TERRY HALPIN

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 businessís 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 canít 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 youíre used to ER, you might think this approach is a better way of doing things. But itís 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 conceptual 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.

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

TABLE 1. Some data about athletes.

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 donít 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.

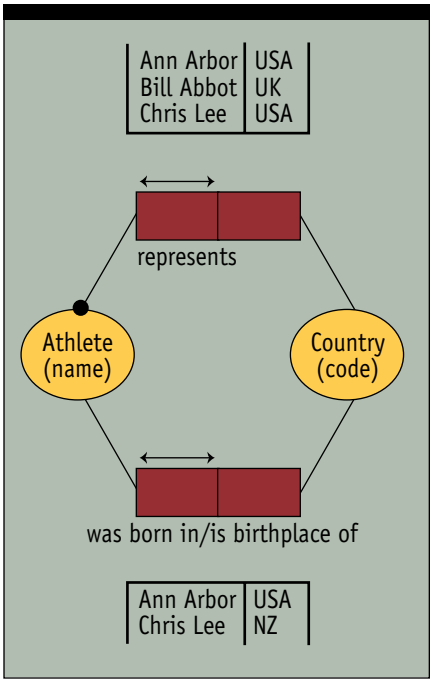

Figure 1. Populated ORM diagram.

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?

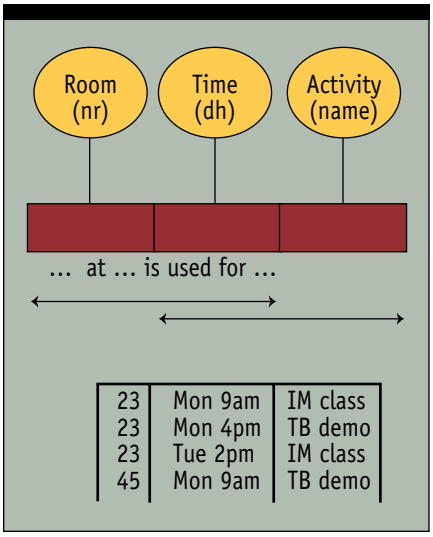

Figure 2. A ternary fact type.

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:

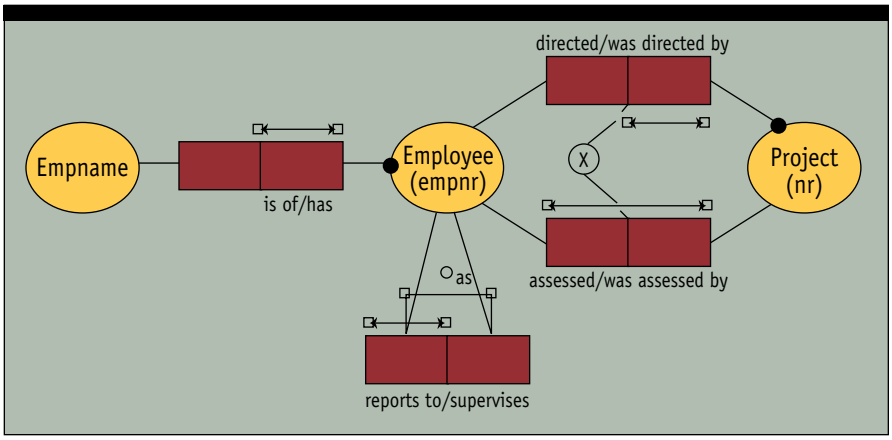

Figure 3. Graphical rule notation in ORM.

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 canít 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 Contract-Manager, which is a subtype of both ContractEmployee and Manager.

