Transforming Conceptual Models to Object-Oriented Database Designs: Practicalities, Properties, and Peculiarities

Wai Yin Mok and David W. Embley

Brigham Young University, Provo, Utah, USA
email: {wmok, embley}@cs.byu.edu

Abstract. More work is needed on devising practical, but theoretically well-founded procedures for doing object-oriented database (OODB) design. Besides being practical and having formal properties, these design procedures should also be flexible enough to allow for peculiarities that make applications unique. In this paper, we present and discuss an OODB design procedure that addresses these problems. The procedure we discuss is practical in the sense that it is based on a common family of conceptual models and in the sense that it does not expect users to supply esoteric, difficult-to-discover, and hard-to-understand constraints (such as multivalued dependencies), nor does it make hard-to-check and easy-to-overlook assumptions (such as the universal relation assumption). At the same time, the procedure is well-founded and formal, being based on a new theoretical result that characterizes properties of interest in designing complex objects. It is also flexible and adaptable to the peculiarities of a wide variety of applications.

1. Introduction

Database design has had a long history. Object-oriented database design is more recent and "clearly needs more work" [11]. Over the long history of database design, many ad-hoc and empirical techniques have been used. There has also been a flurry of theoretical research that has provided us with some interesting insights, but has sometimes disappointed us in terms of practical realities. How can we bring together and extend the best of this work for use in designing object-oriented database systems?

Addressing this topic is an arduous task, and we do not pretend to have all of the answers. Nevertheless, we present here a point of view that builds on and extends past successes, is practical, has a formal foundation with provable properties of interest, and makes adjustments for several types of application peculiarities. In doing so, we do not provide a broad, superficial coverage of this vast area, but rather succinctly provide our contributions in the form of some specific model transformations and algorithms for scheme and method-signature generation, and in the form of several specific adjustments based on insights about application peculiarities and theoretical anomalies.

Succinctly stated, our procedure is as follows. We use as a foundation a particular type of conceptual model that is sufficiently rich for modeling applications of interest, but not so esoteric as to be unusable by analysts and designers with ordinary abilities.† Based on this conceptual model, we characterize and consider a restricted

† As anecdotal evidence on this usability point, we and others have successfully used the model we discuss here in hundreds of hours of work with employees of the Utah Division of Family Services for analyzing their Child Welfare System.

set of model instances, namely those from which we can derive a particular type of acyclic hypergraph and from which we can automatically extract the constraints we need.† Based on the hypergraph properties and on the extracted constraints, we present an algorithm to generate initial schemes with formal properties such as elimination of potential redundancy with respect to the constraints extracted. We then make adjustments for application characteristics such as large objects, update frequencies, and computations, and we generate a final set of schemes and method-signatures for OODB designs.

To bring this all together in a single paper, we build here on some earlier work we have done. The conceptual model we use is described elsewhere [6], as are the basic ideas we use for hypergraph generation and for design transformations [5]. We say enough here about the model, hypergraph generation, and design transformations to make the paper self contained, but we minimize this discussion to leave room for discussing the unique contributions of this paper. These include (1) a precise characterization of a restricted set of model instances from which we can generate designs, (2) a scheme generation algorithm that yields an initial set of schemes with some expected formal properties of interest for design, (3) a set of adjustments to tailor these initial designs for particular application characteristics, and (4) a way to do method placement for object-oriented designs.

Our contributions differ from those of others who have addressed these problems. The procedures presented in some recent books [2,9,16] show how to derive database designs from conceptual models. However, their models are different, as is the theoretical basis for their procedures, and their main focus is on standard database design rather than on OODB design.

We proceed with the development of our approach as follows. In Section 2, we present a running example and use it to illustrate our approach to conceptual modeling, hypergraph generation, and design transformations. We also define the particular type of hypergraph we need for our restricted set of model instances. In Section 3, we give our basic scheme-generation algorithm, show that it has the properties we expect. In Section 4, we augment our algorithm to take application characteristics into account. We summarize and conclude in Section 5.

2. Model Instances and Model-Instance Properties

The conceptual model we use is *OSA — Object-oriented Systems Analysis* [6]. Figure 1 shows the OSA model instance we use as a running example in this paper. The rectangles in the OSA diagram in Figure 1 are *object sets* and the labeled connections among object sets (with or without a diamond) are *relationship sets*. Dashed rectangles represent a set of *lexical* objects, whose representations are strings or specialized strings such as integers or percentages, or are images of various types. Solid rectangles represent a set of *non-lexical* objects, whose representations are object identifiers (OIDs). Role names, if given, are object sets representing the subset of objects participating in a relationship set. Numbers and number-pairs on relationship sets are *participation constraints*, which constrain the number of times an object in the connecting object set participates in a relationship set — 1 for exactly once, 1:* for one or

† We, and others [7], believe that most real-world applications satisfy these restrictions. When these restrictions do not hold, knowing what to do is application dependent, but resolvable based on time-space arguments. Lack of space prevents us from exploring these application-dependent, time-space arguments here.

more, and 0:* for zero or more. Constraints of the form $A_1, ..., A_n \rightarrow B_1, ..., B_m$ are *co-occurrence* constraints, which are (generalized) functional dependencies (FDs) that constrain the cardinality of objects co-occurring in tuples of objects in an n-ary relationship set. OSA also has a behavioral component that lets users specify both the individual behavior of objects and the interaction among objects, but this is not of concern to us here.

