Relational Database Reverse Engineering: A Model-Centric, Transformational, Interactive Approach Formalized in Model Theory

David W. Embley
Brigham Young University
Provo, UT 84602, U.S.A.
embley@cs.byu.edu

Mingkang Xu
Lynden Corporation
Idaho Falls, Idaho, USA
richardx@lyndencorp.com

Abstract
Approaches to relational database reverse engineering often expect that the input has desirable characteristics and that it is complete; they also often fail to provide formal guarantees that their results are faithful to the initial input. Both of these problems can be addressed by using an incremental approach based on a formally defined target model. The incremental approach we propose here quickly produces an initial model instance that is provably equivalent to the original relational database, which is assumed to be correct but may lack desirable characteristics and may be incomplete. The approach then proceeds incrementally using provably correct transformations. These incremental transformations allow for user interaction to provide needed information that may be missing or hard to obtain because the input lacks some desirable characteristics.

1 Introduction
Over the past few years several proposals have been made for reverse engineering a relational database into a conceptual model. (See, for example [2] and [7], and the list of references in [9].) Much has been accomplished, but much remains to be done.

This paper advocates an approach to reverse-engineering an SQL database that is (1) model-centric, (2) transformational, (3) interactive, and (4) model-theoretic. The approach is model-centric because it makes an immediate translation to a rudimentary semantic model instance and lets users do all further work in terms of the model. The approach is transformational because it first translates to an initial model instance and then translates step-by-step from the initial model instance to a final model instance. The approach is interactive because both system and user contribute appropriately — with the burden being shifted as much as possible to the system. The approach is model-theoretic because we use model theory as a basis for proving that the transformations preserve information and constraints. This paper describes a tool we have developed that supports this approach to reverse engineering.

Besides advocating a specific method and tool, this paper also proposes the use of the Object-Relationship Model (ORM) as a target for reverse engineering relational databases. The ORM is the data modeling component of OSA (Object-oriented Systems Analysis) [3]. We make this choice because the ORM has a rich but simple semantics and is formally defined. ORM semantics are based on only two simple constructs — object sets and relationship sets — rather than three — entities, relationships, and attributes. This means that we have less work to do because we only have to distinguish among two concepts, not three. It also means that we can always connect any objects sets with a relationship set without regard to whether the object sets are playing the role of an attribute or an entity. Furthermore, we do not lose the richness of attributes, because we have shown elsewhere [5] that we can derive attributes from ORM object and relationship sets and ORM cardinality constraints. (We return to this point later when we show how to derive high-level views from ORM model instances.) The ORM has a formal definition, which is based on model theory. This provides us with a basis for our model-theoretic approach to proving that our transformations preserve information and constraints.

2 Tool Transformations
Our tool is called SQL2ORM [9]. It assumes that the input is a set of SQL Create Table statements, along with Create Index statements and Create View statements. The output is an ORM model instance. As implemented, the tool produces an ORM model instance in terms of a model-equivalent language that is fully compatible and integrated with OSA and that can textually represent every graphical construct [6]. Thus, the output of the tool is actually in a textual form, but since a model-equivalent language is one-to-

372

0-8186-8147-0/97 $10.00 © 1997 IEEE
one with a graphical representation, we use the graphical representation here to present our work. Further enhancements to the SQL2ORM prototype would be necessary to make it work as we would wish. As it is, however, the prototype shows that the basic ideas are in place and are workable.

Our method for translating a relational database scheme to an ORM model instance consists of a set of rules such as what kinds of information are converted to object sets and what kinds are converted to relationship sets, how to handle primary keys and foreign keys, and how to create generalization/specialization hierarchies. The rules, however, may not always yield unique results. In these cases, SQL2ORM either consults the user for a decision or makes a decision and then either asks for a confirmation or allows the user to make some final adjustments to the translation.

We first give the basic transformations and then illustrate with an example, which is based on the sample SQL database in [4]. As given, the example has enough information for the tool to work entirely on its own. SQL2ORM can also work with less information. In this case the tool operates interactively to obtain the information it needs to accomplish the same task.

Table 1, along with the explanation below, gives a succinct summary of both the criteria we seek to do our transformations and the transformations themselves. As we work through our example, we explain some of the details of several of these transformations. Full details are in [9].

Case 0: (initial transformation)  
Transform the table name to a nonlexical object set (an object set whose elements are object identifiers — OIDs) and each attribute name to a lexical object set (an object set whose elements represent themselves — e.g., strings); connect these object sets with relationship sets and retain all key information, information about nulls, and all foreign-key information.

