Black Belt Design:

Asymetrix Corp.'s Dr. Terry Halpin

By Maurice Frank

This interview appeared in the September 1995 issue of DBMS and is reproduced here by permission.

Using object role modeling to design relational databases.

For many developers, entity relationship (ER) modeling and relational database design go hand in hand. But a growing number of designers have found database design nirvana using another methodology known as object role modeling (ORM). ORM uses English sentences to express how objects relate to one another, and what constraints and business rules apply to these relationships. While based on techniques dating back to the early 1970s, ORM recently gained popularity with the February 1994 release of InfoModeler, by Asymetrix Corp. (Bellevue, Wash.).

Because ORM expresses design details using English sentences, many developers have found it an effective way to present a database design to non-technical end users who can easily understand it, and correct it where necessary. This ability to verbalize a model is a unique and compelling advantage of ORM.

Dr. Terry Halpin is the man behind the model at Asymetrix. As the head of research at the company, Dr. Halpin guides the implementation of ORM in InfoModeler and other related products Asymetrix is developing. Halpin began working with ServerWare before Asymetrix acquired that firm. He is also a senior lecturer in Computer Science at the University of Queensland, Australia, and the author of Conceptual Schema and Relational Database Design, 2nd Edition, (Prentice Hall, 1995). Terry has a black belt in Judo and a blue belt in Karate.

DBMS Technical Editor Maurice Frank discussed ORM with Dr. Halpin at the IFIP Data Semantics Conference in Stone Mountain, Georgia, in May. An edited transcript of their conversation follows.

DBMS: What is object role modeling?

HALPIN: ORM is a conceptual modeling method for designing information systems. It views the application world in terms of objects that play roles. These roles may be played in isolation (for example, a person jogs) or as parts in relationships (if a person writes a book, the person plays the writing role and the book plays the role of being written). In general, you can have relationships with any number of roles.

DBMS: Is ORM a method for designing a relational database?

HALPIN: It's a conceptual modeling method. It deals with the facts of interest at a conceptual level without committing the designer to any particular implementation structure. We model in terms of elementary facts, so we break the information up into simple units. To regroup these units into structures is typically an implementation issue, not a conceptual issue. Because of the dominance of relational databases, we usually map the conceptual design to a relational database design, but you can map it to network, hierarchical, object-oriented structures, or whatever you want.

There are some aspects of object-oriented databases that are conceptual, but there's a lot of implementation and logical stuff in there. It's not a clean way to begin a design. We design at a conceptual level, independent of where we're going to map it, and then we can take it to whatever database we want.

DBMS: How did you begin working with ORM?

HALPIN: Professors Shir Nijssen, Eckhard Falkenberg, and Dirk Vermeir, three of the pioneers, came to my university. I worked on the first edition of the schema design book with Shir Nijssen and made contact with other ORM pioneers such as Professor Robert Meersman.

DBMS: How did you join Asymetrix?

HALPIN: With the advent of Windows it was becoming very easy for people to build databases without giving much thought to how they are designed. Jim Harding (then president of ServerWare) was looking for a means by which to step above the logical level to drive the application development process. After considerable research on the topic, he read the first 80 pages of the schema design book and said: "Eureka! This is the conceptual framework on which database applications should be based." So I began consulting with ServerWare, and the company built a prototype of a tool that was later acquired by Asymetrix. Asymetrix funded a research laboratory back at my university.

DBMS: What has been your role in developing ORM?

HALPIN: I think the formalization of the method was important. One modeler might describe a universe in one way, and another in another way, but are they really talking about the same world? I showed how to prove formally whether one model was equivalent to another and answer questions like that.

In terms of extending the method, I changed the order in which some steps are done, and I added some steps into the method, such as logical derivations. I changed the way subtyping was done, so it's now run on formal subtype definitions. I added new constraints, such as ring constraints. But it's not just me. There are many people who have been contributing to ORM for a long time.

DBMS: What about the possibility of misunderstandings?

