GENERATING OBJECT-ORIENTED VIEWS FROM AN ER-BASED CONCEPTUAL SCHEMA

Tok-Wang LING
Department of Information Systems
and Computer Science
National University of Singapore

Pit-Koon TEO
Data Resources
DBS Bank

Ling-Ling YAN
Institute of Systems Science
National University of Singapore

ABSTRACT

A 3-level schema architecture is proposed to address some inadequacies in recent object-oriented data models. The conceptual schema is based on a normal form object-oriented entity relationship (OOER) diagram. A set of mapping rules is defined that allows OO external schemas, based on the O2 data model, to be generated from the conceptual schema. Similar mapping rules can be defined to generate external schemas for other OO data models. Other features of this model include the incorporation of derived attributes, deductive rules, triggers and methods.

1 INTRODUCTION

There exist many inadequacies[16] in the object-oriented (OO) paradigm, eg. lack of a formal foundation, general disagreement about OO concepts, a navigational, as opposed to a declarative, interface, lack of a standard declarative query language etc. Some of these inadequacies have been addressed but many issues remain unresolved. In this paper, a three-level schema architecture, comprising an external schema level, a conceptual schema level and an internal schema level, is proposed to address a set of OO data modelling issues, viz. (1) the problematic way in which m-n, n-ary and recursive relationships are handled in the OO approach, (2) the lack of general and flexible support for external schemas or views, and (3) the inability to differentiate good OO schema design from bad. Our approach leverages research efforts in Entity Relationship (ER) data modelling [5].

The OO data modelling issues are resolved at the conceptual schema level. To represent the conceptual schema, the notion of a ***normal form OOER diagram*** is introduced. A normal form OOER diagram has all the structural features of a classic ER diagram[5,14]. It goes beyond the classic ER diagram in two areas. First, a behavioural dimension is included that incorporates the notions of methods, derived attributes, deductive rules and triggers. This yields an OOER diagram. Second, the structural quality of the OOER diagram is improved by applying the techniques of [14] to convert the OOER diagram into a normal form OOER diagram. Informally, a normal form OOER diagram is a normal form ER diagram [14] extended with the notions of methods, derived attributes, deductive rules and triggers.

Each OO database schema (based on any of the proposed OO data models [3,6,7,17 etc]) is treated as an external schema that is generated from the conceptual schema by applying a set of mapping rules. Distinct sets of mapping rules are needed to handle different OO data models. Each OO database schema is, therefore, a view of the conceptual schema. User applications (OO or otherwise) can then be written based on the OO external schemas that are generated from the conceptual schema. The advantages of the OO approach are therefore preserved at the external schema level. To illustrate our approach, a set of mapping rules is defined in this paper to generate, as examples, O2 class definitions from a normal form OOER diagram. Similar mapping rules can be defined for other OO data models.

Section 2 provides some background information. Section 3 compares the OO and ER data models. Section 4 presents some salient features of our approach. In Section 5, a set of mapping rules for deriving O2 external schemas from a normal form OOER diagram is given. Section 6 discusses some related work. Section 7 concludes.

2 BACKGROUND

The OO concepts presented in this section are based primarily on the O2[6] data model, supplemented with examples from ORION[3] and POSTGRES[17].

A ***conceptual entity*** is anything that exists and can be distinctly identified. For example, a person, an employee, a student etc are conceptual entities. In an OO system, all conceptual entities are modelled as ***objects.*** An object has structural properties defined by a finite set of attributes and behavioural properties defined by a finite set of methods. Each object is associated with a logical non-reusable and unique ***object identifier*** (OID) [11]. The OID of an object is independent of the values of its attributes.

All objects with the same set of attributes and methods are grouped into a ***class,*** and form instances of that class. We distinguish between a lexical class and a non-lexical class. A ***lexical class*** contains objects that can be directly represented by their values, eg. integer, string etc. A ***non-lexical class*** contains objects, each of which is represented by a set of attributes and methods. Instances of a non-lexical class are referred to by their OIDs. Examples of non-lexical classes include PERSON, EMPLOYEE, SUPPLIER etc.

The domain of an attribute of a non-lexical class A can be one of the following:

(Case a) a lexical class such as integer, string etc. An attribute with this domain is called a ***data-valued*** attribute[8].

(Case b) a non-lexical class B. An attribute with this domain is called an ***entity-valued*** attribute [8].

Note the recursive nature of this definition. There is an implicit ***binary relationship*** between A and B. The value of the attribute is the OID of an instance of B, which must exist before it can be assigned to the attribute. This provides ***referential integrity.*** A special case exists in which the class B is, in fact, A. This represents a ***cyclic definition*** in the OO model. Cyclic definitions are used to specify recursive relationships such as a part and its subparts, a course and its prerequisite courses etc. A cyclic definition needs not be direct; for instance, a class A may reference a class B1 which in turn reference class B2 and so on until class Bn is referenced by class Bn-1 for some n. Then Bn can reference A, thus completing the cycle. In some OO systems, eg. ORION, the relationship between A and B can be given semantics such as IS-PART-OF, in which case A is a composite object [12] comprising B. Orion also supports the concept of an ***existentially-dependent object,*** in which the existence of the object depends on the existence of its parent object. The deletion of an object triggers a cascading delete of all objects that are existentially dependent on the deleted object. This adds to the integrity features of ORION.