Case 1: (creation of a generalization/specialization)  
If a is a PRIMARY-KEY attribute of table A and b is a PRIMARY-KEY attribute of table B, then for $A[a] \subseteq B[b]$, we create a generalization/specialization relationship set in which $A$ becomes a specialization, $B$ becomes a generalization.


Case 2b: For $A[a] \subseteq B[b]$, we create a new relationship set between $A$ and $B$.

Case 3: (creation of a new object set and a generalization/specialization)  
For $A[a] \subseteq B[b]$, we create a new object set and a generalization/specialization between $A$ and the new object set in which $A$ becomes a specialization.

Case 4a: For $A[a] \subseteq B[b]$, $A$ becomes a relationship set that connects all the lexical object sets embedded in $A$.

Case 4b: For $A[a] \subseteq B[b]$ we create a new relationship set that connects $A$ and $a$.

Case 5: (creation of a new relationship set)  
Let $A_1$ be an object set and $a_{11}, a_{12}, \ldots, a_{1j_1}$ constitute a PRIMARY KEY (or a COMPOSITE PRIMARY KEY) of $A_1$; and $\ldots$; and let $A_n$ be an object set and $a_{n1}, a_{n2}, \ldots, a_{nj_n}$ constitute a PRIMARY KEY (or a COMPOSITE PRIMARY KEY) of $A_n$; and let $B$ be an object set and $b_{11}, b_{12}, \ldots, b_{1j_1}, b_{21}, b_{22}, \ldots, b_{2j_2}, b_{31}, b_{32}, \ldots, b_{nj_n}$ constitute a COMPOSITE PRIMARY KEY of $B$. If $B[b_{11}, b_{12}, \ldots, b_{1j_1}] \subseteq A_1[a_{11}, a_{12}, \ldots, a_{1j_1}], \ldots, B[b_{n1}, b_{n2}, \ldots, b_{nj_n}] \subseteq A_n[a_{n1}, a_{n2}, \ldots, a_{nj_n}]$, then object set $B$ becomes a relationship set with connections to $A_i$ ($1 \leq i \leq n$).

Figure 1 shows the input SQL for our example, and Figure 2 shows the output. SQL2ORM first applies the initial transformation (Case 0) to each of the tables in Figure 1. This initial transformation converts SQL to an equivalent ORM model instance and thus lets the user work within the conceptual modeling paradigm from the very beginning. As an example of the initial transformation, we can see in Figure 2 the results for transforming the CITY-POP table — the only table for which the there is not a subsequent transformation that alters it. In this initial transformation the name of the SQL table, CITY-POP, becomes a nonlexical object set and the attributes, City and Population, become lexical object sets. In an ORM diagram we denote nonlexical object sets by solid rectangles and lexical object sets by dotted rectangles. The initial transformation always adds a has relationship between the lexical object set and each nonlexical object set. Because City is a PRIMARY KEY, the participation constraints for the relationship set to City are 1's on both connections. For the relationship set to Population, the participation constraints are 0:1 on the CITY-POP side and 1:* on the Population side — the 0 in 0:1 because Population may be null, the 1 in 0:1 because there is at most one Population; the 1 in 1:* because any recorded Population associates with CITY-POP, and the * in
Table 1: Transformations

1:* because a Population value could potentially be the population for more than one city.

Create Table CITY-POP (City char(20) PRIMARY KEY,
Population integer)

Create Table COUNTRY (Name char(20) PRIMARY KEY,
Capital char(20) NOT NULL UNIQUE,
Currency char(10) NOT NULL UNIQUE,
Foreign Key(Capital) References CITY-POP(City))

Create Table CURRENCY-WITH-VALUE (Currency char(10) PRIMARY KEY,
Value_in_Dollars real,
Foreign Key(Currency) References COUNTRY(Currency))

Create Table MEMBERSHIP (Country char(20) NOT NULL,
Organization char(25) NOT NULL,
Entry_Date date,
PRIMARY KEY(Country, Organization),
Foreign Key(Country) References COUNTRY(Name))

Create Table EXPORT (Supplier char(25) NOT NULL,
Consumer char(25) NOT NULL,
Amount real,
PRIMARY KEY(Supplier, Consumer),
Foreign Key(Supplier) References COUNTRY(Name),
Foreign Key(Consumer) References COUNTRY(Name))

Create Table COMPANY (Name char(25) PRIMARY KEY,
Country char(20),
Revenues real,
Foreign Key(Country) References COUNTRY(Name))

Create Table EUROPEAN (Country char(20) PRIMARY KEY,
Population integer,
Foreign Key(Country) References COUNTRY(Name))

Figure 1. Sample SQL.