HALPIN: Notice that we're communicating in natural language, which is the normal way to do things, and most people are pretty good at that. Some clients you're interviewing

don't know any formalizations. I agree that natural language is ambiguous, which is good for things like poetry and making puns. However, for building things like database models, we want an unambiguous, formal subset of natural language, such as Asymetrix's FORML [Formal Object Role Modeling Language]. It's completely unambiguous, but also natural, so you get the best of both worlds.

DBMS: How does entity relationship (ER) modeling compare to ORM?

HALPIN: ORM has theoretical foundations in logic and linguistics, providing a rigorous yet natural way of modeling. It has many advantages over ER modeling. For example, ORM allows the information to be verbalized naturally without resorting to artificial constructs, thus improving communication between the modeler and the client. Plus, its graphical notation often expresses more constraints and reveals the underlying domains (the semantic glue that binds an application together). Its diagrams also may be populated with fact instances, allowing safe validation with the client, and it makes no initial use of attributes, so you don't have to agonize over whether some feature is to be modeled as an entity type or an attribute. Its fact-based nature simplifies reengineering and schema evolution. It has a more comprehensive treatment of subtyping, schema transformations, and mapping. For such reasons, ORM is better for developing, transforming, and evolving a conceptual model. Although a handful of ORM dialects exist, their graphical notations are almost identical, unlike the large family of ER notations that are often vastly different from one another.

In terms of fundamental differences, we don't talk about attributes when we do the original model. We're delaying commitment on importance. How important is something? You don't really know until you find out all the things it does and what constraints apply to it. Once you know that, you can apply a simple procedure to the ORM model to generate an ER view with attributes. So, ORM also gives you the best way to develop an ER model, and you've still got the ORM model there to reveal the extra detail.

We're also different because our entire framework is set up so that you can describe things naturally. That means you can populate all of the fact types. The role-based notation lets you express all of these other constraints very simply.

As a simple example, consider the ORM diagram [see Figure 1], which could be a fragment of a model maintained by a book publisher.

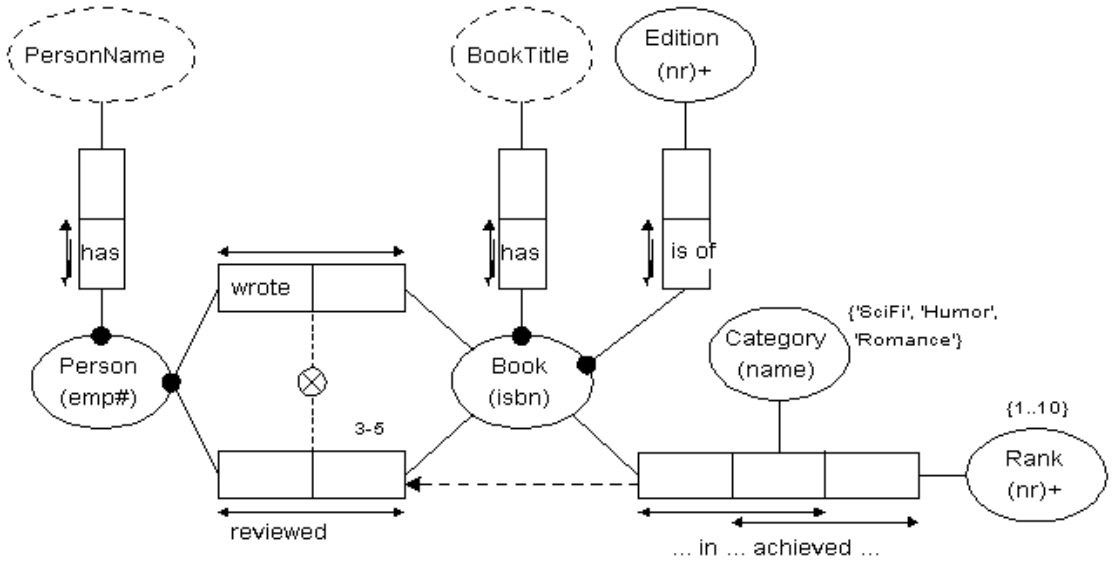

Figure 1: An Object Role Model An Object Role Modeling diagram for a book publisher. The notation expresses various kinds of fact types, roles, and constraints. Sample data may also appear.

