Converting OO Models into RDBMS Schema

Michael Blaha, William Premerlani, and Hwa Shen
GE Corporate Research and Development

With the advent of new technologies come new ways of dealing with relational database-management systems. The object-oriented paradigm, for example, is useful in designing such systems because object models are expressive, concise, easy to develop, and less prone to the update anomalies that plague attribute-based database-design techniques. Likewise, RDBMSs are a viable implementation vehicle for object models because the technology is mature and many commercial packages are available.

RDBMSs offer unique advantages like a firm theoretical foundation, mature technology that is partially standardized, and the ability to work with a declarative language. Object-oriented database-management systems are also available, and they have their own advantages, like quick navigation of data structures, support of a greater number of data types (like audio and video), and the ability to cleanly integrate with at least one language. However, ODBMSs are relatively new and somewhat unstable, so many developers still find it worthwhile to create RDBMS applications and tools.

The complementary strengths of RDBMSs and ODBMSs make it desirable to have them coexist in some form — getting the best of both worlds. We believe this coexistence will continue for at least the next decade, although the two are likely to service different application niches. RDBMSs will probably stay in business and high-reliability applications, while ODBMSs will be used more in
computer-aided engineering, design, and manufacturing. Eventually this distinction will disappear, however, as new features, like the recognition of inheritance and the enrichment of data types are added to RDBMSs, and as ODBMSs acquire powerful query facilities.

Applications that combine an object-oriented development method with RDBMS implementation can benefit from the use of an approach that separates design concerns from implementation issues. That way, application experts become free to concentrate on design instead of low-level tasks like tuning a database, just as high-level languages free programmers from repetitive tasks like assigning variables to registers.

Any architecture that preserves this separation of concerns must have the ability to hide low-level details from the application expert. One way to do this is to automate mapping rules through an object-model compiler. Experts can then focus on the nuances of the application and avoid rediscov-ering database-design principles for each new project. Moreover, the accuracy of the tables no longer depends on the person doing the conversion.

In this article, we describe such an architecture, which ensures separation with an open approach: the front-end CASE tool that models object diagrams is completely separate from the back-end compiler that generates database schema. For our experiments, we have used OMTtool, a model editor developed by James Rumbaugh of GE Corporate R&D, and Schemer, a batch compiler we developed to generate RDBMS code. Other object-model editors and database-schema generators could be integrated in a similar way.

Both OMTtool and Schemer run on MS-DOS PCs as well as Sun and Hewlett-Packard workstations. Schemer is implemented through Data Structure Manager, an in-house object-oriented language similar to Eiffel. The largest model we have compiled to date has roughly 100 classes; for this model, Schemer generated SQL code in 30 seconds.

ARCHITECTURAL OVERVIEW

Figure 1 shows the architecture for automating object-model-to-RDBMS conversion. The architecture has three main parts: OMTtool, Schemer, and the RDBMS itself. Central to Schemer are the set of mapping rules and the object and ideal-table metamodels that formalize and map the object-modeling notation into database schema.

OMTool. OMTtool stores two views of the object model in OMT modeling notation (described in the box on pp. 34-35. The graphical model describes the geometry of boxes, lines, and text and their placement on the screen. It is used to enhance user interaction and represents the object model visually for documentation. The logical model abstracts the underlying meaning of a picture and describes the classes and how they relate to each other. This distinction between logical and graphical models is similar to that made by Smalltalk's model-view controller.

OMTool has several features that make model generation easier. It lets users create, load, edit, save, and print object diagrams automatically. It maintains logical connectivity as the user moves related entities about the screen. The interface offers pop-up annotation windows that query you for details not shown on the object diagrams, such as attribute domains, permissibility of nulls for each attribute, and primary keys. There is also a window for specifying the mapping rules to be applied to all classes and relationships in the model, which essentially provides a global strategy for database implementation. Users can override the global mapping rules for individual classes or relationships.

Figure 1. Creating an application database and optional metadatabase using OMTtool and Schemer
to fine-tune trade-offs among efficiency, extensibility, and integrity in the RDBMS code.

When the user has made the implementation decisions, OMTool writes them and the object model’s logic to an ASCII file for input to a database-schema compiler, in this case, Schemer.

