Incomplete Objects—A Data Model for Design and Planning Applications

Tomasz Imielinski
Rutgers University
New Brunswick, NJ 08903

Shamim Naqvi
BELLCORE
445 South Street
Morristown, NJ 07960-1910

Kumar Vadaparty
Rutgers University
New Brunswick, NJ 08903

Abstract

We are motivated by applications within design, planning and scheduling areas, where current research appears to be focused on syntactic issues of performance and volume. We take a more semantic view of applications within these areas and discover several useful functionalities that are poorly supported. For example, facilities for handling incomplete specifications are quite inadequate. We introduce a notion of OR-objects and show that it captures incomplete specifications naturally. In particular, a database with OR-objects represents a set of possible worlds, e.g., a world for each design or schedule, and queries can either be evaluated in the "interpretations" of the database, or in the database itself. We formalize these notions of interpretations and hypothetical queries in an object-oriented setting, and provide a complexity characterization for our queries.

1 Introduction

Non-traditional applications in disparate domains such as CAD, CASE, decision support systems, and office information systems have led database research and system building to new object oriented and/or complex-object data models and systems ([J 82], [VERSO 87], [LDM 87], [AK 89], [Ban 87], [LRV 88]).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-425-2/91/0005/0288...$1.50

These models have the power to model the structure of many complex domains. However, we observe that these models lack some important functionalities required by design/planning applications. These functionalities include (i) Coexistence of objects in different stages of completion (e.g., complete and incomplete designs); (ii) Distinction between representation and interpretation of data; (iii) Ability to ask and evaluate hypothetical queries; (iv) Encapsulation of non-determinism and a notion of choice in the data model. The need for them was brought to our attention by engineering applications within Bellcore. In particular, a system called Sub-Assemblies Model (SAM) has been developed in Bellcore that stores templates of designs for telecommunications equipment. See Figure 1 for a typical template showing alternative ways of designing a component of a DLC channel. The template shows a hierarchical part-subpart structure in which the edges carry extra semantics: an edge labeled R implies that the subpart is mandatory, OM indicates optional mix (pick 0, 1, or many), etc. All together, we have six different labels. The engineer in order to complete a design of a part, retrieves the corresponding template for that part, and records design choices while following the template. At any given time, different DLC channels will be in different stages of completion—some will be finished, while others may be at some intermediate stage. In principle, all such designs may share some resources, e.g., inventory of all the available parts.

Below we illustrate each of the functionalities listed above in terms of the SAM application:

1. Incomplete and complete designs of DLC channels will have to coexist in the database.

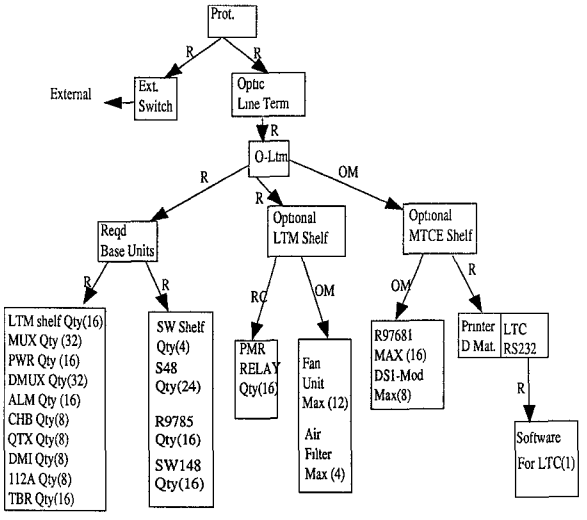

Figure 1: A Design Template from SAM.

2. We need to distinguish between two levels of querying: *implementational* and *structural*. At the implementational level, an incomplete design corresponds to the set of all its possible final designs, while at the structural level it corresponds to the design itself. This distinction is best understood in terms of queries: The query (Q_1) "If we choose part #2342, do all possible final designs cost more than \$100," addresses implementations, while the query (Q_2) "What are the choices for a DLC channel," addresses the structure of the design.

3. The query "If we choose part #2342, do all designs cost more than \$100," above illustrates the need for hypothetical queries.

4. Since queries addressing complete designs occur frequently in these applications, encapsulating the notion of *implementations* into the query language will help in lowering application development time.

However, neither SAM nor any other system/model supports the above functionalities. We observe in the next section that these functionalities are not unique to this particular example, but occur in every design/planning/scheduling application. *It is our goal in this paper to identify the essential formal notions responsible for providing these functionalities, and incorporate these in the existing formalisms.*

Incompleteness and non-determinism in the design specifications are nicely represented by AND/OR trees with the OR-nodes representing the choices in the design process. Whereas a number of existing formalisms developed for design/planning applications can represent and reason with conjunctive information (AND-nodes) by using sets, surprisingly, none of them support disjunctive information (OR-nodes). Our model enables storing disjunctive information *directly* by introducing a new type called *OR-Type*, whose instances are called *OR-Objects*. Thus, we capture incomplete design specifications by nested sets and OR-Objects (i.e., AND/OR-trees) in a natural way. Note that AND/OR-trees are generic in the sense that one can use them to represent not only incomplete designs but also incomplete schedules/plans/etc.