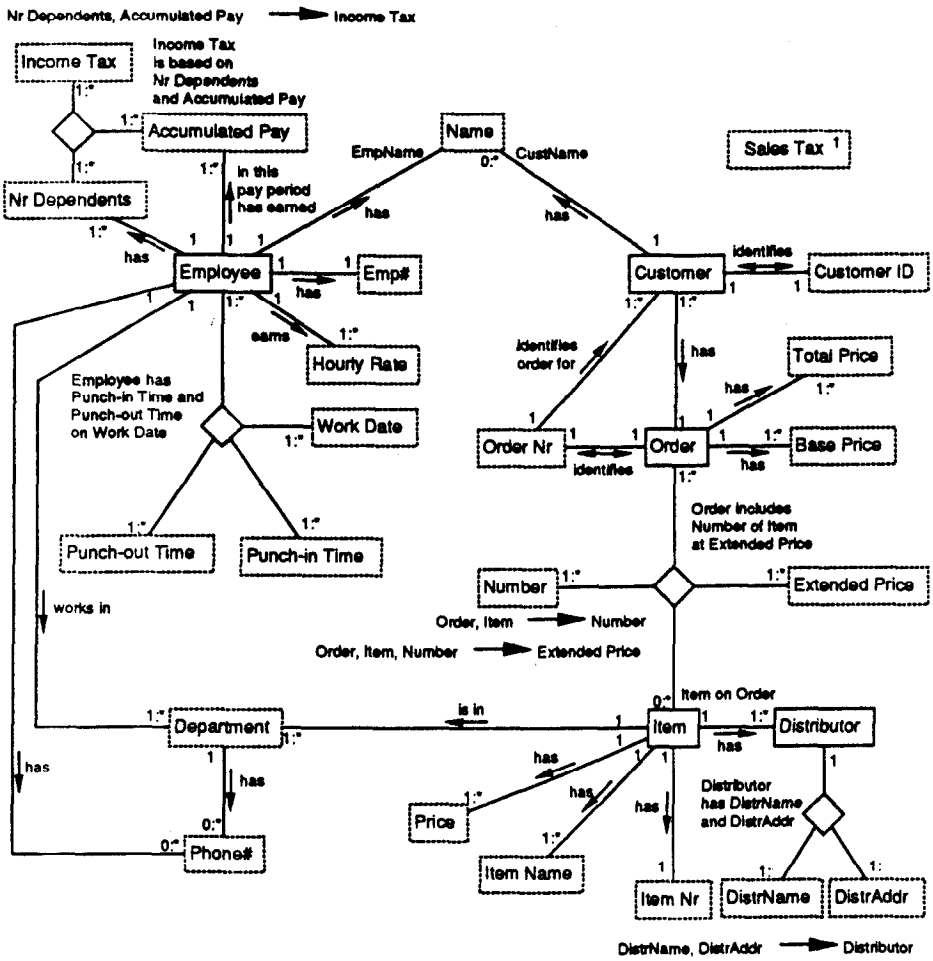

Figure 1. Sample OSA model instance.

OSA is similar to and can be classified as a member of the family of conceptual models commonly referred to as Object-Role Models (e.g., [9]). An important feature that distinguishes this family of models from the family of Entity-Relationship Models (e.g., [16]) is that there are only two types of basic sets — entity sets (or object sets as we call them in OSA) and relationship sets. This feature allows us to translate ORM model instances directly to hypergraphs, which are used so prevalently in

relational theory. Basically, each object set is a node and each relationship set is an edge. Since we can convert all model instances from the ORM family to the type of hypergraph view of a model instance we present here, all the remaining results follow directly. Thus, with minor variations, what we say and illustrate here for OSA also holds for all members of the ORM family of models.