(Case c) a set, set(E) where E is either a lexical class or a non-lexical class. An attribute with this domain is called a ***set-valued*** attribute.

If E is lexical, values from E are stored in the set. If E is a non-lexical class, members of the set can either be an instance of E or its subclasses. In this case, the set comprises instances from possibly heterogeneous classes. Only the OID of each instance is stored in the set. In some systems, eg. O2, both sets and lists are supported. While it is correct to differentiate between a list and a set, this paper treats a set and a list as equivalent.

(Case d) a query type whose values range over the set of possible queries coded in a query language. An attribute with this domain is called a ***query-valued*** attribute.

The value of a query-valued attribute is the result of the query, which is a set of objects satisfying the query condition. An example of a system that allows attributes of type query is POSTGRES.

(Case e) a tuple type. An attribute with this domain is called a ***tuple-valued*** attribute.

This represents an aggregation of attributes of the tuple type, which is treated as a composite attribute of A. An attribute of the tuple type can be a data-valued, entity-valued, set-valued, query-valued or tuple-valued attribute.

A ***type constructor*** is a mechanism for building new domains. For example, the set and tuple constructs described above are used to create domains for set-valued and tuple-valued attributes respectively. A ***complex object*** is built using type constructors such as sets, tuples, lists and nested combinations of these. The structure of objects (complex or otherwise) is hierarchical.

The **class hierarchy** is one of the most important features of the OO paradigm. It provides a taxonomy of classes that are related through the ISA relationship. Given two classes X and Y, X ISA Y implies that each instance of X is also an instance of Y. This has a ***set inclusion*** semantics. We call X a ***subclass*** of Y and we call Y a ***superclass*** of X. A class hierarchy provides an inheritance mechanism which allows a class to inherit properties (attributes/methods) from its superclass(es), if any.

3 COMPARISON BETWEEN OO AND ER MODELS

The ER model was first introduced in [5]. It incorporates the concepts of entities, relationships and attributes and allows the structural representation of a database to be captured in an ER diagram. The ER diagram has proven to be a useful database design tool. Good normal form relations can be generated from a normal form ER diagram [14] and directly implemented on any relational DBMS. Figure 1 shows an ER diagram which will be used to illustrate the various concepts of this paper.

We believe that all the structural properties of the OO approach can be derived (generated) from the ER model. For instance, an OO class hierarchy can be directly represented using the ER relationship type 'ISA'. Figure 1 shows a representation of a class hierarchy rooted at the PART entity type. The composite entity type VEHICLE with component parts ENGINE and DRIVETRAIN is represented using the IS-PART-OF relationship type. Weak entities (eg. DEPENDENTS) correspond to existentially dependent objects [12] which cannot exist independently of their parent objects. Table 1 provides a summary of the comparison between these two models. Note that there is no equivalent concept of methods in the ER model.

The ER model supports functional dependencies, multi-valued dependencies and cardinality constraints among entities, relationships and attributes. These concepts are not inherent in the OO approach and have to be explicitly enforced using, for example, programming. Each regular entity or relationship type is associated with an identifier which is dependent on the values of the primary key attributes. This is different from the OID of an object. The OO approach provides only one type of ISA relationship. In contrast, the ER approach provides special ISA relationship types[14] such as UNION, INTERSECTION, DECOMPOSE etc. which consider the set properties between entity types.

4 RESOLVING OO DATA MODELLING ISSUES

The OO approach suffers from a number of OO data modelling inadequacies, viz. (1) the problematic way in which m-n, n-ary and recursive relationships are handled in the OO approach, (2) the lack of general and flexible support for external schemas or views, and (3) the inability to judge the quality of an OO database schema. We address these issues by adopting the following steps:

(1) Represent a database schema by an ER diagram.

(2) Imbue the ER diagram of Step(1) with methods, derived attributes, deductive rules and triggers to obtain an OOER diagram.

(3) Convert the OOER diagram of Step(2) into a normal form OOER diagram, using the techniques introduced in [14]. The normal form OOER diagram is used as the conceptual schema.

(4) Apply a set of mapping rules to generate OO external schemas from the normal form OOER diagram.

Step 1 ensures that relationships such as m-n, n-ary, recursive, ISA, IS-PART-OF etc are ***explicitly*** represented. In the OO approach, to represent m-n, n-ary and recursive relationship types, or to answer symmetric queries, some data redundancy may need to be introduced, eg. through the use of inverse pointers. This redundancy is not controlled and introduces problems similar to those of the hierarchical model. By providing strong support for association among entity types, it eliminates the need to maintain inverse pointers among related entities, because their semantic association is now conceptually represented by a relationship. Another advantage of supporting relationships directly is that the cardinalities of entity types participating in a relationship type are displayable on the ER diagram. This is a cardinality constraint which is absent from most OO data models.