**Schemer**. Schemer converts the logical file into the SQL statements needed to create RDBMS tables, indexes, domains, access permissions, and other parts of schema. During this process, it generates a log of any errors, like names that are too long or that clash and missing information. It also checks that the restrictions concerning multiple inheritance, derived identity, and data types are satisfied and warns if inconsistent mapping rules are selected.

**Conversion process**. Schemer converts the object model to RDBMS tables in three stages. First, it reads the logical model in the ASCII file produced by OMTool, and populates the object metamodel. Second, it maps object-oriented constructs from the metamodel into ideal RDBMS tables using the ideal-table metamodel. Third, it maps the ideal tables to the SQL dialect supported by the target RDBMS.

Partitioning compilation into these stages simplifies the software because the second stage must be done only once, although the entire process is inherently iterative. Users generally edit an object model, run Schemer only partially, check for errors and evaluate the suitability of the schema. Tables are not actually created until the user is satisfied with the output (somewhat like the compile-link-execute cycle in some languages; you don’t complete the cycle until you fix all the compile errors). The exact details of the third stage depend on the particular RDBMS. We have written five versions of the third stage, one for each RDBMS we directly support: Oracle, Ingres, Sybase, Rdb, and Informix. For other RDBMSs, Schemer provides a compiler switch that invokes a generic version of the third stage, which reads the RDBMS features the user has added to a configuration file in Schemer.

**Metamodels**. The object metamodel in Schemer describes the structure of object diagrams clearly and concisely, including all the features of OMT notation. It also simplifies the implementation of the rules for generating a database schema. Finally, it provides a common data dictionary for multiple RDBMSs, providing information about the object model from which the schema was mapped. This information acts as an extension to the data dictionary normally provided with an RDBMS. With the additional information, users have a common dictionary that can be used with any RDBMS.

The ideal-table metamodel describes all the possible database tables and lets users customize the tables for a particular RDBMS (hence the name ideal). This is useful because despite attempts at standardization, such as the use of SQL, RDBMSs vary slightly from vendor to vendor.

**Optional data dictionary**. In addition to creating application tables, Schemer can manage its own data dictionary.

**Figure 1** shows how the user can opt to have Schemer generate a script (Data dictionary entries) of SQL statements to insert information in its data dictionary. The script describes both the object and ideal-table models.

In certain types of applications, you can use such a data dictionary at runtime to develop a system that is independent of a specific model. To recapture the specifics, the application simply accesses the data in the data dictionary. For example, an RDBMS can use the data dictionary at runtime to discover the structure of database tables.

The Schemer data dictionary itself is an RDBMS schema that we had Schemer generate using the object and ideal-table metamodels as input.

**RDBMS**. The RDBMS creates the tables in the application database and maintains the data in the Schemer data dictionary by executing the SQL statements in the scripts that Schemer generates.

**Limitations**. Our approach has two limitations. First, Schemer requires that the object model satisfy certain constraints:

- Structured attributes are not allowed. All attributes must be simple data types.
- Although multiple inheritance is allowed, each inheritance path must trace to a single, common ancestor.
- Class attributes, which set one value for all the objects in a class, are not allowed. Instead, an attribute is instantiated for each object.

If these constraints are not satisfied, Schemer will report an error and the generated schema will be incomplete.

The second limitation is in the mapping rules. To simplify the OMTool-to-Schemer interface, we decided that Schemer should support only commutative mapping rules. That is, the order in which Schemer applies the rules will not necessarily correspond to the order in which the user specifies them. We recognize that this decision limits optimization somewhat because some categories of mapping rules are not commutative (different sequences will yield different tables).
However, restricting Schemer to commutative rules simplifies the OMTool-to-Schemer interface and Schemer operation.

To increase optimization, we plan to implement transformations in OMTool. A transformation differs from a mapping rule: a mapping rule converts part of an object model into part of an ideal table, while a transformation converts part of an object model into another object model, which is why transformations must occur in the object-model editor, not in Schemer. We plan to extend OMTool to include generic transformations on object models for databases and languages. Each transformation will accept an object-model fragment as input and will return an object-model fragment as output. Thus, we preserve the object model's coherence and, because OMTool transformations would be interactive, commutativity is not required. We also believe that combining OMTool transformations and Schemer's mapping rules is a more effective way to optimize database design.