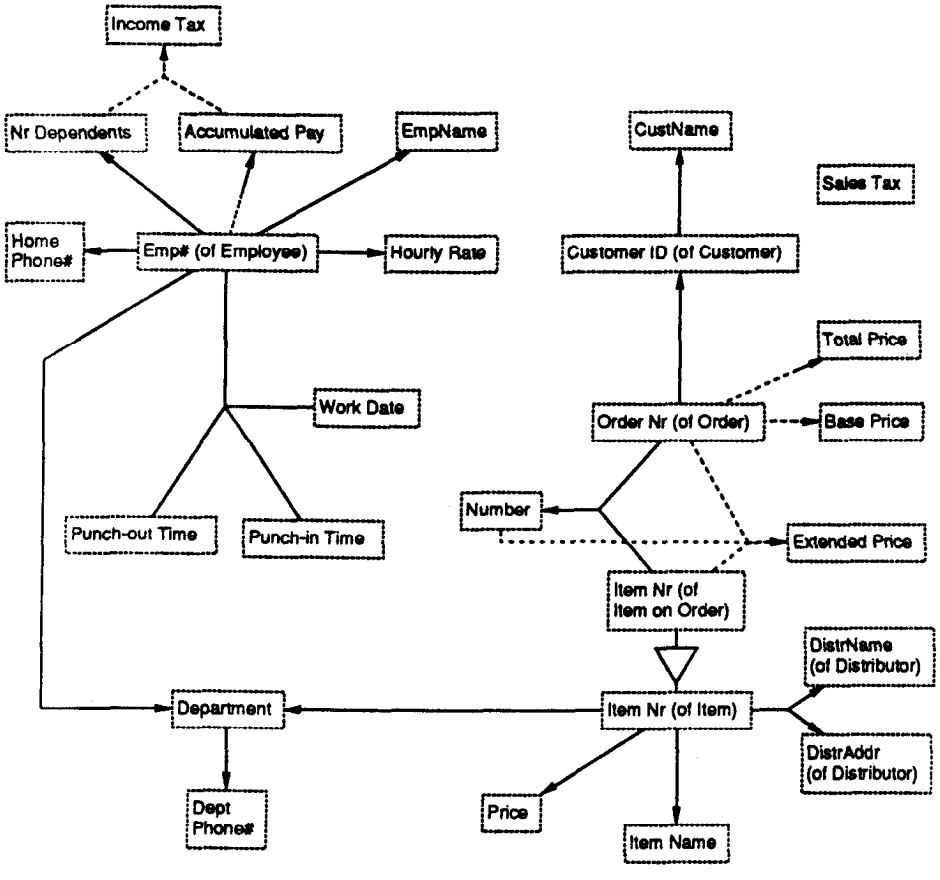

Figure 2. Design hypergraph for the OSA model instance in Figure 1.

For the OSA diagram in Figure 1, we can derive the hypergraph in Figure 2. Although relatively straightforward, the full details of the derivation are extensive. Moreover, we not only translate an OSA diagram to a hypergraph, but we also reduce the hypergraph through information-preserving and constraint-preserving transformations, for example, by removing redundant edges and edge components, consolidating connections, and converting n-ary edges to lower-degree edges. These transformations are similar to the transformations in [9]. We have reported the details of both converting an OSA diagram and reducing the resulting hypergraph in a previous ER conference [5]; here we only discuss the highlights of the translation of Figure 1 to Figure 2. To be specific in our discussion, we call the hypergraph we are deriving a *design hypergraph.*

A central idea in the initial creation of a design hypergraph is the extraction and representation of FDs. A 1 participation constraint yields an FD. For example, the 1 in the *Department has Phone#* relationship set in Figure 1 means that a department has only one phone. Thus, we have the FD *Department* → *Phone#*, which we can directly represent in a design hypergraph by a directed edge from *Department* to *Phone#*. When a relationship set has more than one 1 participation constraint, we obtain more than one FD. From the *Employee has Emp#* relationship set, for example, we obtain both *Employee* → *Emp#* and *Emp#* → *Employee*. Co-occurrence constraints directly represent FDs embedded in some *n*-ary edge of an OSA diagram. The *Order includes Number of Item at Extended Price* relationship set, for example, has two FDs: *Order, Item* → *Number* and *Order, Item, Number* → *Extended Price*. We put both of these directed edges in our design hypergraph.

One of the reduction transformations we can apply is called lexicalization, which lets us reduce a diagram by representing a non-lexical object set by a lexical object set with which the non-lexical object set has a one-to-one correspondence. We may for example represent employees by employee numbers. When we lexicalize in this way, we keep track of the original non-lexical object-set name in a parenthetical of-clause. Thus when we lexicalize the object set *Employee* by *Emp#*, we obtain *Emp#* (*of Employee*) as Figure 2 shows. In a variation of this lexicalization transformation, we have the pair *DistrName* (*of Distributor*) and *DistrAddr* (*of Distributor*) which, as a pair, are in a one-to-one correspondence with *Distributor*, and can thus replace the *Distributor* object set.

In another transformation, we can specifically introduce object sets for roles. Thus, for example, we have object-set rectangles for *EmpName* and *CustName*. Roles are specializations, which we denote by a triangle whose apex connects to a generalization and whose base connects to one or more specializations. In Figure 2, for example, *Item Nr* (*of Item on Order*) is a specialization of *Item Nr* (*of Item*). If the union of the specialization object sets is equal to the generalization object set and there are no other connections to the generalization object set, we can discard the generalization. Thus *Name* does not appear as an object set in Figure 2.

As we transform an OSA diagram to a design hypergraph, we discard redundant edges and edge components. We base these reductions on classical reductions such as right reduce and left reduce as discussed in [12]. For example, since *Order Nr* → *Order* and *Order* → *Customer*, *Order Nr* → *Customer* is redundant. Like [13], however, we do not assume that the universal relation assumption holds [10,17]. Thus, when we find a potential reduction based on an implied FD, we must check its meaning before we make the reduction. For example, we see from Figure 1 that we have the derived FDs *Employee* → *Department* and *Department* → *Phone#*, which imply the FD *Employee* → *Phone#*. However, the *Employee has Phone#* relationship set is redundant only if it has the same meaning as the join and projection over the relationship sets *Employee works in Department* and *Department has Phone#* — in other words, if the employee's department phone number is the same as the employee's phone number. This may be true, but is not true if, as we assume here, the *Employee has Phone#* relationship set represents an employee's home phone number. The resolution in this case is not to remove the relationship set, which in fact is not redundant, but to add the roles, *Dept Phone#* and *Home Phone#*. Then, similar to the transformation for names, we add object-set rectangles for specializations and then, since we

realize that the union of the two phone-number object sets constitutes all the phone numbers of interest in the application, we discard the *Phone#* object set.