After applying the initial transformation to each table, our SQL2ORM tool performs the actions described below. Although not mentioned above, because it did not arise in the example, the initial transformations records all the foreign-key constraints for each table in the form we show below. The initial transformations also record the co-occurrence constraints Supplier, Consumer \( \rightarrow \) Amount and Country, Organization \( \rightarrow \) Entry_Date, which is the way we record composite keys.

SQL2ORM applies Case 1 to the translated foreign-key constraint EUROPEAN[Country] \( \subseteq \) COUNTRY[Name] to create the generalization/specialization between COUNTRY and EUROPEAN. When a primary key has a foreign-key reference to another primary key, the objects represented by the tables are usually the same. A European country is a country. In an ORM diagram
generalizations attach to the apex of a triangle, and specializations attach to the base.

SQL2ORM applies Case 3 to CURRENCY-WITH-VALUE to create the new object set CURRENCY and make CURRENCY-WITH-VALUE a specialization of CURRENCY. When a primary key has a foreign-key reference to an attribute that is not a primary key, the objects represented by the referencing table are usually the same as the objects represented by the referenced attribute. Currency in CURRENCY-WITH-VALUE is a currency. Observe in Figure 2 that OSA, which only has object sets (no attributes) allows us to make the table name CURRENCY-WITH-VALUE a specialization of Currency, except that we must provide a one-to-one correspondence with a new object set CURRENCY so that both the generalization and specialization are nonlexical. Because OSA does not distinguish between attributes and object sets, there is no requirement to restrict inheritance to object sets (in fact we often do not know whether table names or attributes represent the objects we eventually want until after we have done some analysis).

SQL2ORM applies Case 2a to MEMBERSHIP to create the ternary relationship set MEMBERSHIP(Country, Organization, Entry_Date). When an attribute that is not a primary key has a foreign-key reference, the connection is usually a relationship set with a role. In our example, Country is not a key for MEMBERSHIP, but it is a role of a country in a relationship among organizations, entry dates, and countries. In OSA a role on a relationship set is a specialization denoting those objects in the connected object set that participate in the relationship. If we wish we can explicitly show the specialization by using the triangle notation we have used elsewhere for generalization/specialization, but we may also simply add the name (Country in this example) to the connection as we have done in Figure 2.

SQL2ORM applies Case 2b to COUNTRY[Capital] \subseteq CITY-POP[City] to create the Capital, COUNTRY relationship in Figure 2 and also applies Case 2b to COMPANY[Country] \subseteq COUNTRY[Name] to create the Country, COMPANY relationship. These cases are similar to Case 2a where a non-key attribute has a foreign-key reference. The difference is that for Case 2b the role name applies to referenced table so that Capital is a role of CITY-POP and Country is a role of COUNTRY.

SQL2ORM applies Case 5 to the two foreign key constraints in EXPORT to create the ternary relationship set EXPORT(Supplier, Consumer, Amount). In this case we have multiple non-key attributes that reference the same or different tables. Since each referencing attribute becomes a role, SQL2ORM adds the role names Consumer and Supplier as Figure 2 shows.

SQL2ORM does these transformations automatically. Since there are alternatives in Table 1 for COUNTRY[Capital] \subseteq CITY-POP[City] and CURRENCY-WITH-VALUE\subseteq COUNTRY[Currency], SQL2ORM explains what the alternatives are and asks for a confirmation of its default choice or lets the user select an alternative. The alternative in both examples here is Case 1. In the first example, the alternative would have made COUNTRY a specialization of CITY-POP, which a user would reject. This is sometimes a reasonable alternative because Capital is a key (although not a primary key) for COUNTRY and City is a key for CITY-POP. If these were both the main or only identifiers, it may be reasonable to assume that they are identifiers for the same kind of object — but this is clearly false in this example. In the second example, the alternative would have made CURRENCY-WITH-VALUE a specialization of COUNTRY, which a user would also clearly reject.

3 Further Transformations

Once we have a basic diagram, we have the option to continue making transformations to improve the diagram. A number of transformations are possible,
however, we only discuss the ones that apply to our example.