Step 2 provides a behavioural dimension to an ER diagram to obtain an OOER diagram. (See later subsections)

Step 3 converts an OOER diagram into a normal form OOER diagram, using the techniques of [14]. A normal form OOER diagram is considered 'good' structurally since it satisfies the criteria for a normal form ER diagram[14]. Note that a normal form ER diagram is based on a firm theoretical foundation provided by normalisation/dependency theories.

Step 4 allows the generation of OO external schemas from the conceptual schema. This addresses the problem of a lack of general and flexible support for views or external schemas in the OO approach. In Section 5, a set of mapping rules is provided to generate O2 external schemas. Similar mapping rules can be defined for other OO data models.

The rest of this section describes the behavioural component of our model, viz. methods, triggers, derived

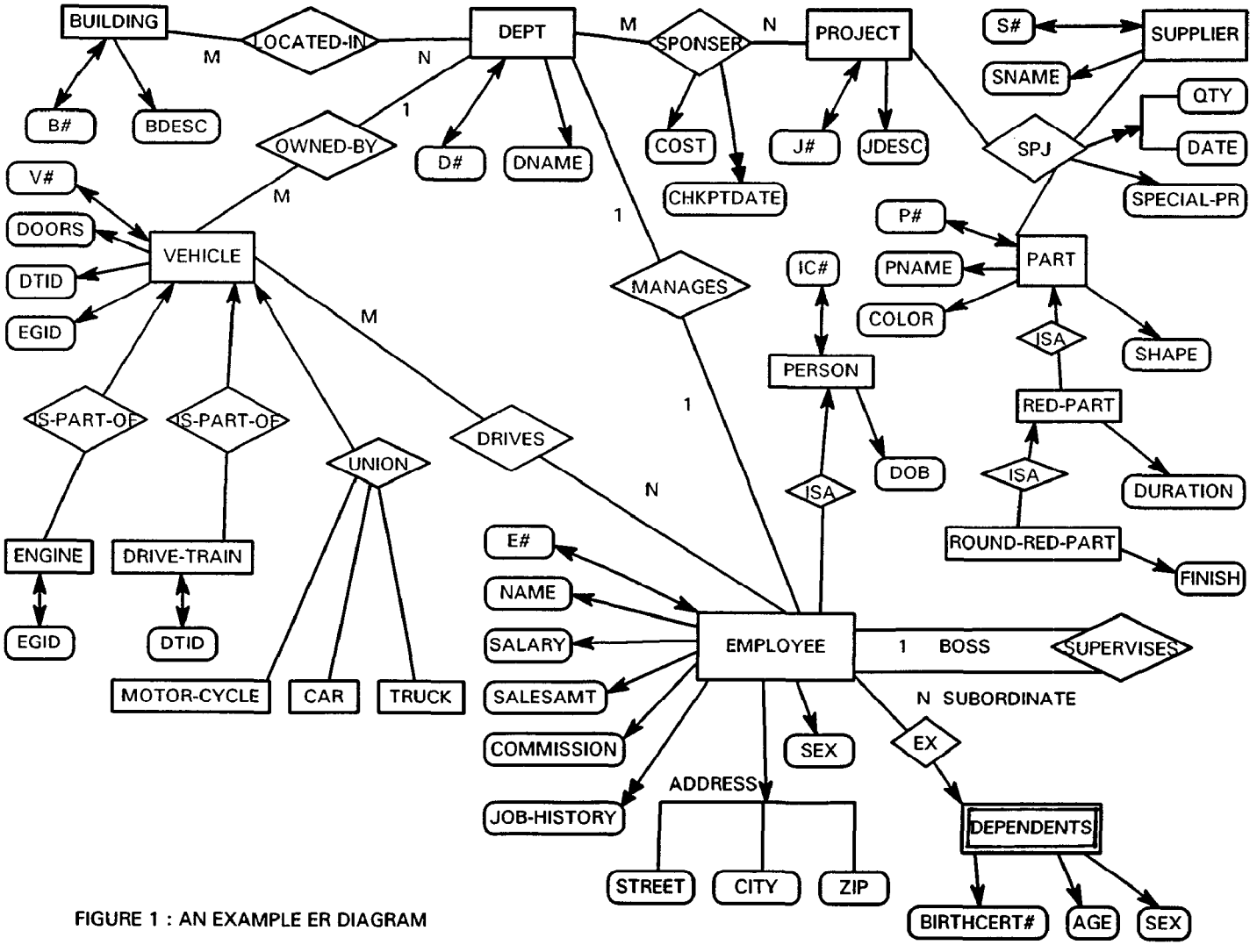

FIGURE 1 : AN EXAMPLE ER DIAGRAM

Object-Oriented Data Model	Entity Relationship Model
Class	Entity Type or Relationship Type or Domain of attributes
An instance of a class	Entity or relationship or attribute value
Class has attributes describing its structural properties	Attributes
Domain of an attribute of a class can itself be a class. Implicitly, a binary relationship exists between these two classes.	The relationship type between two or more entity types is made explicit.
Existentially Dependent object	Weak Entity (with associated EX or ID relationship)
Methods	No equivalent concept
Composite object	Entity types connected by IS-PART-OF relationship type
Use of cyclic definition (see text) to represent recursive relationships (eg. part-subpart, course-prerequisites etc)	Natural way of representing recursive relationships. Role names are used to distinguish entity types participating in recursive relationships.
Class hierarchy	ISA, INTERSECTION constructs
Set-valued attributes	Multi-valued attributes
Tuple-valued attributes	Composite attributes

Table 1: Comparison of ER and OO Data Models

attributes and deductive rules.