Design hypergraphs that are fully reduced by reduction transformations such as these are called *reduced design hypergraphs*. In particular, a design hypergraph is *reduced* if it is non-redundant, right reduced, left reduced, JD-edge reduced, lexicalized, and minimally consolidated. We have illustrated several of these reductions here. In [5] we describe all of them.

Not yet discussed are the dashed edges in the design hypergraph in Figure 2. These represent computations. Given value(s) from the tail-side object set(s), there exists a function that computes a value for the head-side object set. We can, for example and as we suppose here, compute an *Income Tax* value given an *Accumulated Pay* value and a *Nr Dependents* value. Sometimes, we need additional information obtainable from the tail-side objects by a query to do the derivation. We compute an *Accumulated Pay* value for an employee, for example, based on the employee's hourly rate and the information in the employee's punch-in/punch-out work record.

As we explained in the introduction, we are interested in a restricted set of model instances and hypergraphs. We call the type of design hypergraph we seek a *restricted reduced design (RRD) hypergraph*. An *RRD hypergraph* is a reduced design hypergraph, which, after discarding any generalization/specialization edges and removing any computed object sets from their respective edges, satisfies the following two conditions: (1) the hypergraph is γ-acyclic [8], and (2) every edge of the hypergraph is in BCNF [4].

The notion of γ-acyclic has been defined elsewhere [8], but we discuss its definition here for the sake of completeness. We first define what it means for two nodes in a hypergraph to be connected. A *path* from node s to node t is a sequence of $k \geq 1$ edges $E_1, ..., E_k$ such that (1) s is in E_1, (2) t is in E_k, and (3) $E_i \cap E_{i+1}$ is nonempty if $1 \leq i < k$. Two nodes are *connected* if there is a path from one to the other. Similarly, two edges are connected if there is a path from one to the other. A set of nodes or edges is connected if every pair is connected. A *connected component* is a maximal connected set of edges. Two edges E and F are *incomparable* if $E \not\subseteq F$ and $F \not\subseteq E$. A hypergraph is *γ-cyclic* if it has a pair (E, F) of incomparable, nondisjoint edges such that in the hypergraph that results by removing $E \cap F$ from every edge, what is left of E is connected to what is left of F. A hypergraph is *γ-acyclic* if and only if it is not γ-cyclic. By Theorem 14 in [3], γ-cyclicity can be checked in polynomial time.

The definition for BCNF is well known. A scheme R is in *BCNF* if for every nontrivial FD $X \rightarrow Y$ (given or implied) such that $XY \subseteq R$, $X \rightarrow R$. We adapt this to a restricted design hypergraph by letting directed edges be the FDs and by considering each edge to be a scheme whose attributes are the object-set names of the object sets in the edge.

3. A Scheme-Generation Algorithm

We base our fundamental scheme-generation algorithm on Nested Normal Form (NNF), which we have recently defined [14]. NNF precisely characterizes potential redundancy with respect to a given set of FDs and multivalued dependencies (MVDs) for nested relation schemes that are consistent with the given FDs and MVDs.† We do

†Note, by the way, that in Section 2 we said nothing directly about MVDs. Indeed, with our procedure, a designer need

not reproduce the definition of NNF here nor the lengthy set of preparatory definitions. Instead, since a consistent nested relation scheme S is in NNF if and only if S has no potential redundancy, we explain what NNF is by discussing redundancy in nested relations. Along the way, we also introduce the vocabulary necessary for giving our algorithm for producing NNF schemes from RRD hypergraphs.

We can graphically represent a nested relation scheme by a tree, called a *scheme tree*. If U is a given set of attributes and T is a scheme tree constructed from the attributes in U, then the nodes in T are nonempty subsets of U. We denote the set of attributes that appear in T by $Aset(T)$. We further stipulate that the nodes in T are pairwise disjoint and that their union is $Aset(T)$. In a scheme tree T, if N is a node in T, $Ancestor(N)$ notationally denotes the union of attributes in all ancestors of N, including N. Figure 3 shows a scheme tree T along with $Aset(T)$ and $Ancestor(C)$. Figure 4 shows a possible nested relation for T. For NNF, we require PNF (Partition Normal Form) [15], so that in a nested relation there can never be distinct tuples that agree on the atomic attributes of either the nested relation itself or of any nested relation embedded within it. The nested relation in Figure 4 is in PNF.

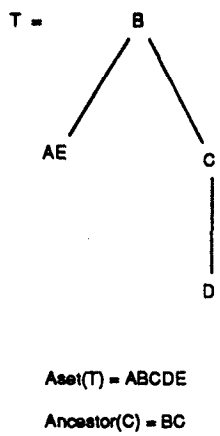

Figure 3. Scheme tree, Aset, and Ancestor examples.

The scheme in Figure 4 (equivalently the scheme tree in Figure 3) may or may not be in NNF depending on the given set of FDs and MVDs. As examples, it is in NNF for $\{B \twoheadrightarrow AE,\ C \rightarrow B,\ C \twoheadrightarrow D\}$, but not for $\{B \twoheadrightarrow AE,\ D \rightarrow C\}$ nor for $\{B \twoheadrightarrow AE, C \twoheadrightarrow D\}$ [14]. Note that the nested relation in Figure 4 is not valid with respect to the first set of dependencies since it violates $C \rightarrow B$ (4 is associated with both 1 and 9). However, it is valid with respect to $\{B \twoheadrightarrow AE,\ D \rightarrow C\}$ or $\{B \twoheadrightarrow AE,\ C \twoheadrightarrow D\}$ and both of these two sets of dependencies cause it to have redundant data values. A data value in a nested relation r is redundant if it is uniquely

never specify MVDs! They are implied and can be automatically extracted from an RRD hypergraph [12]. We observe, however, that a designer does need to be able to individually consider an n-ary relationship set for $n > 2$ and determine whether it can be losslessly reduced to a lower-degree relationship set. This is what we mean by JD-edge reduced, as mentioned above and as described in [5].