Our formalism uses the notion of *interpretation* of an AND/OR-tree to capture the idea of implementations. An interpretation of an AND/OR-tree is obtained by making choices at every OR-node. *Thus, an interpretation can represent a final design or a final plan or a final schedule etc.* Thus *interpretation* is also a generic notion. Now, one can direct *implementational queries* such as Q_1 above to the interpretations of the AND/OR-trees and *structural* queries such as Q_2 above to the AND/OR-trees themselves. One uses the notion of *views* to direct queries this way. All these notions are formalized in this paper.

The main contributions of our work are as follows.

1. We develop a *simple* data model that enables storing of AND/OR trees. This model is obtained by extending [AK 89] with a new type called *OR-Type*. As a part of our data model, we introduce a new type of object-identifiers (oids) called *mutable object identifiers* which generalize the marked nulls of [IL 84].

2. We formalize a notion of *interpretations*. An interpretation is obtained by making a choice at every OR-node of an AND/OR representation. The presence of mutable oids makes the notion of interpretations non-trivial. In fact, there exist databases for which this notion is not well-defined. We define *OR-Cycles* and show that *the notion of interpretations is well defined for every instance of a schema S iff S has no OR-Cycle.*

3. We formalize a notion of *views* which enables querying at different levels: *representational* or *interpretational* or *a combination*. Finally we show that the

data complexity ([CH 82], [Vardi 82]) of our query language is CoNP-Complete and identify a class of queries, *SOME-queries*, which have PTIME data complexity.

2 Motivating Examples

Our work is motivated by applications in manufacturing, design and scheduling. As mentioned in section 1 these applications require (i) Storing AND/OR trees (ii) Addressing implementations or interpretations of AND/OR trees (iii) Specifying the level of a query – *representational* or *interpretational* or *a combination*. In this section we illustrate these functionalities.

We introduce a complex type, called *OR-Objects*, to represent OR-Nodes. Sets represent AND-Nodes, nested AND/OR objects represent AND/OR trees, and finally *AND/OR-Tables* or simply *tables* store these AND/OR-Objects (AND/OR-Trees). Informally, a table is a nested relation with a pre-specified type.

Example 2.1 (Interpretations)
The table $TRAVEL$ below shows a travel schedule for various employees of a company. The first entry in this table is $\langle John, i\rangle$ with $v(i) = [CA, NJ]$ whose meaning is: *"John" plans to go to a place which is yet not known (denoted by i) but it will be either NJ or CA.* The second entry is $\langle Jack, j\rangle$ with $v(j) = [NJ, MA]$ with a similar meaning. The third entry shows a complete specification. An interpretation of $TRAVEL$ is obtained by replacing every oid i' in $TRAVEL$ by a member of $v(i')$. Different interpretations are obtained by choosing different members for this replacement.

Employee	Place
John	i
Jack	j
James	CA

Employee	Place
John	NJ
Jack	MA
James	CA

TRAVEL **Its interpretation**

Note that each interpretation of $TRAVEL$ corresponds to a schedule of the employees. Define a schedule to be valid only if no two employees in a certain binary relation $CONFLICT$ go to the same place. Then the query (Q_1) "Is every schedule invalid?" is at *interpretational level* while (Q_2) "Are there two persons belonging to $CONFLICT$ such that their sets of choices intersect?" is at *structural level.* Note that Q_2 views OR-Objects as *sets* of choices —not as disjunctions. ■

Note that if we had more tables, we could get queries that have a component of the interpretational level, and a component of the structural level. Thus, the above example motivates the need for having a facility that specifies the level of a query. We introduce *views* in section 5 which achieve precisely this goal. Finally, one can show by reducing the graph-colorability problem ([GJ 79]) that the data complexity of query $Q1$ is complete for Co-NP; hence, $Q1$ is not expressible in relational languages without recourse to host language facilities (unless $P = CoNP$). Thus, an essential contribution of our work is the encapsulation of the notion of non-determinism/incompleteness into the data model enabling queries to address implementations directly and thus decreasing application development time.

3 The Data Model

Our presentation follows closely the development of [AK 89]. Our model has two kinds of atomic constituents: *object identifiers (oids)* and *constants.* An oid is an element chosen from a denumerable set $\mathcal{O}$ and a constant is an element chosen from the denumerable set $\mathcal{D}$. Complex-objects, called o-values, are formed from these atomic constituents by forming finite sets, OR-Objects and tuples. An OR-object of $o_1, \ldots, o_k$ is denoted by $[o_1, \ldots, o_k]$ and tuples and sets are represented in the usual way. Both sets and OR-Objects are duplicate free, un-ordered collections; the semantics of sets is well known. OR-Objects will be interpreted disjunctively, as we shall explain shortly.

