach has problems. To begin with, it is often too unnatural. If the domain expert thinks in terms of a ternary, why force him/her to rethink the model in terms of binaries? More importantly, this solution does not always work in UML. For example, suppose we have the additional constraint that no ties are allowed for sport ranks. There is no symbol in UML to express this rule on the binarized solution (although it can be expressed on the ternary).

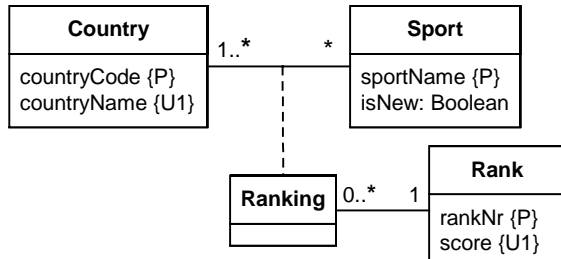


Figure 5 Alternative UML schema for Figure 2

The only way to express the no-ties rule with the binarized model would be to extend UML with the additional notion of an external multiplicity constraint that can span model elements from different associations. ORM already includes such a constraint. For example, Figure 6 shows the binarized solution in ORM with an external uniqueness constraint (circled “u”) to indicate that each Sport-Rank pair is associated with at most one Country in the overall association. ORM shows an objectified association by enclosing the association in an envelope. Although this works, it is more awkward to think about than the ternary solution for this case.

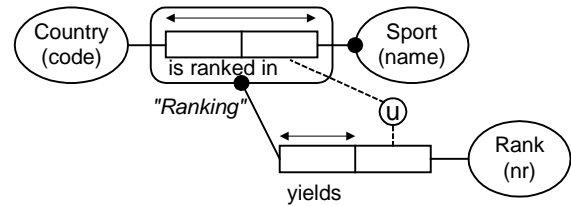


Figure 6 A nested ORM model that forbids ties

The next example illustrates the no-ties rule on the ternary, as well as another defect of multiplicities in UML. Consider the report shown in Table 2. In this UoD, no ties are allowed, and we are interested only in the first two ranks. Moreover, we may list a sport before any other details (e.g. ranking) are known for it. If a sport is ranked, we must know both its first and second place getters.

Table 2 A data use case for a somewhat different UoD

Sport	Rank	Country	Points
Aikido	?	?	?
Archery	1	US	10
	2	GB	5
Baseball	1	US	10
	2	JP	5
Basketball	?	?	?
Cricket	1	AU	10
	2	GB	5
...

An ORM schema for this situation is shown in Figure 7, together with a sample population for the ternary association and for the object type Sport. Here the “!” on Sport indicates it is an independent object type (instances of it can exist without playing any fact role). A meta-rule in ORM implies that any population object must play in some fact unless it is declared independent. There is no space here to extol the virtues of this rule, but its practical utility is such that we believe it should be added to UML.

Notice the uniqueness constraint over the roles played by Rank and Sport in the ternary. This enforces the no-ties rule. In ORM the positive verbalization of this constraint is: **given any Rank and Sport, at most one Country has that Rank in that Sport.** This is supported by the sample population. The negative verbalization of the constraint is: **it is impossible that more than one Country has the same Rank in the same Sport.** The negative verbalization is especially useful in using *counter-examples* to check the constraint. For instance, if we gave both Australia and Great Britain the rank 1 in cricket (as in Figure 3), this would violate the constraint. Such concrete counter-examples make it easy for domain experts to validate doubtful constraints.

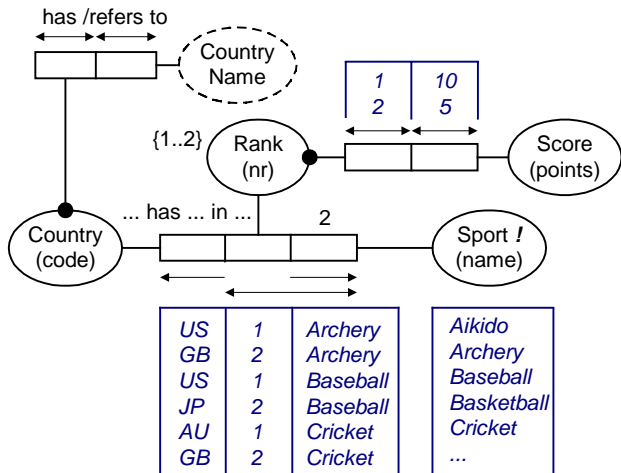


Figure 7 ORM schema for Table 2 with sample data

The frequency constraint of 2 on the Sport role means any sport that plays that role does so exactly twice. In the context of the uniqueness constraints and the value constraint of {1..2} on Rank, this ensures that both ranks are recorded for any ranked sport. Again, the population clarifies the constraint. Notice that some sports (e.g. Aikido) have not yet been ranked. Figure 8 shows the UML solution. There is no way of specifying the frequency constraint via a multiplicity constraint, so it has been added informally in a note.