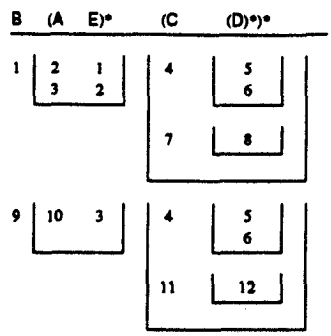

Figure 4. A sample nested relation.

determined by a constraint and the other data values in r. When the FD $D \rightarrow C$ holds for T, both 4's are individually redundant, for if either one were another value, $D \rightarrow C$ would not hold. Likewise, when the MVD $C \twoheadrightarrow D$ holds for T, both 5's and both 6's are individually redundant, for if any one were another value or were missing, $C \twoheadrightarrow D$ would not hold.

We are now ready to present our basic scheme-generation algorithm, which we give as Algorithm 1. In essence, the algorithm first creates a single-node scheme tree from a chosen edge in the given hypergraph. Since every edge is in BCNF, a single-node scheme tree created from an edge is in NNF vacuously. The algorithm continues to extend an existing scheme tree by adding other edges in the hypergraph as long as the scheme tree satisfies the conditions set forth. Intuitively, these conditions ensure that a scheme tree follows the natural hierarchical structure of the hypergraph — that is, we can continue adding as many one-one and one-many relationship edges as we wish, but we must stop as soon as we encounter many-one or many-many edges. Since each edge is included in only one tree, the algorithm runs in polynomial time with respect to the number of edges.

Theorem 1. Algorithm 1 generates NNF schemes.†

Theorem 2. For OSA model instances that correspond to RRD hypergraphs, the design process we have defined here is constraint-preserving and information-preserving.†

4. Application-Dependent Adjustments for Generated Schemes

In this section, we augment Algorithm 1 in several ways. The objective here is to show that we do not want to just blindly apply the algorithm for every application. There are application-dependent factors we need to consider as well. In the following subsections, we discuss some important additional factors that we should take into account when we generate nested relation schemes from a conceptual model.

† Proofs for these theorems are in a longer version of this paper available on the World Wide Web, URL: http://osm7.cs.byu.edu/HomePage.html.

Algorithm 1

input: An RRD hypergraph H. (Recall that this means that H is reduced, is γ-acyclic, has each edge in BCNF, has no generalization/specialization edge, and has no computed object set. The hypergraph in Figure 2 is an RRD hypergraph if we remove all computed edges — dashed edges — all the computed nodes — the object sets pointed to by dashed edges — and the generalization/specialization edge — the edge with a triangle.)

output: A set of scheme trees, each of which is in NNF with respect to the FDs and MVDs derived from H.

Repeat until all edges have been marked:

> Select an unmarked edge R in H and let the nodes in R be the set of nodes in the root of a new scheme tree T. Mark R as USED.
>
> While there is an unmarked edge R in H (i.e., not marked USED or DO_NOT_USE) such that $R \cap Aset(T) \neq \emptyset$, do:
>
> > If there is a node N in T such that $Ancestor(N) \subseteq R^+$ (the closure of R with respect to the FDs in the given RRD hypergraph) and $(R - Ancestor(N)) \cap Aset(T) = \emptyset$,
> >
> > > Extend T by adding a new node $N' = (R - Ancestor(N))$ as a child of N. If $Ancestor(N) \rightarrow A$ where $A \in N'$, move A up to N. If N' becomes empty, delete N' from T. Mark R as USED.
> >
> > Else T cannot be extended without violating NNF. Thus:
> >
> > > Mark R as DO_NOT_USE so that in this time through the while-loop, R will not be considered again.
>
> For each edge R that is marked DO_NOT_USE, unmark R.

For each object set S that is not involved in any edge in H, create a single-node scheme tree whose only node is S.

4.1. Chosen Roots

Algorithm 1 leaves an important question unanswered, namely, "What are the best roots for the nested relation schemes?" Since, in general, this depends on the semantics of the application, we cannot provide an answer in advance. However, we can provide some guidelines.

One of the purposes of having complex objects in OODBs is to have hierarchies of subobjects attached to these complex objects. Therefore, if we know which objects are important in the application, these object sets should be the roots of the nested relation schemes.

Sometimes, however, we may not know which objects are the most important, or after having generated a few trees, we may not know what is most important from

what is left. In this case, we should probably choose roots that will give large scheme trees since normally we want to cluster as much data together as we can. Algorithm 2 explains how to choose roots so that Algorithm 1 will yield large scheme trees.

Algorithm 2

input: The same input as Algorithm 1.

output: A set of large NNF nested relation schemes.

(1) For each edge E in the RRD hypergraph H, find the closure E^+ of E with respect to the FDs in H.