SAMPLE CONVERSION

Figure 2 shows a sample conversion from an object-model diagram into RDBMS tables. Figure 2a shows the diagram. The tables generated depend on the choice of mapping rules. By default each object class maps to an RDBMS table, although some associations and generalization relationships can also map to a table. The possible tables that Schemer can generate in this example are Airline, Flight, Employee, Pilot, Flight Attendant, and Flight-Flight Attendant.

You can use different rules to control the trade-offs among system performance, integrity, and extensibility without degrading the logical model of the database. For example, you might implement a many-to-one association as either a distinct table or a foreign key (set of columns in one table that are used to reference another table). (Even though many RDBMSs don't support foreign keys explicitly, they still exist conceptually and the developer must be aware of them. Schemer helps by at least making sure that the data types are correct and creating unique or nonunique indexes on the appropriate columns, particularly if there is a qualifier on the association from which the foreign key is mapped.)

In this case, we decided to optimize away the Flight-Attendant and Airline tables, arbitrarily opting to improve performance at the expense of future extensibility. Figure 2b shows two of the remaining four tables (to save space, we omit the Flight-Flight Attendant and Pilot tables). Schemer does not generate output in this form but for illustration we have converted the script into these tables.

Note that we mapped the many-to-many association to a distinct table and chose to bury the other associations as foreign keys in object tables. For example, the Schedule association between Airline and Flight in Figure 2a is implemented with the airline-name attribute. The advantage of a distinct table is that it separates the implementation of the association from the implementation of the classes being associated. Thus, at runtime, you do not have to change the information in the tables that were mapped from the classes when the association information changes. With a buried foreign key, there is one less table to worry about and one less join to perform when posing a query that involves the association mapped as a foreign key. A buried foreign key also lets you enforce existence constraints, such as for each flight there must be an airline.

Also, the Flight table has IDs, although the Employee table does not.
**Abstraction**

**Nonaggregation**

**Aggregation**

**I**

**association**

**Part role**

**Assembly role**

**Abstraction**

**Module**

**Enum value**

- **Enum number**
- **Enum value**

**Domain**

- **Domain name**
- **Emitted string**

**Attribute**

- **Attribute name**
- **Are nulls allowed**

**Qualifier**

- **Candidate key**
- **Role**

**Instance attribute**

**Discriminator**

**Generalization**

- **Concrete**
- **Is overlapping**
- **Mapping strategies**

**N-ary association**

**Object class**

---

**Figure 3.** Object metamodel used by Schemer to hold descriptions of the object models as it compiles schema. The object metamodel also models the part of the Schemer data dictionary concerned with object models.

IDs are artificially generated keys that give objects an identity apart from their attribute values. Schemer supports the use of IDs but does not mandate them, since IDs may or may not be helpful. The user can specify whether or not to use IDs for an entire object model, and override this decision for specific classes in the OMTTool annotation windows.

**WORKING WITH METAMODELS**

The object and ideal-table metamodels are so named because they are models of models. The object metamodel uses OMT notation to describe the logical aspects of object diagrams used by Schemer. The ideal-table metamodel uses OMT notation to describe the logical aspects of ideal tables, which Schemer uses to describe relational tables.

The metamodels have cross links that let you navigate between object-oriented and table-oriented constructs. This can be useful for advanced applications that must exploit the object-oriented semantics of the application but store the data in RDBMS tables.

The cross links also promote traceability between requirements and the delivered code. For future work, they may make it easier to extend Schemer to accommodate schema evolution (generate code only for the changes to the object model), for example.

**Object metamodel.** Figure 3 summarizes the object metamodel, which is not yet complete. We are still working on how to model procedural aspects of object-oriented software and assertions, for example.

- **Enum value.** An enumerated value belonging to an enumerated domain. Color, for example, might have values red, blue, and green. Not all domains are enumerated, such as dollars, which does not have discrete values. OMTTool provides facilities for defining enumerations in an object model.

- **Domain.** A named database data type. Each attribute must be assigned a domain either implicitly or explicitly. Domain has a name and an emitted string, which is the data type declared. Street, for example, might have the emitted value `char(30)`. OMTTool provides facilities for defining domains in an object model.

- **Qualifier.** An attribute can be a qualifier, instance attribute, or discriminator. An attribute has a name and information about whether nulls are allowed. The nulls information is not shown in the object model but appears in the OMTTool annotation window.