Like any other object oriented formalism, ours also uses oids to "label" or "name" o-values. This "naming" or "labeling" is specified by a value-specifying function which is any 1-1 function mapping from the oids $\mathcal{O}$ to the set of all o-values $\mathcal{V}$. Thus if v is value-specification function and $v(i) = \langle a, b\rangle$, we say that the "value of the oid i is the tuple $\langle a, b\rangle$", or that "i labels the tuple $\langle a, b\rangle$".

In our data model, the OR-Objects are to be interpreted disjunctively. However, there seem to be two possible "interpretations" of the OR-Objects. Thus, if $\langle john, i\rangle$ is a

tuple, with $v(i) = [a, b]$, it can be interpreted as a tuple $\langle john, i\rangle$ with a disjunction in the value as $(v(i) = a) \bigvee (v(i) = b)$ or, it can be interpreted as a disjunction in the tuple as $\langle john, a\rangle \bigvee \langle john, b\rangle$. In the former, the oid i is thought of as a name for the object whose value is not known yet, while in the latter i is viewed as a "place-holder". The former is called a *persistent oid* while the latter is referred to as a *mutable oid*. Both kinds appear to be useful in modelling user's intuitions.

One may argue, however, that we do not need mutable oids, if we encode the choice within the tuples as in $\langle John, [ca, nj]\rangle$ and interpret the OR-Object $[ca, nj]$ as $ca \bigvee nj$. This meets the desired interpretation of the above example. Thus, under this scheme any OR-Object $[a, b]$ will be interpreted as $a \bigvee b$, and every oid is persistent. This approach, however, can not account for shared incomplete information. Suppose that two teachers *Tomasz* and *Diane* want to teach the same course which will be either *cs*540 or *cs*541. This is a classic example of structure-sharing in object-oriented databases, the only difference being that the shared structure is an incomplete object. Hence, this structure needs to have a name (oid), labeling it. We can encode the above as $Teach(Tomasz, i)$ and $Teach(Diane, i)$ with $v(i) = [cs540, cs541]$. Clearly, the interpretations of the above tuple require us to treat i as a place-holder. Once a decision has been made, i gets replaced by the appropriate course number. Thus, i has to be mutable, and not persistent. It must be noted that *mutable oids* are not a completely new notion, either. The marked nulls of [IL 84] are mutable oids. One difference is that a marked null can be replaced by any domain constant, while a mutable oid i can be replaced only by a member in the OR-Object $v(i)$. Thus our mutable oids generalize the marked nulls of [IL 84].

Two further conclusions may be drawn from the above discussion.

- Mutable oids label only OR-Objects because they are meant to capture the intuition of "place-holders".
- There is no need for OR-Objects to appear without an oid labeling them because $\langle John, [a, b]\rangle$ can be re-written as $\langle John, i\rangle$ with $v(i) = [a, b]$. Thus, without loss of generality, we require that an OR-Type be always associated with a "pointer". The definition given below enforces this.

Now we introduce, *tables*, the structures in which we store the o-values. A *table* is a nested relation with a pre-specified type. It also stores oids, and OR-Objects unlike the usual nested relations. The *type of a table* B, denoted by $T(B)$, gives the type of any member of B. The following definition formalizes the language of type expressions. Note the explicit use of "pointer-types" (similar to programming languages).

Definition 3.1 (Types(B)) *Let* $\mathbf{B} \subset \mathcal{B}$ *be a finite set of table names. Every* $B \in \mathbf{B}$ *is a type expression, and so is* $\mathcal{D}$. *Furthermore, if* τ *is a type expression, then so are* $\{\tau\}$, $\uparrow[\tau]$ *and* $\uparrow\tau$; *if* $\tau_1, ..., \tau_k$ *are type expressions for some* $k \geq 0$, *so are* $\langle \tau_1, ..., \tau_k\rangle$ *and* $\tau_1 \bigvee ... \bigvee \tau_k$.

A type expression of the form $\uparrow \tau$ *is called a pointer type and an expression of the form* $\uparrow [\tau]$ *is called an OR-pointer type.*

The following observations are important.

- Since table names can appear in the definition of types of other tables, "recursive" definitions such as $T(P) = \langle \mathcal{D}, \uparrow P\rangle$ are possible.
- There exist certain recursive types that are unintuitive, e.g., $T(P) = \{P\}$. We disallow such structures; the mechanisms for this are not germane to our presentation here and are left for a fuller presentation elsewhere.

The next question is how to specify the contents of a table and verify if they conform to the appropriate type. The contents of any table are specified by the function π, called *o-assignment*. To perform static type-checking, we need to know the oids that are associated with various "pointer-types". We use π to define this as well. For example , let $T(B) = \langle \mathcal{D}, \uparrow \mathcal{D}\rangle$, $\pi(B) = \{ \langle a, i_1\rangle, \langle John, i_2\rangle\}$ and $\pi(\uparrow \mathcal{D}) = \{i_1, i_2\}$. Clearly, the contents of B conform to $T(B)$, because, a and $John$ are known to be constants. The following defines π.