(2) List the edges in H in the order $E_1, ..., E_n$ where if E_i^+ is a proper subset of E_j^+, then $i < j$. (Note that there may be more than one ordering of edges. In this case, we arbitrarily choose one possible ordering.)

(3) Select the first edge in the list as the edge to start running Algorithm 1. As each edge E in H is marked USED in Algorithm 1, remove E from the list. For the next root, repeat this step by choosing the first edge left on the list, and so forth until the list is empty.

As an example, consider the RRD hypergraph in Figure 2, (i.e., hypergraph in Figure 2 after removing all computed edges, the computed nodes, and the generalization/specialization edge). Running Algorithm 2 yields the edge *Department has Dept Phone#*. When we then run Algorithm 1 starting with this edge, we obtain the scheme tree in Figure 5a. After eliminating all edges marked by Algorithm 1, we again run Algorithm 2, and this time we obtain the edge *Customer ID (of Customer) has Cust-Name*. When we then run Algorithm 1 starting with this edge, we obtain the scheme tree in Figure 5b. At this point, only the lone object set *Sales Tax* remains. Thus this object becomes a scheme by itself as Figure 5c shows.

4.2. Flexibility in Edge Configurations

Algorithm 1 is more tightly specified than it needs to be. In particular, the initial root need not be an entire edge. We can create a single-branch scheme tree T from the chosen edge so long as for every node N in T, if $Ancestor(N) \rightarrow Y$ holds for T with respect to the given MVDs and FDs, $Y \subseteq Ancestor(N)$. The reminder of the algorithm is the same, and we can still guarantee NNF. This gives us some additional flexibility, which we can use to adapt Algorithm 1 to specific applications.

As an example, consider the RRD hypergraph in Figure 6. The FDs and MVDs implied by this hypergraph are equivalent to $\{B \twoheadrightarrow AE,\ C \rightarrow B,\ C \twoheadrightarrow D\}$. We stated above that the scheme tree in Figure 3 is in NNF with respect to $\{B \twoheadrightarrow AE,\ C \rightarrow B, C \twoheadrightarrow D\}$. Using the current form of Algorithm 1, however, we cannot generate the scheme tree in Figure 3 because we cannot have a single node in the root. Observe also, that if we make ABE the root, then we cannot attach D, and if we make BC the root, we cannot attach either A or E. With our new modification here, however, we can derive the NNF scheme tree in Figure 3. Suppose we initially select the edge BC. Now, since $B^+ = B$ and $C^+ = BC$, if we create a single-branch scheme tree with B as the root and C as the child, then when $Ancestor(N) \rightarrow Y$ holds for T with

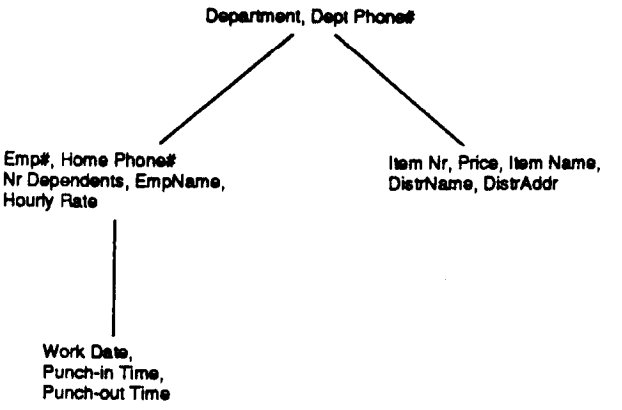

Figure 5a.

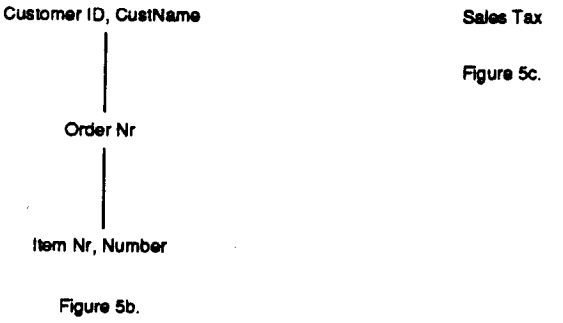

Figure 5b.

Sales Tax

Figure 5c.

Figure 5. Generated scheme trees.

respect to the given MVDs and FDs, $Y \subseteq Ancestor(N)$. Note, however, that we could not have C as the root and B as the child because then we would have $Ancestor(C) = C$, $C \rightarrow B$, but $B \not\subseteq C$. We can now complete Algorithm 1 as written to obtain the scheme tree in Figure 3.

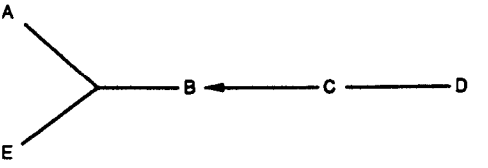

Figure 6. Design hypergraph for the scheme tree in Figure 3.

This idea of a single-branch scheme tree applies to subtrees as well as to the root. We can thus further modify the algorithm and gain even more flexibility. For example, in the scheme tree in Figure 3, we can either place A below E or E below A and still satisfy NNF.

Two heuristics guide the development of edge configurations within a scheme tree. (1) We should generally place attributes as high as possible in a scheme tree, and (2) we should generally include as many attributes as possible in a scheme tree. Unfortunately, these two guidelines conflict with one another. For our example here, if we attempt to place attributes as high as possible by choosing ABE as the root, there can be no child node, and thus we have not included as many attributes as possible in the scheme tree. Choosing either BC or CD as the root also does not work because either one prevents us from including both A and E as attributes in the scheme tree. We leave a resolution of this problem of finding best edge configurations and the potential problems of NP-completeness it may involve to future research.