One of these transformations is lexicalization, which reduces a one-to-one relationship set between a lexical and a nonlexical object set. In Figure 2, for example, we can apply lexicalization to City and CITY-POP. This discards the relationship set and replaces CITY-POP by City. Similarly, we can replace CURRENCY by Currency and in this case also propagate the lexicalization down the generalization/specialization hierarchy to replace CURRENCY-WITH-WALUE by Currency.CURRENCY-WITH-WALUE. We could also apply lexicalization to COUNTRY and Name, but in this case, we lose more than we gain. We need not apply lexicalization unless it helps clarify the ORM model instance.

We can always rename object sets and relationship sets and add or delete role names. For example, we may choose to rename the object set EUROPEAN to be EUROPEAN COUNTRY and to rename the Country, COMPANY relationship as is in with a reading direction arrow from COMPANY to Country so that the full name of the relationship set would be COMPANY is in Country. Both roles Country on COUNTRY are redundant, so we delete them. We also delete the role for Currency in the relationship set connected to Value-in-Dollars because it serves no useful purpose. EUROPEAN COUNTRY is also a role, and we can choose to represent it as a role and for consistency again rename it to be European Country.

To provide a more abstract view of our diagram, we can determine which object sets should be attributes, and we can place these attributes inside high-level object sets. Whenever we have a lexical object set whose only attachment is to a nonlexical object set, we can assume that the lexical object set is an attribute for the nonlexical object set. Thus, for example, Name and Population are attributes of COUNTRY and Name and Revenues are attributes of COMPANY.

Extending this idea further, we may nest lexical object sets inside other lexical object sets to create a hierarchy of lexical object sets. In our example, we may nest Population in City and place this inside COUNTRY. Similarly, we may nest Value-in-Dollars in Currency and also place this inside COUNTRY. After making these changes, we have the ORM diagram in Figure 3a. If we wish, we can also implode the high-level object sets to obtain the more abstract ORM diagram in Figure 3b. More discussion on these ideas for abstraction can be found in [6].

4 Proofs for Transformations

To prove that the ORM model instance generated by SQL2ORM preserves the information and constraints in the given SQL relation schemes, we use model theory [8]. We translate both the SQL statements and the generated ORM model instance to an equivalent model-theoretic view of a relational database [1]. We then prove the equivalence of these two model-theoretic views.

To translate SQL statements to an SQL model-theoretic view of a relational database, we write relation names as predicates that have one place for each attribute, and we write constraints as closed well-formed formulas over the predicates. To translate an ORM model instance to a model-theoretic view of a relational database, we write predicates for each object and relationship set, and we write a closed well-formed formula for each explicit and implicit constraint. For an object set whose name is S, we write the one-place predicate S(x), and for an n-ary relationship
set whose name is $N$, we write an $n$-place predicate $N(x_1, x_2, \ldots, x_n)$. The translations are detailed, but straightforward [9].

To avoid using "model" both in the model-theoretic sense and as a reference to a model from which we create a model instance, we call an interpretation for which all closed formulas hold a valid interpretation. With this understanding, we base our proofs on the following two definitions:

**Definition 1** Let $R = \{ R_1(X_1), R_2(X_2), \ldots, R_n(X_n) \}$ be a database scheme or a model-theoretic database view of an ORM model instance. Let $S$ be an ORM model instance. A transformation from $R$ to $S$ preserves information if for every valid interpretation $I_R = \{ r_1, r_2, \ldots, r_n \}$, where $r_i$ is defined on the corresponding scheme $R_i$ ($i = 1, 2, \ldots, n$), there exists a valid interpretation $I_S$ corresponding to model instance $S$, such that $I_R$ can be derived from $I_S$.

**Definition 2** Let $R$ be a database scheme or a model-theoretic database view of an ORM model instance whose set of constraints is $C_R$. Let $S$ be an ORM model instance whose set of constraints is $C_S$. A transformation from $R$ to $S$ preserves constraints if $C_S$ implies $C_R$.

Given these definitions, we prove a series of lemmas showing that each of the cases in Table 1 preserves information and constraints. With the proofs of these lemmas in hand, the proofs of the following two theorems are immediate.

**Theorem 1** The SQL2ORM transformations preserve information.

**Theorem 2** The SQL2ORM transformations preserve constraints.

See [9] for the detailed proofs.

5 Concluding Remarks

We have described a tool SQL2ORM that works with a user to transform a relational database scheme given in SQL to an ORM model instance. Users can enhance the result by renaming, by rearranging object and relationship sets, and by converting the result to a higher level view. Each transformation SQL2ORM makes is provably information preserving and constraint preserving.

References