Definition 3.2 *An o-assignment* π *is defined for a finite set* $\mathbf{B} \subseteq \mathcal{B}$ *of tables and the type-expressions* $Types(\mathbf{B})$ *and is such that (i) for each pointer type expression* $\uparrow \tau \in Types(\mathbf{B})$, $\pi(\uparrow \tau)$ *is a finite set of oids; (ii) for each table* $B \in \mathbf{B}$, $\pi(B)$ *is a finite set of o-values. Furthermore,*

- *For any* $\uparrow \tau \neq \uparrow \tau'$, $\pi(\uparrow \tau) \cap \pi(\uparrow \tau') = \emptyset$.

- *If i is a mutable oid, and $i \in \pi(\uparrow \tau)$ then $\uparrow\tau = \uparrow[\tau']$ for some τ'*

The first condition is similar to the disjoint oid-assignment of [AK 89]. We can relax this if we consider a type hierarchy. The second condition assures us that every mutable oid has an OR-Object as a value. This is to meet the intuition that mutable oids act as *place-holders*. Such an oid when interpreted must vanish. If however, its value is a set/tuple, there is no way it can vanish. Hence the value of a mutable oid must be an OR-Object.

Let $\mathbf{B}$ be a set of table names, and π an o-assignment for $\mathbf{B}$. Then the domain, $|\tau|_\pi$, of $\tau \in \text{Types}(\mathbf{B})$ under π is the set of all objects of type τ. It is defined as in [AK 89] except for OR-types for whom the definition proceeds as follows:

- $|\uparrow \tau|_\pi = \pi(\uparrow \tau)$ if τ is not of the type $[\tau']$ for any τ'
- $|\uparrow [\tau]|_\pi = \pi(\uparrow [\tau]) \bigcup |\uparrow \tau|_\pi \bigcup |\tau|_\pi$

Note that the domain of $\uparrow [\tau]$ is a union of the domains of τ and $\uparrow \tau$ and the oids assigned to $\uparrow [\tau]$. This is done to capture the meaning of OR-Types: *They are inherently a union of more than one type.* Consider Example 2.1 wherein $TRAVEL) = \langle Name : \mathcal{D}, Choice :\uparrow [\mathcal{D}]\rangle$. The above definition allows the following three kinds of tuples in $TRAVEL$:

1. $\langle John, i\rangle$ with $v(i) = [ca, nj]$.
2. $\langle John, my\text{-}place\rangle$ with $v(my\text{-}place) = ca$.
3. $\langle John, ca\rangle$.

Thus, the above definition of *domain* of an OR-Type allows the presence of data of multiple types in an attribute declared to be of OR-Type. The definition captures the desired intention that *an OR-Type specifies the maximum allowable incompleteness—not that every member of the OR-Type must be incomplete.* In this respect, an OR-Type differs from a set-type. If an attribute is declared to be of set-type, every member of that attribute must be a set.

Now, we formalize the notion of a schema and an instance of a schema. A *schema* specifies a set of table names, and a typing function that gives the types of these tables, while an instance of a schema S gives the contents of the tables such that they conform to the types specified in S.

A database schema S is a pair $\langle \mathbf{B},\ T\rangle$, where $\mathbf{B} \subset \mathcal{B}$, is a finite set of tables and $T : \mathbf{B} \rightarrow Types(\mathbf{B})$ is a typing function.

A database instance I_S of a schema $S = \langle \mathbf{B},\ T\rangle$ is a tuple given by $I_S = \langle\ \pi, v, \alpha\ \rangle$ where π is an o-assignment for $\mathbf{B}$, v is a value-specification function that maps each oid appearing in I_S to an o-value, and α maps each oid to p or m depending on whether it is a persistent or a mutable oid, such that

1. For every $B \in \mathbf{B}$, $\pi(B) \subseteq |T(B)|_\pi$.
2. For each oid i of type $\uparrow \tau$, $v(i) \in |\tau|_\pi$.

Example 3.1 (A schema and an Instance)
Let $S_1 = \langle \mathbf{B}, T\rangle$ be a schema such that $\mathbf{B} = \{TEACH\}$, and $T(TEACH) = \langle Tname : D, Course :\uparrow [D]\rangle$. Note the different levels of incompleteness in the information present in the attribute $Course$. This is facilitated by the OR-Types.