4.1 Adding Methods to the OOER Model

Methods are categorized into *system generated methods, database administrator (DBA) definable methods* and *programmer-definable methods.* System generated methods exist in both the conceptual and external schema levels and include code to browse instances of a class, or to retrieve or update an object's attribute values. These methods are more efficiently optimised and they free programmers from writing trivial code. Updates to identifiers of entity types are handled independently of system generated methods. A DBA must treat these updates as special one-time operation and ensure that database consistency is maintained after the updates are done. DBA-definable methods are accessible by all authorised users of the database and can be defined at both the conceptual and external schema levels. For instance, the DBA can define methods to provide generic computations such as calculation of interest, commission based on generic formulas, etc. Programmer-definable methods, eg. hire an employee, print an entity etc are application specific and not generally accessible. The proper place for programmer-definable methods is at the external, rather than conceptual, schema level. This is no different from conventional practice, in which a database programmer writes application programs based on an external schema available to him.

4.2 Triggers.

Triggers can be system generated to enforce simple integrity constraints, or explicitly coded by DBA and programmers at both the conceptual and external schema levels. For example, consider Figure 1, in which projects sponsered by departments are shown by the relationship SPONSER. A simple trigger to enforce the referential constraint that exists between PROJECT and SPONSER is:

AFTER DELETE ON PROJECT
REFERENCING OLD AS OLD_PROJECT
DELETE FROM SPONSER S WHERE S.P# = OLD_PROJECT.P#.

Note that it is difficult to enforce more complicated constraints using triggers, eg. suppliers who do not supply all red parts must supply a blue part or a green part with quantity greater than 100.

4.3 Derived Attributes

Derived attributes can be obtained from other attributes, based on some derivation rules. For example, in Figure 1, the 'Commission' attribute of the EMPLOYEE entity type is derivable from the 'Salary' and 'SalesAmt' attributes by some computation rule. Derived attributes can be treated as being computed by a method in the OO sense. Derived attributes are explicitly coded by either the DBA (conceptual and external schema levels) or programmers (at external schema level).

4.4 Deductive Rules

Simple deductive rules can be defined for classes by predicating on attributes of a class. For example, a RED_PART deductive rule can be defined as:

RED_PART(P#, Pname, Price) :- PART(P#, Pname, 'red', Price).

The RED_PART deductive rule is characterised by the fact that the positions of the variables (ie P#, Pname, Price) in each clause are important. However, the positions of attributes in an ER diagram representation of an entity are immaterial. In [19], we propose a notation to address this difference. Example 4.4.1 illustrates this notation.

Example 4.4.1 : The above RED_PART deductive rule can be either be rewritten as:

RED_PART(X:P#, Y:Pname, Z:Price) :-
PART(X:P#, Y:Pname, 'red':Color, Z:Price).

OR

RED_PART(Z:Price, X:P#, Y:Pname) :-
PART(Y:Pname,X:P#,'red':Color,Z:Price).

In the above representations, a variable is placed to the left of ':' and a corresponding attribute name or role name is placed to the right of ':'. This notation provides for more expressive power in the sense that each variable in a

predicate is tagged with an attribute or role name. It ensures that the positions of variables in a predicate no longer matter; two attributes can be interchanged without affecting the meaning of the rule. Another advantage of this notation is that it reduces the need for anonymous variables.

5 MAPPING RULES

In this section, a set of mapping rules which allows O2 external schemas to be constructed from a normal form OOER diagram is provided. The justifications and motivations for these rules are given through several examples. Mapping rules for other OO data models can be similarly obtained. Appendix I gives an example O2 schema which will be used to illustrate the mapping rules.

5.1 Generic Rules

(Rule 1) Each (regular or weak) entity type can be mapped to an external class. Only regular relationship types can be mapped to an external class.

Two points are to be noted: (1) It is possible to have class specifications for regular relationship types (eg. SPJ_VIEW1 in Appendix I is based on the SPJ relationship type in Figure 1). As noted in [4], it is necessary to provide class specifications for ternary and higher degree relationship types if it is required to model such relationship types in an external schema. (2) A weak entity type A may be mapped to an external class A' without also mapping the entity type on whose existence A depends. See Rule 6 for a mapping rule that applies to weak entity types.

(Rule 2) Not all entity/relationship types have a corresponding external class in the external schema.

This satisfies the maxim that an external schema is an abstraction of the conceptual schema which emphasises relevant details while suppressing unwanted details.

(Rule 3) An entity/relationship type of the conceptual schema may have more than one corresponding external class specification.