- **Instance attribute.** A named property of an object that distinguishes among the set of objects at the “many” end of an association.

- **Discriminator.** An attribute of enumeration type that indicates which property of a class is being addressed by a particular generalization.

- **Module.** A coherent subset of a system containing a tightly bound group of classes and their relationships. Each module has a name and mapping strategies, which specify the default mapping rules to be used for that module. In general, classes in the same module should be more strongly connected with associations and generalizations than classes in different modules. The classes in a module should have a uniform level of detail. Module is a well-known term, appearing in such languages as Modula-2 and Ada, which use it as a construct for grouping related procedures. OMTTool supports partitioning a system into modules.
is a superclass and one or more object classes that are subclasses. The is-concrete attribute in the metamodel tells if every object must be further described by a subclass. The is-overlapping attribute tells if an object is described in more than one subclass.

- **Candidate key.** The minimal set of instance attributes and roles that uniquely identify an abstraction. OMTTool provides an annotation window for defining candidate keys.

- **Role.** The manner in which a class is used in an association. It can be part, assembly, or assoc. The part role in an aggregation is the role that represents composition. An assembly role in an aggregation is a thing that is made up of other things. The notation for an assembly role is a diamond. An assoc role describes the participation of a class in an ordinary association.

- **Abstraction.** Either an n-ary association or an object class. An abstraction recognizes the similarities between classes and associations and captures their common behavior. An abstraction has a name and mapping strategies, which override any mapping strategies for the module in which it is defined. An abstraction is described by zero or more instance attributes. An abstraction may assume different roles in relationships with other abstractions. Each abstraction may have multiple candidate keys.

- **N-ary association.** A relationship among instances of two or more classes. Schemer supports binary and ternary associations. N-ary associations may have instance attributes through the Abstraction superclass.

- **Object class.** A description of a group of objects with similar properties, common behavior, common relationships, and common semantics.

- **Nonaggregation association.** An ordinary association.

Both classes and associations have names, deal with instances, may participate in associations, and have candidate keys. The difference is that objects in a class have intrinsic identity; the identity of links in an association is derived from related abstractions.

Generalization and association are the fundamental relationships in the object metamodel, both of which involve two or more object classes. The difference is that association describes the pattern between two or more instances; generalization organizes the class structure for describing a single instance. In the metamodel, we treat aggregation in a similar way to association. However, aggregation adds properties like transitivity (a part of a part is part of the whole) and the ability to propagate operations and default values (the color of a car may propagate to the doors).

**Ideal-table metamodel.** Figure 4 shows the ideal-table metamodel, which is much simpler than the object metamodel. This is not surprising, since tables form a more primitive basis for modeling than classes, associations, and generalizations.

There are three classes in the ideal-table metamodel:

- **Ideal Table.** An idealized representation of an RDBMS table.

- **Ideal Table Column.** A column in the ideal table.

- **Column List.** An ordered collection of Ideal Table columns.

Between ideal-table and Column List are five named associations:

- **Frequently accessed attribute.** A list of columns that are frequently used together.

- **Prim key.** An arbitrarily chosen candidate key that provides an identity for Ideal Table.

- **Cand key.** A list of columns constituting a candidate key of Ideal Table.

- **Foreign key.** A list of columns constituting a foreign key of Ideal Table.

- **Foreign key reference.** The ideal table to which a foreign key refers. Depending on the target RDBMS and the selected options, Schemer may create a unique index on the list of columns in the generated schema for prim key and cand key. It may also create a nonunique index for each foreign key.

The Attribute and Abstraction classes from the object metamodel are also in the ideal-table metamodel to show how Ideal Table Column relates to Attribute and how Ideal Table relates to Abstraction.

Each Abstraction is implemented with one or more Ideal Tables. An abstraction can correspond to multiple tables if data is distributed across a network or partitioned among tables. Similarly, multiple abstractions can be implemented by one table, depending on the mapping rules used.

![Figure 4. Ideal-table metamodel used by Schemer to hold descriptions of the RDBMS tables as it compiles schema. The ideal-table metamodel also models the part of the Schemer data dictionary concerned with RDBMS tables.](image-url)
Each Ideal Table has many Ideal Table Columns, which derive from attributes in the object metamodel. These columns are grouped into Column Lists for defining primary, candidate, and foreign keys, as well as frequently accessed attributes.