4.3. Large Objects

Suppose our sample application in Figure 1 has pictures for items (images that can be displayed on-line). Each item is in some picture and many items can be in the same picture. Thus, we would have a new object set *Picture* and a new FD *Item* $\rightarrow$ *Picture* in the generated design hypergraph in Figure 2.

Now, when we run Algorithm 2 as we did before, *Picture* would be placed into the node that contains *Item Nr* in the scheme tree in Figure 5a. There is no NNF violation. However, the large size creates two problems that we can ignore for small atomic objects such as a name or a dollar amount, but should not ignore for large objects such as images. (1) Since many items may share the same picture, it would be unwise to replicate the picture many times for different items, and (2) the natural clustering would likely be lost because of interspersing large images among small objects. To solve these problems, we should replace *Picture* in the scheme tree by *Picture*$\uparrow$, which is a set of pointers or OIDs that reference the pictures in *Picture*, and we should make *Picture* a single-node scheme tree by itself.

4.4. No Updates

Theoretically, NNF is a necessary and sufficient condition to characterize data redundancy caused by FDs and MVDs in nested relations whose schemes are consistent with the given FDs and MVDs. Thus, if a nested relation scheme is in NNF, we can guarantee that there will be no instance for the scheme that will have data redundancy caused by FDs and MVDs. We should not forget, however, that our goal is not to reduce redundancy(!), but to make applications efficient. Reducing redundancy usually helps us achieve the goal of making applications efficient because it generally reduces both the time to process updates and check integrity constraints and the space to store the data. However, if there are no or only a few updates, and if we are not concerned about space for a particular application, data redundancy and the corresponding update anomalies are not a problem. In this situation, we may want to violate NNF and create larger nested relation schemes for the chosen roots.

One way to create larger schemes (which may not be in NNF) is to relax the condition of the if-clause in the while-loop of Algorithm 1. We may replace "If there is a node N in T such that $Ancestor(N) \subseteq R^+$ and $(R - Ancestor(N)) \cap Aset(T) = \emptyset$" by "If there is a node N in T such that $(R - Ancestor(N)) \cap Aset(T) = \emptyset$." We

keep everything else in Algorithm 1 the same. With this modification to Algorithm 1, it is possible to extend an existing scheme tree until we exhaust all the edges in a connected component of an RRD hypergraph. Of course, the designer may choose to stop earlier.

As an example in Figure 2, consider explicitly inheriting the object sets *Price* and *Item Name* from *Item Nr* (*of Item*) through the generalization/specialization to *Item Nr* (*of Item on Order*). Now, with the FDs *Item Nr* (*of Item on Order*) → *Price* and *Item Nr* (*of Item on Order*) → *Item Name*, if we run Algorithm 1 as we did before, we would *not* be able to include these two edges in Figure 5b. By relaxing the algorithm as just explained, however, we could include *Price* and *Item Name* in the leaf node in Figure 5b with *Item Nr* and *Number*. In our application, this would redundantly store the price and item name in a customer order of every item on order. For this application, however, this may be exactly what we want. We would not expect orders, once they are placed, to be updated. We would expect, however, that we may want to retrieve them to fill and ship orders and to check orders. This retrieval operation would be much faster with the redundant items stored than would be an operation that would have to join the nested relations for the scheme trees in Figures 5a and 5b so that the price and item name could be displayed with an order.

4.5. Computed Objects

Since computed object sets contain objects that are not stored, they cannot cause data redundancy and thus NNF should not apply to them. We can use our NNF algorithms, however, to properly group methods with schemes.

If S_c is a functionally computed object set that depends on object sets S_1, ..., S_n, $n \geq 1$, then there is an FD $S_1 \ldots S_n \rightarrow S_c$. It is possible that some of the S_i's are also computed object sets. For a generated scheme tree T, we can use the FDs that have computed object sets to determine which computed object sets are functionally determined by $Aset(T)$. If $Aset(T) \rightarrow S_c$ by using the RRD hypergraph FDs plus the FDs that have computed object sets that appear on either on the left- or right-hand sides, S_c becomes a method of T.

Note that it is also possible that a computed object set may not be associated with any generated scheme tree. For this case, we can generate a virtual scheme for the method — virtual in the sense that it does not have an extent. Thus, for the FD $S_1 \ldots S_n \rightarrow S_c$ we would have the flat virtual scheme $S_1 \ldots S_n, S_c$. We could also store other methods with this virtual scheme whose left- and right-hand side attributes are in the closure $(S_1 \ldots S_n S_c)^+$.