An entity/relationship type can therefore be viewed in different ways. In Appendix I, external classes EMPVIEW1 and EMPVIEW2 have the same underlying entity type EMPLOYEE. Classes SPJ_VIEW1 and SPJ_VIEW2 are different views of the same underlying relationship type SPJ.

(Rule 4) An external class A' whose underlying entity type is A must include at least one key of A as its attribute. This is an *entity-preserving* condition, which ensures the updatability of A'. However, an external class A' generated from a relationship type may not possess a key of the underlying relationship type as its attribute.

This rule ensures that apparently frivolous and meaningless views (eg. a class comprising only attributes sex and salary generated from the entity type EMPLOYEE in Appendix I) cannot be generated from any entity type. There is no equivalent 'relationship-preserving' condition because, in some cases, it may be semantically meaningful to generate an external class from a relationship set without retaining the key of the relationship set. For instance, given the SPJ relationship set, it is meaningful to generate a view SP which contains information about the parts supplied by suppliers.

In Appendix I, key attributes of external classes based on entity types are indicated using comments. Both SPJ_VIEW1 and SPJ_VIEW2 are created from the relationship set SPJ but SPJ_VIEW1 retains the key attributes of SPJ while SPJ_VIEW2 does not.

(Rule 5) Consider a regular entity type A of the conceptual schema. An external class A' whose underlying entity type is A can base its data-valued attributes on some or all the single-valued attributes of A, or from a single component of a composite attribute of A. A' can construct each of its tuple-valued attributes, if any, from some single-valued attributes of A, or from a composite attribute, or part of a composite attribute, of A. Each multi-valued attribute of A is mapped, if needed, to a set-valued attribute of A'.

In Appendix I, EMPVIEW1 is derived using Rule 5.

(Rule 6) Consider a regular entity type B connected to a weak entity type A by an existence dependent or an identifier dependent relationship type R. B can easily be mapped to an external class, say, B' by using Rule 5. The mapping of A to an external class, say A', depends on R. If R is an existence dependent relationship type, then A' can be derived from A in exactly the same manner as any regular entity type by applying Rule 5. If R is an identifier dependent relationship type, then A' must include the identifier of B as one of its attributes. In each of these cases, if it is desired to impose the existence dependency constraint of A' on B', then B' can specify either an entity-valued attribute whose domain is A' (if the cardinality of A in R is 1) or a set-valued attribute whose domain is set(A') (if the cardinality of A in R is n, $n > 1$). The attribute is given a name which reflects the role of A' in B'.

In Appendix I, classes EMPVIEW1 and DEPENDENTS_VIEW are defined from the regular entity type EMPLOYEE and weak entity type DEPENDENTS respectively. If DEPENDENTS does not have any identifier attribute (thereby changing the relationship type into an identifier dependent relationship type), then DEPENDENTS_VIEW must include E# (the identifier of EMPLOYEE) as an attribute. In this case, inserts to DEPENDENTS_VIEW are possible provided that the EMPLOYEE entity identified by E# already exists. Other updating constraints exist. For example, when an instance of EMPVIEW1 is deleted, the underlying EMPLOYEE tuple will be deleted. This triggers a corresponding deletion of all DEPENDENTS tuples whose existence depends on the deleted EMPLOYEE tuple. The set of instances of DEPENDENTS_VIEW that forms the value of the attribute 'Support' of the deleted EMPVIEW1 instance automatically disappears.

(Rule 7) Recursive relationships are mapped to a cyclic definition (Section 2) in the external schema. Cycles in the conceptual OOER diagram (eg. the cycle involving

VEHICLE, EMPLOYEE and DEPARTMENT of Figure 1) can also be mapped to a cyclic definition.

In Appendix I, VEHICLE_VIEW, EMPVIEW2 and DEPTVIEW1 represent a cyclic definition based on the cycle involving VEHICLE, EMPLOYEE and DEPARTMENT. Further, EMPVIEW2 represents a view of the EMPLOYEE entity type that considers the recursive relationship SUPERVISES in the conceptual schema. A cyclic definition enables the computation of the transitive closure of a class. For example, from EMPVIEW2, it is possible to compute all the bosses (supervisors) of a particular employee.

(Rule 8) Consider an n-ary relationship type R involving n entity types E1,..,En, which have external classes $E'_1,..,E'_n$ derived using Rule 5 respectively. R has associated single valued attributes $Y_1,..,Y_m$ $(m>0)$ of respective domains $D_1,..,D_m$ (not necessarily distinct) and multivalued attributes $Z_1,..,Z_p$ $(p>0)$ of respective domains $F_1,..,F_p$ (not necessarily distinct). An external class E'_i $(0<i\leq n)$ can have a set-valued attribute named Rext with domain constructed using the set and tuple constructors on different permutations of $E'_1,..,E'_n$, except E'_i and the attributes of R. The domain of Rext is therefore a nested set (of tuples) construct. Each entity type $E'_1..E'_n$, except E'_i, is the domain of one, and only one, component of a tuple in the nested set construct. The deepest set includes the attributes of R. Further, to allow for different restructuring, any two entity types E'_u and E'_v $(u<>i, v<>i)$ can be interchanged positionally. Formally, Rext has a domain :