Object types are shown as named ellipses. Value types are shown as broken ellipses (for example, PersonName). Because values are just strings or numbers, they identify themselves. Entity types are shown as solid ellipses (Person), and must have an identification scheme (emp#). Roles played by objects are shown as boxes. Fact types have one or more roles (Person wrote Book). Each fact type may be populated with a fact table of sample facts, with one column for each role. An arrow-tipped bar over roles is a uniqueness constraint (entries in those fact column(s) must be unique). For example, each person has at most one name, a person may write many books (and vice versa), and each book-category combination achieves at most one rank. The ternary fact type also has a second uniqueness constraint: Within each category, each rank is achieved by at most one book (that is, no ties). This rule might be debatable, but can be easily checked with the client by populating the ternary fact type with test fact instances. The black dots are mandatory role constraints. For example, each book has a title, and each person wrote some book or reviewed some book (or both). Possible values for category and rank are listed in parentheses. The circled "X" is a pair-exclusion constraint (nobody may write and review the same book). The "3-5" is a frequency constraint (each book that is reviewed has at least three and at most five reviews. Finally, the dotted arrow is a subset constraint (each book that is ranked must have been reviewed).

There is a textual version of the language, so, for clients who don't like diagrams, we can discuss the model in plain English (the English version of FORML -- we also have French and German versions).

DBMS: What's involved in translating an ORM model into a relational model?

HALPIN: Mapping fact types into tables is trivial. We automatically generate a fifth normal-form database by saying each fact type maps into just one table in a way that it can't be repeated. You won't see repetition of an elementary fact, and we are free of

redundancy automatically. This stuff is taught back in Australia to high school students and they just lap it up. What is difficult is to include constraint mapping. Most tools on the market do an incomplete job of constraint mapping, partly because they didn't express constraints well enough in the first place. All you get are keys, and not nulls, and some foreign keys, and that's about it. That's pretty primitive. Because we express all these other constraints, we've got to worry about mapping them too. This is a little bit complex. That's why a tool is worthwhile.

DBMS: How does InfoModeler perform this mapping?

HALPIN: If you've done your ORM model correctly so that every fact type on it is an elementary fact type, you get a fully normalized database automatically. You don't have to go through any normalization. If you need to denormalize it, that's another issue. The current version of InfoModeler does not support denormalization, but a later release will support denormalization in a very safe and transparent way.

DBMS: Have you seen InfoModeler used to design data warehouses?

HALPIN: I've seen people use it for executive information systems very successfully. The conceptual query tool we've got coming out is also a decision-support tool. One nice feature is that you see the connections between everything. The information is connected through the domains. Codd's relational model recognizes that domains are the semantic glue that binds the whole thing together. In ORM, the domains are all there so you can traverse the network through the domains, which makes it easy to see all the links. It's a fantastic way of getting information out. It's so easy. Something in SQL might have a complex correlated subquery with all these joins. Joins are simply moving through a domain onto another predicate.

DBMS: How does ORM compare to Semantic Object Modeling (SOM) by David Kroenke?

[See "Waxing Semantic," DBMS, September 1994, page 60.]

HALPIN: Kroenke had a good idea when he saw that ER has one too many fundamental concepts: It has entities, attributes, and relationships. In ORM we threw out the attributes, and that was the right decision. Kroenke decided to throw out the relationships, and that was the wrong decision. So all he has are objects and attributes, and all the relationships are represented in terms of attributes. That has a number of very bad consequences. One of them is that it makes it impossible to express lots of very important constraints, including exclusion constraints, subset constraints, ring constraints, disjunctive mandatory roles.É I could go on and on. Constraints that are just so easy to express if you've got relationships and a role based notation, you just can't do in Kroenke's model, at least not diagrammatically.

For example, consider the constraint that if a person writes a book, then that person should not review the same book. As we've seen, this is trivial to capture in ORM, but SOM doesn't handle it. Even if a new notation were added to SOM to cater to it, where would you put the constraint? In SOM and OO for that matter, the author and review fact types are encoded twice (in both the Person and Book objects). Would you now encode the constraint twice as well?