The references to object metamodel classes (Abstraction, Attribute) in the ideal-table metamodel preserves traceability among models. For a specific object class, we can query the Schemer data dictionary and find the corresponding ideal tables.

**MAPPING RULES**

Schemer relies on a set of mapping rules to govern its conversion of the object model into RDBMS tables. The rules are designed both to map the elements of the model and to optimize the resulting RDBMS schema. They handle simple and qualified associations, inheritance, and link classes. Schemer does not support the splitting and combining of classes because we felt that this would best be left to the OMTool transformations.

The user specifies global strategies for selecting rules, which can be overridden for individual classes and associations. We did not want to place too much optimization logic within Schemer because optimization decisions

---

**OMT OBJECT-MODELING NOTATION**

The OMT notation extends the entity-relationship approach by offering new concepts, such as qualifiers and a methodology for both programming and database design. It has three basic constructs: classes, generalization, and association. The object-orientation community has long recognized the importance of generalization and its supporting constructs, inheritance, as a way to succinctly organize classes and improve code reuse and extensibility, respectively. They are less aware of the importance of association, however, which many mistakenly equate with a pointer. (Although you can implement an association by a pointer, pointers are just one of several ways to accomplish this.) Information modelers, on the other hand, have long recognized the importance of associations as a logical concept, as evidenced by the popularity of entity-relationship modeling.

Figure A shows a sample object diagram that contains examples of these three concepts. In the example, an airline has many employees, each of whom has a name and social security number. The two employee categories of interest are pilot and flight attendant.

An airline schedules many flights, each of which is staffed with a pilot and many flight attendants. Both pilots and flight attendants service many flights. Each airline has many aircraft and the same type of aircraft is used by many airlines. For this example, an aircraft is composed of only three elements: engine, instrumenta-

tion, and seats. Finally, flight information may appear in many directories, from official airline guides to on-line databases.

**Classes.** In Figure A, Airline, Employee, Pilot, Flight Attendant, and Flight are examples of classes. A class describes objects with common properties, behavior, and semantics. Thus class Airline describes objects United Airlines, Lufthansa, and Air New Zealand. A class is denoted by a box with the class name in boldface. The second portion of the class box lists attributes which hold properties of the objects in a class. For example, class Employee has attributes name and social security number. In general, the choice of class depends on the semantics of the application and the purpose of the object model.

**Associations.** Figure A contains seven associations, which are depicted as lines that connect class boxes, indicating that the classes underlying objects are rel
can interact in surprising ways. For example, there are several possible rules for implementing a generalization relationship. In an inheritance hierarchy, the schema generated depends on the order in which the rules are processed. This is the primary reason we limited Schemer to commutative mapping rules.

**Simple association.** Schemer implements binary associations with an explicit table or by burying the association in a related table. Explicit association tables improve uniformity of design and extensibility but sacrifice performance in navigating a model. You must also consider multiplicity: the upper limit (one or many) and lower limit (zero or one). You cannot bury a many role because attributes must be single valued. A lower limit of one means that a buried attribute cannot be null. An explicit association table cannot directly enforce a lower limit of one because an association instance that does not exist is not entered in the association table.

When multiplicity permits a choice between an explicit table or a buried association, Schemer by default will bury the association. Switches (specified with OMTool annotation windows) let the user override the default and specify the mapping for the entire model, just one module, or just one class.

Associations have multiplicity and optional role names. Multiplicity specifies how many objects of one class may relate to an object of an associated class. The solid balls in the figure are the notation for many multiplicity, meaning zero or more. Thus, a flight may be staffed with many flight attendants and each flight attendant services multiple flights. A hollow ball (not shown) means zero or one. No ball at the end of an association line means exactly one.

A role name identifies the end of an association and is an optional construct. The pilot and copilot shown next to the Pilot class are role names. A flight has one pilot and one copilot. The same person may serve as pilot on some flights and copilot on other flights.

An association also may be qualified, whereby an attribute forms part of the navigation path. For example, a directory has many flights; each flight within a directory has a unique name. Directory and Flight are object classes and heading is a qualifier attribute. A qualifier attribute is indicated with a box growing out of the side of a class box.