Let $I = \langle \pi, v, \alpha\rangle$ be an instance of S_1 such that

- $\pi(\uparrow D) = \{i_1, i_2, i_4\}$ and $\pi(\uparrow [D]) = \{i_3, i_4\}$
- $\pi(TEACH) = \{\langle Liza, i_1\rangle,\ \langle Eliza, i_3\rangle,\ \langle Beth, CS541\rangle,\ \langle Elizabeth, i_4\rangle\}$
- $\alpha(i_1) = \alpha(i_2) = \alpha(i_4) = p$ and $\alpha(i_3) = m$
- $v(i_1) = CS540$, $v(i_2) = CS542$, $v(i_3) = [CS509, CS506]$ $v(i_4) = [CS539, CS536]$

4 Interpretations

We identify interpretations of a database with the models of the theory obtained by treating OR-Objects as disjunctions. However, as pointed out in section 3, this depends on whether the OR-Object is the value of a mutable oid or a persistent oid. For example, let $v(i) = [a, b]$. Then if i is mutable, *an interpretation* is obtained by *replacing* i with a in the entire database while if i is persistent, *an interpretation* is obtained by putting $v(i) = a$. Other interpretations are obtained similarly. We refer to both the actions as *interpreting an oid*, the former as *interpreting by replacing* and the latter as *interpreting by refining.*

Definition 4.1 (Interpreting an oid)
Let $I = \langle \pi, v, \alpha \rangle$ be an instance and i be an OR-oid such that $v(i) = [j_1, ..., j_k]$ and $i \in \pi(\uparrow [\tau])$ for some τ. Then the set of interpretations Interpret-id(I, i) obtained by interpreting i in I, is defined as follows. In the following, (i/j) (o) denotes the result of syntactically replacing i by j in o.

Interpret-id(I, i)= $\{ I_j \mid j \in \{ j_1, ..., j_k \} \}$ where $I_j = \langle \pi_j, v_j, \alpha_j \rangle$ is given by

- **if i is persistent:**
  - *The value of i is refined and the values of the rest of the oids remains the same, i.e., $v_j(i') = v(i')$ if $i' \neq i$ and $v_j(i) = j$.*
  - *After interpreting, the oid i labels an o-value of type τ— not an o-value of type $\uparrow [\tau]$. Hence $\pi_j(\uparrow \tau) = \pi(\uparrow \tau) \cup \{i\}$, while $\pi_j(\uparrow [\tau_1]) = \pi(\uparrow [\tau_1]) - \{i\}$. For other types, π and π_j agree.*
  - *Furthermore, $\alpha_j = \alpha$*
- **if i is mutable:**
  - *i is replaced by j in the entire database. Hence, for every table B, $\pi_j(B) = \{ (i/j)\ (o) \mid o \in \pi(B) \}$ and for every $i' \neq i$, $v_j(i') = (i/j)\ (\ v(i')\ )$ and $\alpha_j(i') = \alpha(i')$.*
  - *The oid i no longer exists. Hence $\pi_j(\uparrow [\tau]) = \pi(\uparrow [\tau]) - \{i\}$. For any other pointer type $\uparrow \tau'$, $\pi_j(\uparrow \tau') = \pi(\uparrow \tau')$.*
  - *Since i no longer exists, v_j and α_j are undefined with respect to the oid i.*

We now extend the above to *interpretations of I obtained by interpreting a set O* of oids. At a first glance, one may think of obtaining this by applying Interpret-id(I, i) inductively on the set O in some order. The following example illustrates the problems with this approach.

Example 4.1
Consider a simple recursive schema S with just one table P whose type is given by $T(P) = \uparrow [P]$. Consider an instance $I = \langle \pi, v, \alpha \rangle$ of S where (i) $\pi(P) = \{i, j, k\}$ (ii) $v(i) = [j, k]$, $v(j) = [k, i]$ and $v(k) = [i, j]$ (iii) $\alpha(i) = \alpha(j) = \alpha(k) = m$ (i.e. all are mutables).

Consider the interpretations obtained by interpreting the three oids in two different orders $\sigma_1 = \langle i, j, k \rangle$ and $\sigma_2 = \langle j, i, k \rangle$. Let $O = \{i, j, k\}$. Then the reader can easily verify that set of interpretations of I obtained by interpreting O according to the order σ_1 is $\{ \{j\}, \{k\} \}$ while the same according to the order σ_2 is $\{ \{i\}, \{k\} \}$.

Thus, set of interpretations in the two orders are different, and the interpretations contain the mutable oids even after they have been interpreted. ■

The two properties, namely, *existence of mutable oids in interpreted instances* and *the dependence of the interpretations on the order of interpreting* are undesirable properties: the former violates the intuition that mutable oids vanish when interpreted, while the latter disallows the possibility of specifying interpretations declaratively. Defining these two undesirable properties will be helpful later.

Definition 4.2 (Ill-Formed Instance)
If an instance I is such that some of its interpretations obtained by interpreting a set of oids O continue to have some of the mutable oids in O, then I is called an ill-formed instance.

Definition 4.3
If an instance I is such that its interpretations obtained by interpreting a set of oids O depend on the order in which the oids of O are chosen for interpreting, then I is called an order-dependent instance.

We now capture a class of schemas that do not exhibit these properties by defining a notion of *well-behavedness.*

Definition 4.4 (Well-behaved schemas)
A schema is well-behaved if every instance of the schema is well-formed and order-independent.