$$\begin{aligned}&\text{set(tuple}(Q_{(L0)1}:E'_{(L0)1},..,Q_{(L0)q0}:E'_{(L0)q0},\\&\quad P_1:\text{set(tuple}(Q_{(L1)1}:E'_{(L1)1},..,Q_{(L1)q1}:E'_{(L1)q1},..,\\&\quad P_k:\text{set(tuple}(Q_{(Lk)1}:E'_{(Lk)1},..,Q_{(Lk)qk}:E'_{(Lk)qk},\\&\quad Y_{i1}:D_{i1},..,Y_{im}:D_{im}Z_{j1}:\text{set}(F_{j1}),..,Z_{jp}:\text{set}(F_{jp})))..)))),\end{aligned}$$

where

(1) the domain is a set of sets nested k times, q0+q1+..+ qk = n-1

(2) $(L0)_1,..,(L0)_{q0},(L1)_1,..,(L1)_{q1},..,(Lk)_1,..,(Lk)_{qk}$ are different permutations of 1,..,n, except i, i1,..,im and j1,..,jp are different permutations of 1,..,m, and 1,..,p respectively,

(3) $Q_{(L0)q0'},Q_{(L1)q1'},..,Q_{(Lk)qk'}$, $P_1,..,P_k$ are attribute names,

(4) $E'_{(L0)q0'},E'_{(L1)q1'},..,E'_{(Lk)qk'}$ are external classes whose underlying entity types are $E_{(L0)q0'},E_{(L1)q1'},..,E_{(Lk)qk'}$ respectively, and

(5) for all qi, i=0,..,k, qi > 0, ie at each level of nesting, there must be at least one entity-valued attribute.

(6) when n=2 (ie a binary relationship), k = 0 and therefore Rext has a domain set(tuple($Q_{(L0)1}:E'_{(L0)1},Y_{i1}:D_{i1},..,Y_{im}:D_{im},Z_{j1}:\text{set}(F_{j1}),..,Z_{jp}:\text{set}(F_{jp})$)). If the cardinality of the link between R and $E_{(L0)1}$ is 1 (ie R is m-1), then Rext is a tuple-valued attribute; if R is m-1 and all attributes of R are discarded, then Rext is an entity-valued attribute.

In Appendix I, SUPPLIER_VIEW1, SUPPLIER_VIEW2 and SUPPLIER_VIEW3 represent three views created from the ternary relationship set SPJ. DEPTVIEW1 is an external class, based on DEPT in Figure 1, that considers the binary relationship sets SPONSERS and MANAGES. If the cardinalities of the links between R and its entity types are equal to 1, then external schemas can be generated in some preferred ways. For example, if the cardinality of the link from SPJ to PROJECT in Figure 1 is 1, then SUPPLIER_VIEW1 is preferred over SUPPLIER_VIEW2 and SUPPLIER_VIEW3, because it is semantically more correct. As another example, the attribute 'Manager' of DEPTVIEW1 is an entity-valued attribute of domain EMPVIEW2. This is because the cardinality of the link between EMPLOYEE and MANAGES in Figure 1 is 1, and no attributes are associated with MANAGES. If MANAGES has associated attributes, then the attribute 'Manager' of DEPTVIEW1 is tuple-valued.

(Rule 9) Given a relationship type R, we can construct a new relationship type from R by :

(a) projecting out some of the associated attributes of R,

(b) projecting out part of a composite attribute of R,

(c) restricting the values of some attribute of R, and/or

(d) removing the participation of some entity types of R.

The previous mapping rules can then be applied to the constructed relationship type to derive external classes.

Note that the cardinalities of attributes of the constructed relationship type may not be the same as their original cardinalities. For example, SPJ_VIEW2 is derived from a relationship type constructed from SPJ by removing the participation of PROJECT. Because the cardinality of the link from SPJ to PROJECT is m, the cardinality of the attribute 'SpecialPrice' of SPJ is changed from single-valued to multi-valued when PROJECT is removed. However, if the cardinality of the link from SPJ to PROJECT is 1, then the cardinality of 'SpecialPrice' will not be changed.

5.2 Rules Arising From The Inheritance Lattice

(Rule 10) Given an external class specification A with an attribute A' of domain B, the value of the attribute A' can be drawn from B or any of the subclasses of B.

(Rule 11) The structure of an inheritance lattice can be viewed in different ways, as long as the inclusion dependency between a class and its subclasses is maintained.

In Appendix I, ROUND_RED_PART_VIEW is a direct subclass of PART_VIEW. This is a consequence of the fact that the ISA relationship type is transitive.

(Rule 12) If two entity types A and B are connected by an ISA relationship such that A ISA B, then A inherits and may possibly override attributes/methods of B. An external entity type A' can then be derived from A using the relevant mapping rules, eg. rule 5.

EMPVIEW1 shows a view of EMPLOYEE in which the attribute DOB is inherited from PERSON entity type.

(Rule 13) Given two entity types A and B such that A ISA B, and B is participating in a relationship type R, we can construct a virtual relationship type R' from R in the ER diagram by replacing the participant B in R by A. We can then use R' for constructing external classes.