Aggregation is a special case of association. It describes the relationship between a part and a whole and, because both relationships occur between object instances, it is drawn as a diamond added next to the assembly part. An assembly containing many types of parts corresponds to multiple aggregation relationships. Aggregation is the part-whole or a-part-of relationship. For example, in Figure A, Engine, Instrumentation, and Seat are part-of relationships of Aircraft.

**Multiplicity for aggregation relationships.** The example shows some additional aspects of associations. An association (or aggregation) may have link attributes that are properties of the association and not of either class. For example, between seat and Aircraft, there is a many-to-many aggregation. The aggregation is characterized by the link attribute quantity, which specifies the number of different types of seats that can be included in a particular aircraft. The box with a loop to the association line denotes a link attribute.

**Generalization.** A generalization organizes classes by their similarities and differences. In Figure A, Employee is a superclass that refines into two subclasses: Pilot and Flight Attendant. Pilot and Flight Attendant inherit Employee attributes. Each object in the Pilot class is described by both Employee attributes and Pilot attributes. Each object in the Flight Attendant class is described by both Employee attributes and Flight Attendant attributes. Employee type is the discriminator and indicates for each Employee object whether it is further described by the Pilot subclass or the Flight Attendant subclass.

Discriminators are not needed in object-oriented languages but can be useful in traversing relational database tables. The domain of a discriminator is an enumeration type, where each value corresponds to one subclass. A triangle denotes generalization. A discriminator is shown as a name near a generalization relationship.

For those interested in further study of object-modeling notation, we give more details elsewhere.²

**REFERENCES**

Schemer gets the information it needs to apply a mapping rule to an association from the object metamodel. For a nonaggregation binary association between two object classes, for example, the two ends of the association are instances of the assoc role. Schemer gets the name of each role, if any, and the multiplicity information from Role, the superclass of the assoc role. To find the object classes being related by the nonaggregation binary association, Schemer traverses the associations between Role and Abstraction. In this case, each Abstraction will be an object class.

Link attributes do not affect the multiplicity constraint on mapping rules, but do influence the aesthetics of deciding whether or not to explicitly realize association tables. Unrestricted ternary associations infrequently occur and are usually best handled with explicit tables. Aggregation is a special case of association and observes the same mapping rules.

Qualified association. A qualified association relates two classes and a qualifier attribute; the qualifier distinguishes among a set of related objects. Qualification frequently occurs in real problems, often because of the need to provide a context for names.

Figure 5 lists possible multiplicity combinations for qualified associations. Schemer can implement the qualified associations for Figures 5a and 5b with explicit tables or bury them in the B class. However, Figures 5c and 5d require explicit tables because the underlying association without the qualifier is many-to-many.

Figure 6 shows another common example: cascaded qualified associations, a sequence of qualified associations that forms a linear path in an object model. This example is from an application for chemical-equipment design. Each piece of equipment is characterized by a sequence of names: site, plant, section, and equipment. (There are other attributes, but we omit them in the interest of space.) Schemer can either fragment the qualification cascade by generating intermediate IDs (default) or preserve the cascade by using concatenated names to form primary keys.

Inheritance. The object metamodel represents inheritance explicitly through the Generalization class.
can support multiple inheritance and inheritance hierarchies because an object class can participate in multiple generalizations as a superclass or subclass, and an object may be described in more than one subclass by the is-a relationship.

There are four possible ways to handle inheritance:
1. Form one superclass table and multiple subclass tables. The tables share a common primary key. This is the simplest approach and the only approach supported by Schemer.
2. Form one superclass table, multiple subclass tables, and a generalization table. The generalization table binds the superclass primary key to the subclass primary key. This approach may be helpful when merging databases.
3. Form only the superclass table. Push subclass attributes up to the superclass level. Subclass attributes that do not apply to a given object are set to null. This may improve performance but violates second normal form, in which each attribute of a table depends on the primary key and nothing else. For option 3, attributes depend on both the primary key and the discriminator.
4. Form only subclass tables. Replicate superclass attributes for each subclass. It may be difficult for an application to determine which subclass table to search.

Schemer cannot support options 3 and 4 because they are not commutative. We could probably extend Schemer to support Option 2 but we don’t really see a pressing need.

All the generalization mappings suffer from the inability of most RDBMSs to constrain values. Options 1 and 2, for example, can use referential integrity to ensure that each subclass record corresponds to a superclass record, but some RDBMSs cannot enforce that correspondence. For option 3, most RDBMSs cannot enforce the correlation in permisibility of non-null values. For option 4, most RDBMSs cannot constrain values across tables.