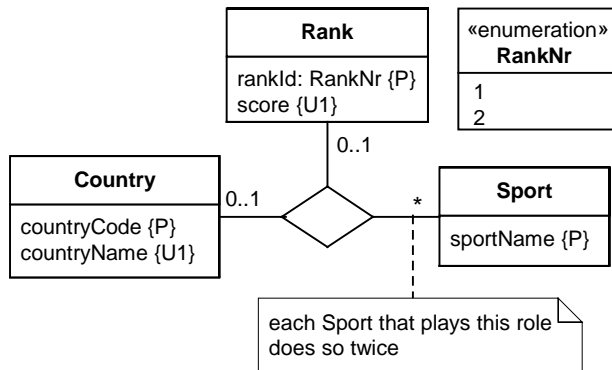


Figure 8 UML schema for Table 2

The two uniqueness constraints are expressed using 0..1 multiplicity constraints. This is possible because uniqueness is a case of maximum multiplicity. The frequency constraint of 2 cannot be expressed on a ternary in UML because it involves a minimum occurrence frequency of 2. An occurrence frequency of n means: if an instance plays the role, it does so n times. *Given any n -ary association, UML multiplicity constraints cannot express a minimum occurrence frequency above 1 for any role (or combination of fewer than $n-1$ roles).*

ORM allows mandatory and frequency constraints over a set of roles (possibly from different associations). Uniqueness constraints are just frequency constraints of 1 with a special notation because of their importance and ubiquity. These constraints are orthogonal, and apply to associations of any arity. UML's multiplicity constraints can express simple mandatory and frequency constraints for binary associations, but cannot express mandatory role constraints or minimum occurrence frequencies above 1 for roles in n -ary associations. So UML's multiplicity notation is far weaker than expected.

In fact, the whole notion of a *minimum multiplicity above 0 is problematic for n -aries in UML*. The UML 1.3 specification offers only the following description for the semantics of multiplicities in n -ary associations: "The multiplicity of a role represents the potential number of instance tuples in the association when the other $n-1$ values are fixed" [31, p. 3-73]. Consider a ternary association $R(A, B, C)$. Let $\text{pop}(A)$, $\text{pop}(B)$ and $\text{pop}(C)$ be the populations of A , B and C (in the database, not necessarily just in R), and $\text{pop}(rA)$, $\text{pop}(rB)$ and $\text{pop}(rC)$ be the populations of the roles in R . Let R have multiplicities $*, *, 2..*$ on the roles of A, B, C respectively. What does the 2 mean? For consistency with the meaning of multiplicities for binary associations, we should define it thus: each pair (a, b) , where a is in $\text{pop}(A)$ and b is in $\text{pop}(B)$, is associated in R with at least 2 instances from $\text{pop}(C)$. But such a constraint is in practice virtually useless, since it far too strong to apply except in pathological cases. To base the constraint on the types rather than populations would be even worse in this regard.

What we really need is a way to define the constraint in terms of R 's population. For example, each pair (a, b) that occurs in the projection $\text{pop}(R)[a, b]$ is associated in R with at least 2 instances from $\text{pop}(C)$. This corresponds to an ORM minimum frequency constraint of 2 on (rA, rB) . Although useful and desirable, this definition is inconsistent with the whole approach to multiplicity constraints in UML. For example, if accepted it would mean that minimum multiplicities of 0 could never occur.

Internal frequency (and uniqueness) constraints in ORM can be efficiently implemented and validated because they apply just to the local population of their predicate. Mandatory constraints refer to the population of an object type, so are ontologically distinct as well as harder to enforce. Because of their global impact, mandatory constraints need to be considered more carefully. For such reasons, the separation of mandatory and frequency constraints is highly desirable.