Rules 12 and 13 apply to special relationships such as

UNION, INTERSECT, DECOMPOSE. For example, if entity types A, B1,..,Bn are connected by the special relationship UNION such that A = UNION (B1,B2,..,Bn), then Bi ISA A for all i=1,2,..,n. External class specifications can then be constructed using rules 12 and 13.

(Rule 14) We may specify the value range of any attribute of an external class. This derives a subclass of the external class based on attribute values.

For example, a class PART with attribute colour can have a subclass RED-PART which is the set of parts with colour='red'. By predicating on the values of other attributes, other subclasses can be defined, eg. in Figure 1, ROUND_RED_PART is predicated on the 'shape' attribute in RED_PART.

5.3 Rules Arising From The Reachability Principle.

An entity type is ***reachable*** from another entity type if both entity types are connected, directly or indirectly, by some relationship type(s) and/or entity type(s). For example, in Figure 1, the entity type PROJECT is reachable from the entity type BUILDING through the relationship types LOCATED_IN and SPONSER and the entity type DEPT.

(Rule 15) An external entity type may include attributes from a reachable entity type. The cardinalities of such attributes in the external entity type may not be the same as their original cardinalities in the conceptual schema.

In general, an external entity type may not include attributes of entity types which are not related according to the reachability principle. The rules of Sections 5.1 and 5.2 can then be applied to derive external class specifications from the external entity type obtained. In Appendix I, DEPTVIEW2 is an example class, based on DEPT, that includes attributes from the reachable entity types BUILDING and PROJECT.

5.4 Rules Relating To Methods

(Rule 16) An external class can contain programmer-defined methods which derive new attributes (properties) for the class.

For example, in Appendix I, a method can be defined for EMPVIEW1 that computes the age of an instance of EMPVIEW1, based on the attribute DOB. Another method can be defined for EMPVIEW2 that allows all the superiors of an instance of EMPVIEW2 to be computed.

(Rule 17) A method defined at the conceptual schema level can be used at the external schema level if it does not refer to resources which are not present at the external schema level or which are protected from access.

For example, EMPVIEW1 can include a method (defined at the conceptual schema level) that derives a count of the number of children for an employee instance, if the execution right on that method has been granted by the DBA to users.

6 RELATED WORK

Most OO database systems (DBMSs) do not fit into the three level schema architecture as spelled out in the ANSI/X3/SPARC[2]proposal for DBMSs. These OODBMSs typically provide a large grained conceptual schema with minimum or no facility for defining views or external schemas. Several proposals have been made to incorporate some view mechanisms in OODBMSs [1,9,13,18]. The works of [9,18] focussed on defining multiple interfaces (views) to an object class. Each interface or views defines a set of methods and different views of an object class may share methods. Generally, these proposals do not sufficiently address the need for a flexible and declarative view mechanism such as that found in relational DBMSs. In [1,13], a query based view mechanism is used to derive subclasses from superclasses. [1] treats views as queries and uses the view mechanism to define virtual classes, which are structured into inheritance lattice. The behavioural aspects of the views defined are generated automatically by the system. Furthermore, such views are not updatable. In [13], updates apply only to views that are based on a join of the primary keys of the base tables. The views are non-recursive. Many of the view mechanisms proposed are defined on OO schemas which themselves suffer from several limitations (eg. problematic way of representing m-n, n-ary (n>2), recursive relationships, presence of redundancy etc). This seriously hampers the usefulness of these view mechanisms.

7 CONCLUSION

The OO paradigm is not a panacea that will address all the data modelling problems of complex applications. This paper has proposed a framework to address several OO data modelling issues, viz. (1) the problematic way in which m-n, n-ary and recursive relationships are handled in the OO approach, (2) the lack of general and flexible support for external schemas or views, and (3) the inability to differentiate good OO schema design from bad. Our approach allows OO external schemas to be generated from a conceptual schema which is represented by a normal form OOER diagram. A set of mapping rules is defined to derive O2 external schemas from the conceptual schema. A normal form OOER diagram incorporates several OO concepts, eg. methods, inheritance, existentially dependent objects etc. The model also include triggers for simple constraint enforcements, derived attributes and deductive rules.

As future work, we are currently investigating the updatability of the external schemas.

REFERENCES

[1] S. Abiteboul, A. Bonner, Objects and Views, Proc ACM SIGMOD Int'l Conf. on Mgmt of Data, Denver, Colorado, 1991.

[2] ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report, FDT (ACM Sigmod bulletin) Vol 7 No 2,1975.

[3] J. Banerjee et al, Data Model Issues for Object-Oriented Applications, ACM Trans. Office Information Syst., Vol 5 No 1, Jan 87, pp. 3-26.

[4] R. Cattell, Object Data Management: Object-Oriented and Extended Relational Database Systems, Addison Wesley, 1991.

[5] P.P. Chen, The Entity-Relationship Model: Toward a Unified View of Data, ACM Trans. on Database Systems, Vol 1, No 1, 1976.

[6] O. Deux et al, The Story of O2, IEEE Trans. on Knowledge and Data Engineering, Vol 2, No 1, Mar 1990, pp. 91-108.