**link classes.** A link class is an association whose links can participate in associations with other objects. The object metamodel represents a link class as an n-ary association related to one or more Roles through the Abstraction superclass.

Each link is an instance of the link class. An example is shown in Figure 7. Users are authorized to use workstations. Each user has a home directory for each authorized workstation, but the same home directory can be shared among several authorizations. The loop denotes a link attribute or link class. The link class has characteristics of both a class and an association. Like an association, link objects derive identity from instances of the associated classes. Like a class, the link class is treated as a class relative to associations in which it participates. Although not shown in this example, a link class may have attributes.

Schemer maps a link class to a table as shown in Figure 7. The association between Authorization and Directory (home-directory) has been implemented as a buried pointer in the Authorization table. Schemer could also have implemented it as a separate table.

**Splitting and combining classes.** Classes must be carefully combined when cycles of relationships exist within a model; the instance at the start of a cycle may differ from that at the end of a cycle. Also the sequence of mappings may not be commutative.

Figure 8 shows two examples of models with cycles that have various, but possibly obscure optimization possibilities. The top part of the figure is an object model for the well-known dining-philosophers problem: The problem assumes that an equal number of forks and philosophers are arranged around a circular table; that there is a single fork between each pair of philosophers; and that each philosopher must have two forks to dine with.

Figure 9 shows some possible resolutions. In Figure 9a, the Philosopher and Fork classes are combined along one association. The philosopher and the fork to the left of the philosopher are combined into a single class (although they may appear to be tables, they are actually classes that will eventually be mapped to tables) containing the attributes of a philosopher and of the fork to the left. As a result of the combination, both ends of the association between a philosopher and the fork to the right of a philosopher are attached to the same class. This is called self-association. Not only is self-association confusing, but it creates ambiguity. How do you know which philosopher is on...
Figure 8. Two examples of relationship cycles illustrating how some associations make optimization difficult. (A) The dining philosophers problem and (B) a cycle through an inheritance hierarchy.

Figure 9. Possible schema for the dining philosophers. (A) Philosopher and Fork classes are combined along one association. (B) Associations and classes are combined. (C) Combining is not done. Both associations are buried in the Philosopher class.

into a single class containing attributes of a philosopher as well as the attributes of the forks to the left and right. This causes redundancy. The values of each fork's attributes will be stored twice, once with the philosopher to the left and once with the philosopher to the right. You must then update fork information in two places, which creates a greater chance for error.

As Figure 9c shows, a reasonable implementation is to have at least two classes (or more if the associations are realized with explicit tables).

A similar problem exists for the model in Figure 8b. You can, with certain mapping rules, combine all the classes into a single table, leaving a self-association. However, burying self-associations is confusing; you should generally implement them in a distinct table.

Because combining classes can have these kinds of subtle implications, we chose to leave this capability out of Schemer.

LESSONS

In developing Schemer, we encountered several unexpected and subtle problems that are worth describing.

- **Database interaction.** In the original design, we had to first load the metatables before Schemer could read them and generate application schema. In our current design, we eliminate the loading step because database interaction made Schemer run more slowly, decreased portability, and complicated program execution. Our general guideline is that using a database introduces inertia into a system; it is only worthwhile to use a database if comparable benefits offset the inertia. For that reason, Schemer does not require database services for its internal operation. It can generate application schema from data structures in memory just as easily. A side benefit is that you can avoid creating metatables when they are not needed.

- **Domains.** Initially, the Domain class in the object metamodel included subclasses Integer, Real, Character, and Enumerated, and we had to define each application domain with an entry in the appropriate subclass. These subclasses complicated the addition of new Domain categories and awkwardly handled variations in RDBMS data-type syntax. Also, the minor distinctions among domain subclasses cluttered the metamodel. The current object metamodel includes only a single domain class with attributes domain name and emitted string. The literal emitted string is passed through to the RDBMS as Schemer generates create-table commands.