To address the problems with UML multiplicities on n -aries, there are a number of possible solutions. Ideally, multiplicity constraints for associations should be replaced by ORM's mandatory and frequency/uniqueness constraints, at least for n -ary associations. However this is

unlikely to ever happen, and would cause backward compatibility headaches. We could try adding extra constraints for mandatory and frequency for n -ary associations. This would achieve the required expressibility but would make UML even more unnecessarily complex than it is now (e.g. the concept of mandatory role would be dealt with by a multiplicity constraint on binaries but by a mandatory constraint on n -aries). A third solution is to use ORM for the original analysis where the constraints can be easily declared and validated, then map the ORM model to UML where the constraints would appear in notes. Since the ORM notation is easily mastered, and requires no change to the UML notation, the third solution seems attractive, and could certainly be automated.

As a final note on the no-ties example, we might try to overcome the problem of expressing the frequency constraint in UML by transforming the ternary into two binary associations: Country is first in Sport; Country is second in Sport, as shown in Figure 9. However apart from the fact that this transformation doesn't scale (e.g. large numbers of ranks), there are now two constraints that get lost.

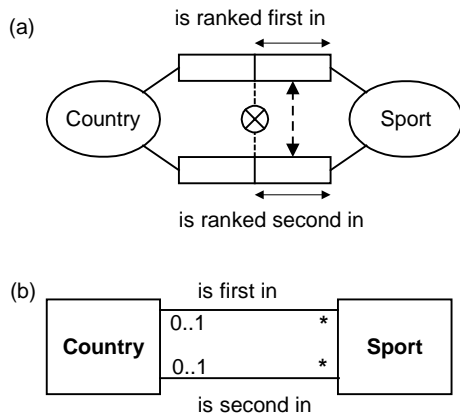


Figure 9 The exclusion and equality constraints in (a) are lost in the UML model (b)

The ORM model in Figure 9(a) shows the missing constraints. The pair-exclusion constraint denoted by a circled “X” enforces the no-ties rule that no country can be ranked first and second in the same sport. The equality constraint shown as a dashed line with arrowheads indicates that if a sport has a winner it also has a runner-up, and vice versa. Although these constraints can be added informally in notes to the UML diagram, it would be better to extend the UML metamodel to support them. Currently UML is very unorthogonal and restrictive with regard to constraints. It supports an exclusive-or (xor) constraint but no exclusive constraint and no inclusive-or constraint. Although UML’s xor constraint is described as applying between associations, it actually applies between

roles. UML supports a subset constraint between full associations but not between parts of associations (e.g. roles). The UML specification also contains a number of inconsistencies in its handling of these constraints. For a formal discussion of such inconsistencies and a means of extending the UML metamodel to adequately capture such constraints, see [19, section 5].

5. Conclusion

Fact-orientation, as exemplified by ORM, provides many advantages for conceptual data analysis, including expressibility, validation by verbalization and population at both fact and constraint levels, and semantic stability (e.g. avoiding changes caused by attributes evolving into associations). ORM also has a mature formal foundation that may be used to refine the semantics of UML.

Object-orientation, as exemplified by UML, provides several advantages such as compactness, and the ability to drill down to detailed implementation levels for object-oriented code. If UML is to be used for conceptual analysis of data, some ORM features can be adapted for use in UML either as heuristic procedures or as reasonably straightforward extensions to the UML metamodel and syntax. These include mixfix verbalizations of associations and constraints for associations, and exploitation of data use cases by populating associations with tables of sample data using role names for the column headers.

However there are some fundamental aspects that need drastic surgery to the semantics and syntax of UML if it is ever to cater adequately for non-binary associations and some commonly encountered business rules. This paper revealed some serious problems with multiplicity constraints on n -ary associations, especially concerning non-zero minimum multiplicities. For example, they cannot be used in general to capture mandatory and minimum occurrence frequency constraints on even single roles within n -aries, much less role combinations. Moreover, UML’s treatment of set-comparison constraints is defective. Although it is possible to fix these problems by changing UML’s metamodel to be closer to ORM’s, such a drastic change to the metamodel may well be ruled out for pragmatic reasons (e.g. maintaining backward compatibility and getting the changes approved).

In contrast to UML, ORM has only a small set of orthogonal concepts that are easily mastered. UML modelers willing to learn ORM can get the best of both approaches by using ORM as a front-end to their data analysis and then mapping the ORM models to UML, where the additional constraints can be captured in notes or textual constraints. Automatic transformation between ORM and UML is feasible, and is currently being researched.