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

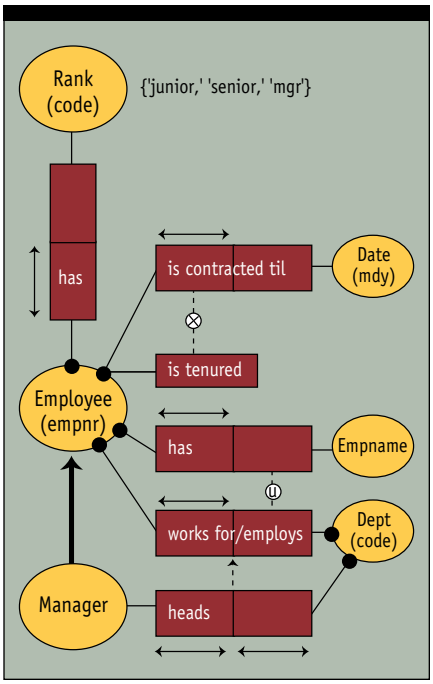

Figure 4. Further constraint examples.

rivation 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 itís 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

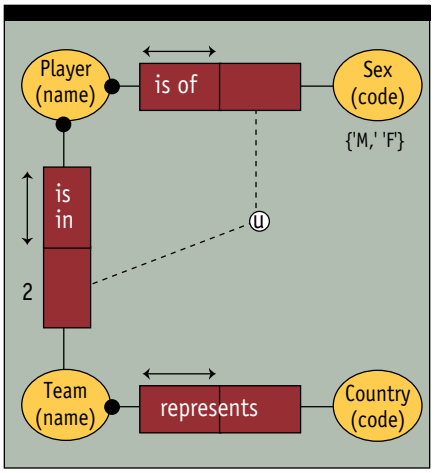

Figure 5. One way to model.

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

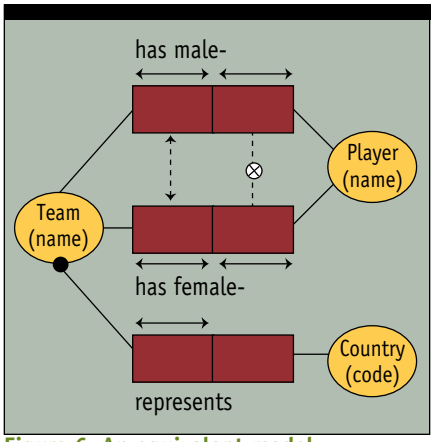

Figure 6. An equivalent model.

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 smallintnot 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):

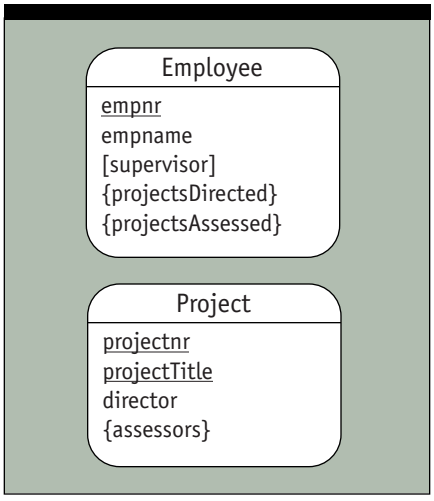

Figure 7. Employee and Project objects.

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

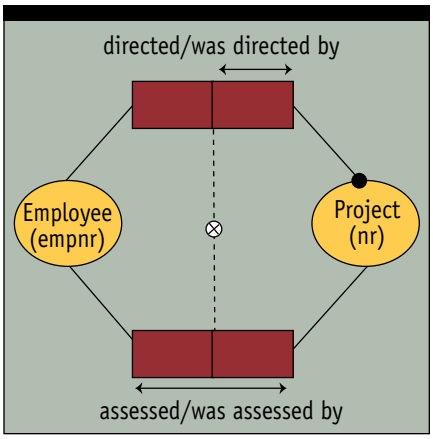

Figure 8. A model fragment.

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)."

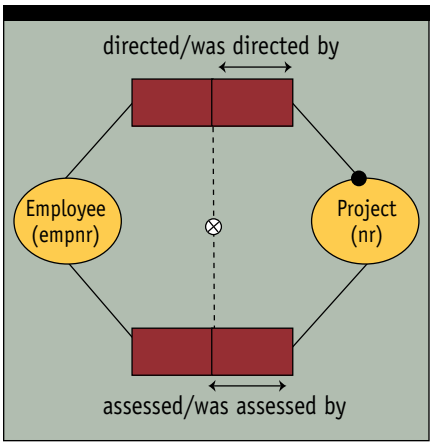

Figure 9. A minor change

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 .