Note that in the above example, the schema has a "cycle" through "$\uparrow [\ ]$". This observation leads to the following definition which will prove useful later.

Definition 4.5 (Definite-pointer types)
A definite-pointer type is any expression of the form $\uparrow \{\tau\}$, $\uparrow \langle \tau_1, \ldots, \tau_k \rangle$, $\uparrow \mathcal{D}$, or $\uparrow B$ where B is any table name.

Definite-pointer types do not contribute to either order-dependence or ill-formedness because their instances are always persistent oids with definite values. Thus, to obtain a syntactic characterization of *well-behaved schemas* we do not have to take definite-pointer types into account.

Definition 4.6 (OR-Cycle)
Given a schema $S = \langle \mathbf{B}, T \rangle$, first replace any definite-pointer-type appearing in the definition of any table in S by a base-type $\uparrow\mathcal{D}$ and then construct a di-graph $G(S) = (V, E)$ where $V = \mathbf{B}$ and $\langle B', B \rangle \in E$ iff B' appears in $T(B)$.

We say an edge $\langle B', B \rangle$ is an OR-Edge if $T(B)$ has a sub-expression $\uparrow[\tau]$ and B' appears in τ. We say that S has an OR-Cycle if $G(S)$ has a directed cycle through an OR-Edge.

The following theorem proves that the absence of an OR-Cycle is both necessary and sufficient for a schema to be well-behaved.

Theorem 4.1 (well-behavedness)
Let S be a schema. Then S is well-behaved iff $G(S)$ has no OR-Cycle.

Proof (Sketch): The proof in the "if" direction first establishes a topological order on the oids of O and then uses the following lemma whose proof will appear in the full paper.

> If O has a topological order, then the interpretations of I obtained by interpreting O are order-independent and well-defined.

In the "only-if" direction, the proof is by construction, i.e., we show an instance with an or-cycle such that its interpretations are ill-formed. ■

Thus, if S has no OR-Cycle, we can define the notion of *interpretations obtained by interpreting a set of oids*, denoted by Interpret-Set(I, O), as follows:

Definition 4.7 *Let I be an instance of a well-behaved schema S, O a non-empty set of oids appearing in I. Then if $O = \{i\}$, for some i, then Interpret-Set (I, O) =Interpret-id (I, i). Otherwise, choose some element i_1 in O and put $O' = O - \{i_1\}$. Furthermore, let $\mathcal{I}$ =Interpret-id (I, i_1). Then Interpret-Set (I, O) =*

$\bigcup_{I' \in \mathcal{I}}$ *Interpret-Set* (I', O')

5 Views and Queries

In this section we introduce the notion of *views* which plays a critical role in our query language. Recall that in our model one may query the original database (i.e., query at the representational level) or the set of interpretations obtained by interpreting some oids (i.e., query at interpretational level).

Example 5.1 *Consider the schema*

$$S = \langle \mathbf{B}, T \rangle,\ \mathbf{B} = \{TEACH\} \text{ and}$$
$$T(TEACH) = \langle Name : \mathcal{D}, Courses : \uparrow[\mathcal{D}] \rangle$$

Thus, $TEACH$ associates each teacher to an oid whose value gives the possible courses the teacher can teach. Consider the following queries:

1. Φ_1 = *Are there two teachers with intersecting choices of courses?*
2. Φ_2 = *Is it true that in every assignment of courses to teachers, at least two different teachers teach the same course?* ■

Let I be an instance of the schema S above. Then Φ_1 is a query at the representational level while Φ_2 is at the interpretational level. Thus, we need a facility to specify the level at which a particular formula is to be evaluated. This is accomplished through *views* by specifying the interpretations in which a formula has to be evaluated. Also we need to know whether the formula is to be evaluated in *every* interpretation or in *some* interpretations. Thus, a query in our formalism consists of (i) a view, (ii) a formula to be evaluated, (iii) SOME/EVERY indicating whether the formula is to be evaluated in some or every interpretation specified by the view. The view in turn specifies the interpretations by giving a set of table-names and the desired interpretations are obtained by interpreting every oid occurring in these tables.

Definition 5.1 (Queries: Syntax)
A query Φ in our query-language is a triple, Φ =

$\langle V, F, Quantifier \rangle$ where the view V specifies a set of table names $\mathbf{B_1}$, F is a existentially quantified conjunctive formula and $Q = ANY/EVERY$.

We illustrate the notion of a query by encoding the queries Φ_1 and Φ_2 of the previous example.

Example 5.2 We encode Φ_2 as the query $Q_2 = \langle \{TEACH\}, F_2, EVERY \rangle$ where
$F_2 = \exists(x\ y\ z) TEACH(x,z) \wedge TEACH(y,z) \wedge (x \neq y)$.

Thus, for any instance I, Q_2 is true in I iff for *every* interpretation of I obtained by interpreting the oids occurring in $TEACH$ entails the formula F_2. The formula F_2 checks if two different teachers teach the same course. Thus, Q_2 encodes Φ_2 correctly. Note that the entailment of a formula by an interpretation is a straightforward extension of First Order entailment.