References

1. Barker, R. 1990, *CASE*Method: Tasks and Deliverables*, Addison-Wesley, Wokingham, England.
2. Bentley, J. 1988, 'Little languages', *More Programming Pearls*, 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. Booch, G., Rumbaugh, J. & Jacobson, I. 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading MA, USA.
7. Campbell, L., Halpin, T. & Proper, H. 1996, 'Conceptual schemas with abstractions: making flat conceptual schemas more comprehensible', *Data & Knowledge Engineering*, 20, 1, 39-85.
8. Chen, P.P. 1976, 'The entity-relationship model—towards a unified view of data', *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36.
9. Dey, D., Storey, V.C. & Barron, T.M. 1999, 'Improving database design through the analysis of relationships', *ACM Transactions on Database Systems*, vol. 24, no. 4, pp. 453-486.
10. Embley, D. 1998, *Object Database Management*, Addison-Wesley.
11. Eriksson, H. & Penker, M. 2000, *Business Modeling with UML – Business Patterns at Work*, John Wiley.
12. Falkenberg, E. 1976, 'Concepts for modelling information', *Modelling in Data Base Management Systems*, G. Nijssen ed., North-Holland, Amsterdam, pp. 95-109 (see esp. p. 104, where "properties" means "attributes").
13. Fowler, M. with Scott, K. 1997, *UML Distilled*, Addison-Wesley.
14. Halpin, T. 1995, *Conceptual Schema and Relational Database Design, 2nd edn* (revised 1999), WytLytPub, Bellevue WA, USA.
15. Halpin, T. 1998, 'Object Role Modeling (ORM/NIAM)', *Handbook on Architectures of Information Systems*, P. Bernus, K. Mertins & G. Schmidt eds, Springer-Verlag, Berlin, pp. 81-101.
16. Halpin, T. 1998, 'Object Role Modeling: an overview', available online at <http://www.orm.net/overview.html>.
17. Halpin, T.A. 1998-9, 'UML data models from an ORM perspective: Parts 1-10', *Journal of Conceptual Modeling*, InConcept, Minneapolis USA, available online from www.orm.net/uml_orm.html.
18. Halpin, T.A. 1999, 'Data modeling in UML and ORM revisited', *Proc. EMMSAD'99: 4th IFIP WG8.1 Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, Heidelberg, Germany (June).
19. Halpin, T.A. 2000, 'Integrating fact-oriented modeling with object-oriented modeling', *Information Modeling in the New Millennium*, eds M. Rossi & K. Siau, Idea Group Publishing Company, Hershey, USA.
20. Halpin, T.A. & Bloesch, A.C. 1998, 'A comparison of UML and ORM for data modeling', *Proc. EMMSAD'98: 3rd IFIP WG8.1 Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, Pisa, Italy (June).
21. Halpin, T.A. & Bloesch, A.C. 1999, 'Data modeling in UML and ORM: a comparison', *Journal of Database Management*, vol. 10, no. 4, Idea group Publishing Company, Hershey, USA, pp. 4-13.
22. Halpin, T. & Proper, H. 1995, 'Subtyping and polymorphism in object-role modelling', *Data & Knowledge Engineering* 15, 3 (June), 251-281.
23. Halpin, T. & Proper, H. 1995, 'Database schema transformation and optimization', *OOER'95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, 1021 (Dec.) 191-203.
24. ter Hofstede, A. 1993, *Information Modelling in Data Intensive Domains*, PhD thesis, University of Nijmegen.
25. 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.
26. 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.
27. Jacobson, I., Booch, G. & Rumbaugh, J. 1999, *The Unified Software Development Process*, Addison-Wesley, Reading MA, USA.
28. Kobryn, C. 1999, 'UML 2001: a standardization odyssey', *Communications of the ACM*, vol. 42, no. 10, pp. 29-37.
29. Levesque, H. 1984, 'A fundamental trade-off in knowledge representation and reasoning', *Proc. CSCSI-84*, London, Ontario, 141-52.
30. Muller, R.J. 1999, *Database Design for Smarties*, Morgan Kaufmann, San Francisco, CA.
31. OMG UML Revision Task Force, *OMG Unified Modeling Language Specification*, version 1.3, available online from <http://omg.org/uml/>.
32. Rumbaugh, J., Jacobson, I. & Booch, G. 1999, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading MA, USA.
33. Warmer, J. & Kleppe, A. 1999, *The Object Constraint Language: precise modeling with UML*, Addison-Wesley, Reading MA, USA.
34. www.microsoft.com (online details about Visio Enterprise).