Applying these ideas to our RRD hypergraph in Figure 2, we can obtain the classes (scheme and method combinations) for our OODB in Figure 7. For illustration we have used an O_2-like syntax [1]. We have also added appropriate types for our attributes and methods. We could have introduced these types initially in our conceptual model and carried them through our discussion, but they would have been ignored until this point. Further, we have added the bulk type list. Buckets generated by our NNF algorithms are bulk types, which by default are sets of tuples, but can alternatively be organized as arrays or lists or as any other appropriate bulk type.

```
class SalesTax public: real end; name SalesTaxValue: SalesTax;

class Picture public: image end; name Pictures: set(Picture);

type EmployeeRecord: tuple(
    EmpNr: string,
    HomePhoneNr: string,
    NrDependents: integer,
    EmpName: string,
    HourlyRate: real,
    WorkRecord: list(tuple(
        WorkDate: Date,
        Punch-outTime: integer,
        Punch-inTime: integer))
    ); class Department public tuple(
    Department: string,
    DeptPhoneNr: string,
    Employee: set(EmployeeRecord),
    Item: set(tuple(
        ItemNr: string,
        Price: real,
        ItemName: string,
        Pict: Picture,
        DistrAddr: string,
        DistrName: string))
    ) method
    public AccumulatedPay(employeeRecord: EmployeeRecord): real,
    public IncomeTax(nrDependents: integer, accumulatedPay: real): real end; name Departments: set(Department);

type OrderRecord: tuple(
    Number: integer,
    ItemNr: string,
    ItemName: string,
    Price: real
    ); type CustomerOrder: tuple(
    OrderNr: string,
    IncludedItem: list(OrderRecord)
    ); class Customer: public tuple(
    CustomerID: string,
    CustName: string,
    Order: list(CustomerOrder)
    ) method
    public ExtendedPrice(orderRecord: OrderRecord): real,
    public BasePrice(customerOrder: CustomerOrder): real,
    public TotalPrice(basePrice: real, salesTax: SalesTax): real end; name Customers: set(Customer);
```

Figure 7. Classes for our application.

5. Concluding Remarks

We presented an approach for designing classes (schemes and methods) for OODBs. Our approach is based on OSA, a conceptual model in the ORM family. In our approach, we first transform an OSA model instance into a design hypergraph. We then reduce the hypergraph according to some model transformations and determine whether it is an RRD hypergraph, which we defined here as being reduced and γ-acyclic and as having each edge in BCNF. Algorithm 1, which we presented here, takes an RRD hypergraph as input and generates schemes that are in NNF. This guarantees that there is no potential redundancy for the generated nested relation schemes.

We explained that blindly applying Algorithm 1 for an application may not generate the schemes we want for our design. We therefore listed several ways to augment Algorithm 1 or adjust the results to better suit an application. These included (1) choosing the right roots, (2) using the allowed flexibility in edge configurations to

create better design schemes, (3) placing large objects in a scheme by themselves, (4) allowing for redundancy when the data for some schemes is expected to be static after it is established, and (5) determining the proper placement of methods.

References

1. F. Bancilhon, C. Delobel, and P. Kanellakis (eds.), *Building an Object-Oriented Database System: The Story of O_2*, Morgan Kaufmann Publishers, San Mateo, California, 1992.
2. C. Batini, S. Ceri, and S.B. Navathe, *Conceptual Database Design*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1992.
3. J. Biskup, "Database schema design theory: achievements and challenges," *Proceedings of the 6th International Conference, CISMOD'95. The proceedings appeared as Lecture Notes in Computer Science #1006*, pp. 14-44, Bombay, India, November 1995.
4. E.F. Codd, "Recent investigations in relational database systems," *Proceedings of the 1974 IFIP Conference*, pp. 1017-1021, 1974.
5. D.W. Embley and T.W. Ling, "Synergistic database design with an extended entity-relationship model," *Proceedings of the 8th International Conference on Entity-Relational Approach*, pp. 118-135, Toronto, Canada, October 18-20, 1989.
6. D.W. Embley, B.D. Kurtz, and S.N. Woodfield, *Object-oriented Systems Analysis: A Model-driven Approach*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.
7. R. Fagin, A.O. Mendelzon, and J.D. Ullman, "A simplified universal relation assumption and its properties," *ACM Transactions on Database Systems*, vol. 7, no. 3, pp. 343-360, September 1982.
8. R. Fagin, "Degrees of acyclicity for hypergraphs and relational database schemes," *Journal of the ACM*, vol. 30, no. 3, pp. 514-550, July 1983.
9. T.A. Halpin, *Conceptual Schema & Relational Database Design, 2nd Edition*, Prentice-Hall, Sydney, Australia, 1995.
10. W. Kent, "Consequences of assuming a universal relation," *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 539-556, December 1981.
11. W. Kim, "Editorial Directions," *ACM Transactions on Database Systems*, vol. 20, no. 3, pp. 237-238, September 1995.
12. D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, 1983.
13. V.M. Markowitz and A. Shoshani, "Representing extended entity-relationship structures in relational databases: a modular approach," *ACM Transactions on Database Systems*, vol. 17, no. 3, pp. 423-464, September 1992.
14. W.Y. Mok, Y.K. Ng, and D.W. Embley, "A normal form for precisely characterizing redundancy in nested relations," *ACM Transactions on Database Systems*, vol. 21, no. 1, pp. 77-106, March 1996.

15. M.A. Roth, H.F. Korth, and A. Silberschatz, "Extended algebra and calculus for nested relational databases," *ACM Transactions on Database Systems*, vol. 13, no. 4, pp. 389-417, December 1988.

16. T.J. Teorey, *Database Modeling & Design: The Fundamental Principles, 2nd Edition*, Morgan Kaufmann Publishers, San Francisco, California, 1994.

17. J.D. Ullman, "The U.R. strikes back," *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 10-22, Los Angeles, California, March 1982.