We encode Φ_1 as the query $Q_1 = \langle \{\ \}, F_1, SOME \rangle$ where $F_1 =$

$$\exists(\ x\ y\ z\ x'\ y'\ z\ z'\ u)\ TEACH(x,x') \wedge TEACH(y,y') \wedge (\uparrow(x') = z) \wedge (\uparrow(y') = z') \wedge member(z,u) \wedge member(z',u)$$

Note that F_1 checks essentially if two teachers have intersecting choices of courses. The view associated with Q_1 does not interpret any oids. Hence, Q_1 is at the representational level. ■

The user who encodes F_2 is directing it at the interpretations of I which have no incompleteness in the second argument of $TEACH$. The user who encodes F_1, however, uses the fact that the second argument of $TEACH$ is a pointer to an OR-Type and hence the formula F_1 makes an explicit dereferencing of these oids. The literals $\uparrow(x')$, etc. achieve this dereferencing. Thus, a user must know how to compute the type of the interpretations after the respective oids in the tables specified by the view have been interpreted. We show later how the new type of a table is obtained from the old type using the operation $Modify(\tau)$.

We now formalize the meaning of a query that uses views by first informally explaining the notion of oids occurring in a table.

We first order the tables of a schema as follows: if B' appears in an expression $\uparrow[\tau]$ of $T(B)$, then B' is *below* B. This can be extended naturally to all tables because we consider schemas without OR-cycles. Since every o-value in our formalism can be viewed as a tree (similar to [AK 89]), every table B corresponds to a tree with root labeled by B, and with the o-values of B as the children of the root. The oids associated with a table B are those oids that occur in the tree labeled by B but do not occur in any tree labeled by a table B' that is below B.

Definition 5.2 (Query: Semantics)
For any schema $S = \langle \mathbf{B}, T \rangle$, a view V is a subset $\mathbf{B_1} \subseteq \mathbf{B}$ of the tables of S. The set of tables $\mathbf{B_1}$ is called the **set of tables specified by** *V. For any instance I, the interpretations associated with V, denoted by $REP(I,V)$, are obtained by interpreting* **every oid occurring in $\mathbf{B_1}$**.

For any instance I, and a query $Q = \langle V, F, SOME/EVERY \rangle$, Q is true in I iff for every (or some) interpretation I' in $REP(I,V)$, $I' \models F$. The notion of entailment is a straightforward extension of first-order entailment in complex-objects.

The next question is how does a view modify the types of certain tables which are interpreted. The operator $Modify(\tau)$ defines this modification.

Definition 5.3 (Modify(τ))
The operator Modify is defined inductively:

- $Modify(\mathcal{D}) = \mathcal{D}$
- $Modify(\uparrow\{\tau\}) = \uparrow\{\tau\}$
- $Modify(\uparrow\langle\tau_1, ..., \tau_k\rangle) = \uparrow\langle\tau_1, ..., \tau_k\rangle$
- $Modify(\uparrow[\tau]) = \uparrow\tau \bigvee \tau$
- $Modify(\tau_1 \bigvee \ldots \bigvee \tau_k) = Modify(\tau_1) \bigvee \ldots \bigvee Modify(\tau_k)$.

It is now clear how views can be specified. Theorem 4.1 guarantees that the interpretations associated with any view V are well-formed and order-independent if the schema has no OR-Cycles. We record this result as the following theorem.

Theorem 5.1 *For any schema S, if S has no OR-Cycles, then for any instance I and for any view V, the set of interpretations $REP(I, V)$ are well-formed and order-independent.*

Proof: Let O be the set of all oids that belong to any table refined by V. From Theorem 4.1 it follows that interpreting O is order-independent, and hence Interpret-Set$(I, O) = REP(I, V)$. The result follows. ■

6 Complexity of Queries

In this section we show that the data complexity of our query language is complete for CoNP. We identify a class of queries of the form $\langle View, Formula, SOME \rangle$, called *SOME-queries*, and show that they have PTIME data complexity. Our definition of data complexity is the same as in [Vardi 82], except that our databases are allowed to have complex objects.

Definition 6.1 *Data complexity of a query $Q = \langle View, Formula, Quantifier \rangle$ is the complexity of the set $G_Q = \{ D \mid D \models Q \}$*

In [IV 89] several CoNP-Hard queries are shown and since the class of databases allowed in the formalism of [IV 89] is a subset of our formalism, it follows that the data complexity in our case is CoNP-Hard also. The upper bound is established below. We consider only queries with $Quantifier = EVERY$ because later we show that the queries with $SOME$ as the quantifier are in PTIME and, hence, obviously are in CoNP.

Lemma 6.1 (Upper Bound)
The data complexity of our language is in CoNP.