Kroenke's SOM is more like a cross between an external and an internal modeling method (like many "object-oriented" approaches).

Kroenke looks at output reports, and he's doing the right thing there. He's got examples, which we use in our method too, and he's trying to go from those examples to the structures. Unfortunately, because of the way he structures things, you can't verbalize the stuff naturally.

DBMS: What happens when rules change over time?

HALPIN: An ORM model is particularly good for schema evolution. You just add or delete fact types, and modify rules as required. If in an ERD of a movie database you have an attribute called director, and later decide you want to record the birthdate of directors, suddenly you've got to change the director attribute into an entity. In ORM you would have modeled directors with a fact type Person directs Movie, and now you just add the fact type Person was born on Date. Schema changes are obviously simpler in ORM.

DBMS: What about exceptions to rules?

HALPIN: One way to handle exceptions is with soft constraints. With hard constraints you reject violations. With soft constraints you issue a warning and maybe take some other action. For example, monogamy is typically the rule in our society, but we know some minority of people are bigamists. So instead of putting a one-to-one constraint on the marriage relationship where that's normally a hard constraint, we can make it soft. We expect each husband to have only one wife, but if we find more than one, we issue a warning or maybe send a message to the appropriate law enforcement agency. You're probably familiar with the event-condition-action model. An update request is an event subject to a constraint condition. If it's a hard constraint you reject the update, but if it's a soft constraint you take another action.

DBMS: Can an ORM diagram show this?

HALPIN: So far we only show the hard constraints, but we have a catch-all constraint for rules that don't fit in elsewhere. We're now talking more about InfoModeler than ORM in general. In later releases we'll support soft constraints.

DBMS: Does ORM deal with behaviors?

HALPIN: The kernel of ORM doesn't address that, but newer versions are being developed to integrate process and event modeling with ORM fairly cleanly. We are working on our own way of using major object types and constructors, and defining operations on those types. In that sense it's looking a bit like an object-oriented approach, but notice that we're doing it as an abstraction on a model in which finer constraints and rules are easier to capture and validate with the client.

DBMS: What criticisms of ORM do you hear most frequently, and how do you respond?

HALPIN: Two misconceptions about ORM are that it is always bottom-up, and it provides so much detail that you can't see the forest for the trees. Well it's important to get

the right forest, and this means getting the trees right too. In fact, for large applications, ORM begins with a top-down division of the application into conveniently sized modules. Within a module, you begin by verbalizing the facts: This part is bottom-up, but that's good because it delays commitment on relative importance. How important a thing is depends on what it does in the application. Until we know this, why waste time guessing?

DBMS: What is the future of ORM?

HALPIN: We and many other researchers are extending ORM in several directions. Apart from the modeling extensions I've alluded to, we're working on conceptual query languages so you can not only specify the model, but you can query the model directly at a conceptual level. With respect to the Book schema, suppose you want to list the employee number and name of each person who wrote a book that got a rank above 5 in the SciFi category. With the tool we're developing you'll be able to query the schema in a way that is analogous to the English sentence just stated. Compare this with the lower-level approach of logical languages, where you have to know what tables the information is stored in and then declare all the necessary joins.

DBMS: That would give you composite or aggregate objects.

HALPIN: Yes.

DBMS: Would it also support nested objects such as a traditional bill of materials?

HALPIN: We already support nested objects. However, you don't really need nesting to model the bill of materials problem: You can just use a fact type like Part directly contains Part in Quantity. With ORM you can also declare the relevant constraints (for example, acyclicity, which involves recursion). When it comes to querying, you also need some kind of recursion. For example, you might say "Give me all the parts of part A whether they're direct or indirect." That involves transitive closure. SQL3 will provide recursive union, for instance, so it will be easier to generate queries like that.

But I am thinking of, for example, a presentation object that might contain a sequence of slides, and then a slide itself may be some kind of set of diagrams or something, and a diagram itself would be an object, and so on. These are recursive structures too, and we've got constructors such as set, sequence, bag, and even schema. So we can build these bigger objects. And when you've got these things you can bind operations to them.