[7] D. Fishman et al, IRIS: An object oriented database management system, ACM Trans. Office Information Syst., Vol 5 No 1, Jan 87, pp. 48-69.

[8] B. Fritchman, R. Guck, D. Jogannathan, J. Thompson, D. Tolbert, SIM : Design and Implementation of a Semantic Database System, Proc ACM Sigmod Conf. 1989, pp 46-55.

[9] B.Hailpern, H.Ossher, Extending Objects to Support Multiple Interfaces and Access Controls, IEEE Trans. Software Engineering,Vol 16,No 11,Nov 1990.

[10] A. Keller, Choosing a View Update Translator by Dialog at View Definition Time, Computer, Jan 1986.

[11] S. Khoshafian, G. Copeland, Object Identity, Proc OOPSLA, 86, Portland, Oregon, 1986.

[12] W. Kim, An Introduction to Object-Oriented Databases, MIT Press, 1990.

[13] ChenHo Kung, Object Subclass Hierarchy in SQL: A Simple Approach, CACM 33, 7, 1990.

[14] T.W. Ling, A Normal Form for Entity-Relationship Diagrams, Proc. 4th Int'l Conf. on Entity-Relationship Approach, 1985, pg 24-35.

[15] T.W. Ling, A Three Level Schema Architecture ER-Based Data Base Management System, Proc. 6th Int'l Conf. on Entity Relationship Approach, 1987, pp 181-196.

[16] T.W. Ling, P.K. Teo, Towards Resolving Inadequacies in Object-Oriented Data Models, Submitted for Publication, 1992.

[17] L. Rowe and M. Stonebraker, The Postgres Data Model, in The Postgres Papers, Memo No UCB/ERL M86/85 Jun 87 (Revised), U.C.(Berkeley).

[18] J. Shilling, P. Sweeney, Three Steps to Views: Extending the Object-Oriented Paradigm, Proc OOPSLA 89.

[19] P.K. Teo, An Object-Oriented Entity Relationship Data Model, MSc Thesis, National University of Singapore, 1992.

Appendix I : Example O2 Schema Based on Figure 1

```
add Class EMPVIEW1 type tuple (
        E#              : string;           /* key */
        Name            : string;
        Address         : tuple(street:string, Zip:string);
        Job_History     : set (string);
        Support         : set(DEPENDENTS_VIEW);
        DOB             : string;)          /* from PERSON */

add Class EMPVIEW2 type tuple (
        E#              : string;           /* key */
        Name            : string;
        Boss            : EMPVIEW2;         /* cyclic definition */
        Subordinate     : set(EMPVIEW2);
        Drives          : VEHICLE_VIEW;)    /* defined below */
add Class DEPENDENTS_VIEW type tuple (
        BirthCert#      : string;           /* key */
        Sex             : char;
        Age             : integer;)
add Class DEPTVIEW1 type tuple (
        D#              : string;           /* key */
        Dname           : string;
        Manager         : EMPVIEW2;
        Sponser         : set(tuple(J:PROJECTVIEW,
                                Cost:float,ChkPtDte:set(string))))
add Class DEPTVIEW2 type tuple (
        D#              : string;           /* key */
        Dname           : string;
        Projects        : set(string);      /* project descriptions */
        Location        : string;)          /* from Bdesc attribute */
add Class VEHICLE_VIEW type tuple (
        V#              : string;           /* key */
        OwnedBy         : DEPTVIEW1;)
add Class SUPPLIER_VIEW1 type tuple (
        S#              : string;           /* key */
        Sname           : string;
        PJ : set(tuple(P:PARTVIEW, J:PROJECTVIEW,
                QtyDate:set(tuple(Qty:integer,date:string)))))
add Class SUPPLIER_VIEW2 type tuple (
        S#              : string;           /* key */
        Sname           : string;
        PJ : set(tuple(P:PARTVIEW, ProjQtyDate:set(tuple
                (J:PROJECTVIEW, Qty:integer, date:string)))))
add Class SUPPLIER_VIEW3 type tuple (
        S#              : string;           /* key */
        Sname           : string;
        JP : set(tuple(J:PROJECTVIEW, PartQtyDate:set(tuple
                (P:PARTVIEW, Qty:integer, date:string)))))
add Class PARTVIEW type tuple (
        P#              : string;           /* key */
        Pname           : string;
        Color           : string;
        Shape           : string;)
Class ROUND_RED_PART_VIEW inherits PARTVIEW
        type finish:string;
add Class PROJECTVIEW type tuple (
        J#              : string;           /* key */
        Jdesc           : string;)
add Class SPJ_VIEW1 type tuple (
        /* key is composite comprising S, P and J */
        S               : SUPPLIER_VIEW1;
        P               : PARTVIEW;
        J               : PROJECT_VIEW;
        QtyDate         : set(tuple (Qty:integer, Date:string));
        SpecialPrice    : integer;)
add Class SPJ_VIEW2 type tuple (
        S       : SUPPLIER_VIEW1;   /* no underlying key */
        P       : PARTVIEW;
        SpecialPrice : set(integer));
```