- **Version control.** It was hard to manage old and new versions of a database design. Before we could run Schemer's output, we had to first delete any old schema in the database. We considered deleting old schema using a metaquery (a query on a data dictionary expressed in such a way that the result of the query is itself a sequence of SQL statements) or using a lag file (a file saved from a previous run of Schemer). Unfortunately, the metaquery introduced a dependence on the data dictionary, making Schemer more difficult to port. Other than that, it was an elegant solution and could accommodate database changes made outside Schemer. The lag file, on the other hand, was vulnerable to inconsistency with the database, but it was simple to port across RDBMS platforms. We opted for the lag file, using careful programming of a Unix command shell to avoid inconsistency. Now each time Schemer runs to com-
pletion, lag files delete the tables from the previous run and optionally delete previous entries in the Schemer data dictionary. New tables (and optionally new data dictionary entries) are then created, and finally new lag files.

- **Bootstrapping.** Originally, we manually sketched object diagrams for both the OMT tool graphical and logical models and manually translated these diagrams into data-structure definitions. As OMT tool matured, we were gradually able to treat it as just another application and use it to update itself (revise its graphical and logical models and generate new data-structure declarations, for example). We now manually update only the procedural portions of the code. Similarly, we initially bootstrapped Schemer by manually drawing the Schemer metamodels and manually converting the metamodels to RDBMS tables. Since then, we have closed the loop and now use OMT tool to construct the Schemer metamodels and Schemer to compile its own metamodels.

Using our own software has forced us to recognize its limitations and failures. Because it is easy to update object diagrams with OMT tool, we have been able to keep our models up-to-date, improving documentation. Object diagrams have clarified our understanding of OMT tool and Schemer. We have saved time and reduced errors by using Schemer to generate RDBMS code for its metamodels.

- **Adding RDBMSs.** In the first version of Schemer, we supported only Ingres, Oracle, and Sybase with users selecting a particular RDBMS as they do now, through a switch. Later, we added Rdb and Informix. There was enough standardization among these RDBMSs that adding another took only a few days. This experience gave us the insight to fix Scheme so that any RDBMS could be supported. We added a configuration file that describes the target RDBMS. To add a new RDBMS, the user merely modifies the configuration file to match the new specifications.

We believe our architecture will be of value to both RDBMS and object-oriented system developers. We plan to extend the capabilities of Schemer and OMT tool in several different directions:

- **Transformations.** Schemer currently performs commutative mappings. We'd like to extend OMT tool to support general transformations for program and database design so that the user can visually evaluate their effects.

- **Schema evolution.** As an object model is updated, we'd like Schemer to generate code only for the changes to the model. This is particularly important if you must modify a populated database.

- **Object-oriented interface.** An object-oriented interface to a database generated by Schemer would let users think in object-oriented terms both during compilation and at runtime.

- **Physical database design details.** We have experimented with several approaches: for specifying the extent size for each table and the assignment of tables to physical files, including an ideal-table editor, direct SQL update of the Schemer data dictionary, and supplemental files merged into the Schemer data dictionary. We have not found an entirely satisfactory solution to date.

- **Three-schema architecture.** This would let users specify views derived from an integrated conceptual model.

**ACKNOWLEDGMENTS**

We thank James Rumbaugh and Rajiv Gupta for their helpful comments. We also thank Marc Laymon, who maintains the source code for Schemer.

**REFERENCES**


**Michael Blaha** is a principal at OMT Associates, a consulting and training firm. He was formerly a computer scientist at GE Research and Development Center. His research interests are object-oriented concepts and methods, architecture and design of database applications, and unusual engineering problems.

Blaha received a DSc in chemical engineering from Washington University in St. Louis. He is a coauthor of Object-Oriented Modeling and Design (Prentice-Hall, 1991) and a member of the ACM and the American Institute of Chemical Engineers.

**Hwa Shen** is an electrical engineer at GE's Research and Development Center, where he is a member of the team that developed Schemer. His interests are in object-oriented methodology and database technology and applications.

Shen received an EngScD in electrical engineering from Columbia University.

William Premerlani is a computer scientist at GE's Research and Development Center, where he is a member of the team that developed Schemer. His research interests include object-oriented methods, applications, and database technology.

Premerlani received a PhD in electrical engineering from Rensselaer Polytechnic Institute. He is a coauthor of Object-Oriented Modeling and Design (Prentice-Hall, 1991) and a member of the ACM and IEEE.

Address questions about this article to Premerlani at GE Corporate R&D, Bldg. K-1, PO Box 8, Schenectady, NY 12301; premerlani@crd.ge.com.