What we're really talking about here is the external level. One good thing about Kroenke's stuff, by the way, is that it's not too bad for external schemas. You've got the conceptual level, the internal level, and the external level. The external level is where people interact, and his method deals more with that. A screen is something I might interact with. When you build these complex objects, and you're talking about operations on them, this is almost like a screen. At the conceptual level you don't commit yourself to what the external stuff is going to look like. Once we say we want to get into external modeling as well, then we say let's group these things, and one way of grouping them is into screens. And here we also apply constructors, and we also bind operations that are the actions that can be performed, or which people can request to be performed.

DBMS: Does ORM have any relationship to building applications?

HALPIN: Yes, InfoModeler doesn't have it now, but we do have a method worked out based on major object types and operations of automatically generating the forms and screens for you. We haven't coded it up yet, but yes, it does lend itself to that. We can do about 80 percent of the work. The user must still do placement and pick colors, and that sort of thing.

DBMS: Is it harder to build an ORM tool than an ER tool?

HALPIN: The answer is yes and no. ORM has a lot more in it. But you can use ORM to do ORM. For instance, I was involved in the meta-modeling when we used ORM to design InfoModeler. ORM is such a powerful technique that it's easier to do a metamodel using ORM than it is using ER. But you've really got to know what you're doing to build one of these tools.

DBMS: Did you use ORM to design the database in which InfoModeler stores its data?

HALPIN: Oh absolutely! We didn't map the ORM metamodel to a relational database either.

DBMS: The CASE market seems to be gaining ground again. What are your views on why this is happening?

HALPIN: It's probably three things: One is hardware, one is software, and the other is people becoming more aware. To get a proper CASE tool, you needed a lot of power (at least a 486 and plenty of RAM). Before a Windows-based environment, these tools were a pain to use. Now that you've got reasonably good performance with an intuitive visual environment to work in, we can give you something to work with that is really going to help you. And now there are people who are beginning to realize it's no good to just shoot from the hip if you want to do a relational database design. Even if it's only got 10 tables in it, if you want it to be a good design, then you shouldn't just go and write down the tables. You've got to think about things first. People have been burnt by too many bad designs. And design is not always the easiest thing in the world to do, so maybe a tool to help you would be a good idea.

DBMS: Where do you see CASE tools going in the next few years?

HALPIN: In InfoModeler you'll be able to take existing applications and reverse engineer them into an object role model. We're also putting in conceptual query tools, so once you've got something specified, you can query directly on those concepts. You can map to SQL and run on almost any system under the sun. Another thing we're working on is all this process event modeling and linking that cleanly to the data modeling. We're also working on object-oriented databases, so we'll be providing more and more support for complex objects with abstractions on top and all the mapping algorithms to take you down there. But unlike typical OO, we'll have all the constraints specified, and we actually map them as well. Chapter 10 of my book discusses some of these ideas.

The amount of business rules we support is going up exponentially. We have several graphic rules, but that doesn't cover every possible rule. In a future release we'll support virtually any other kind of rule. You can roll your own rule and we'll map this for you. It will be even more complete than it is now, so that's pretty exciting.

DBMS: Is InfoModeler a tool for professional developers, or is it also suitable for non-technical end users?

HALPIN: Some end users design their own databases. Consider larger applications that involve subject matter experts and professional designers. The tool could be used by both because it is set up to maximize communication between those two parties. But it's basically a tool for the designers– who we call the modelers.

DBMS: What would you say to someone who has to choose a CASE tool?

HALPIN: If you're building a model, you're typically doing it with the help of some subject matter experts. You typically have to get that application knowledge out of those experts, or at least from examples and reports. Therefore, you should pick a method that's going to improve communication.

You need to promote correct modeling. To get a correct model out of the people who understand the application, it's best to communicate with these people in their terms. You should be able to verbalize the information. You should be able to talk in terms of instances. You need to build a conceptual model, not a logical model committed to a physical structure, so you can support communication.