Proof (sketch):
Let $Q = \langle V, F, EVERY \rangle$ be a query. We exhibit a non-deterministic polynomial time Turing machine T_Q such that on any input I, T_Q halts with a 'yes' iff $I \not\models Q$. Thus the complement of $I \models Q$ is in NPTIME, and hence the data complexity of Q is in CoNP. ■

We now show that the complexity of the class of queries, called *SOME-queries*, of the form
$\langle View, Formula, SOME \rangle$ is in PTIME. The idea is that a *SOME-query* can be answered by examining the structure itself, and without evaluating in the interpretations. We illustrate with an example.

Example 6.1 Let $H_1 = \uparrow[\mathcal{D}]$ and $H_2 = \uparrow[\mathcal{D}]$. Then consider the query Q

$$Q = \langle \{H_1, H_2\}, \exists x \; H_1(x) \wedge H_2(x), SOME \rangle$$

Given an instance I, the query Q is true in I if there is an interpretation of I obtained by interpreting all the oids in the tables H_1 and H_2 such that the interpretation entails the query $\exists x \; H_1(x) \wedge H_2(x)$.

We show that the above query can be translated into a formula F' that queries the structure of the instance I. Intuitively, the desired formula F' checks if there exist two oids x and y in H_1 and H_2 such that their values, i.e. OR-Objects, have some member in common. Thus it treats the OR-Objects as sets and checks for their intersection. Such a formula which queries the structure of I is given below.

$F' =$
$[\exists(x\ x'\ y\ y'\ z\ z'\ u)\ H_1(x) \wedge H_2(y) \wedge (\uparrow(x) = z) \wedge (\uparrow(y) = z') \wedge member(z, u) \wedge member(z', u)] \bigvee$
$[\exists(x\ y\ z\ )\ H_1(x) \wedge H_2(y) \wedge (\uparrow(x) = z) \wedge member(z, y)] \bigvee$
$[\exists(x\ \ y\ \ z)\ H_1(x) \wedge H_2(y) \wedge (\uparrow(y) = z) \wedge member(z, x)]$

In general either H_1 or H_2 can have constants as values, and hence we should also see if a value in H_1 is a member of the OR-Object labeled by an OR-Oid in H_2 and vice versa. The second and third disjuncts of F' do this. Now we need to see if I entails F'. This entailment, similar to the entailment of a formula by an interpretation, is a straightforward extension of first-order entailment for complex-objects.

It is well known (e.g., [AK 90]) that the data complexity of existential queries in complex-object databases is in PTIME. Therefore F' can be evaluated in I in PTIME. Thus, we showed that the data complexity of Q is in PTIME. ■

The above translation is not particular to F and any such query can be translated similarly. The number of disjunctions we get depends on the number of predicates and on the nesting of OR-Types in the original query. Since these

two are dependent on the size of the query, the translated query is also dependent on the size of the original query, and not on the size of the database.

In the following lemma by co-referencing we mean the repeated occurrences of the same OR-id.

Lemma 6.2
The data complexity of the class **SOME-queries** *is in PTIME for the class of datbases that have no co-referencing.*

7 Conclusion

We observed that the existing formalisms for non-traditional applications lack some useful functionalities: (i) ability to represent complete and incomplete designs together in a database (ii) encapsulating frequently used notions such as *final designs/plans/schedules* into the query-language (iii) ability to query at different levels: *implementational and structural.* We showed that these functionalities can be realized by extending object-oriented data models with a new type called *OR-Type.* While we studied only an existential query-language in this paper, a subsequent paper shall show a complete query language for databases with OR-Objects.

Acknowledgements

Work done by T. Imielinski and K. Vadaparty was partially supported by NSF grant IRI 89-22385.

References

[AK 89] S. Abiteboul and C. Kanellakis, *Object Identity as a Query language Primitive*, Proceedings of the 1989 ACM SIGMOD International conference on Management of Data.

[AK 90] S. Abiteboul and P. Kanellakis *Database Theory column: Query languages for complex object databases* SIGACT News, Summer 1990.

[Ba 88] F. Bancilhon, *Object-Oriented Database Systems*, Proceedings of the seventh ACM symposium on PODS, 1988.

[Ban 87] J. Banerjee H.T Chou, J. Garza, W. Kim, D. Woelk, N.Ballou and H.J. Kim *Data model issues for object-oriented applications* , ACM TOIS, Jan 1987.

[BNST 90] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur, *Set Constructors in a Logic Database Language*, accepted for publication in Journal of Logic Programming. Extended abstract in ACM PODS, 1987.

[CH 82] A.K. Chandra and D. Harel, *Structure and complexity of relational queries*, JCSS, 25(1): 99-128, 1982.

[GJ 79] M.Garey and D.Johnson, *Computers and Intractability, A guide to Theory of NP-Completeness*, W.H.Freeman and Company, 1979.

[IL 84] T. Imielinski and W. Lipski, *Incomplete information in relational databases*, JACM 31(4): 761-791, october, 1984.

[IV 89] T.Imielinski and K.Vadaparty, *Complexity of Querying Databases with OR-Objects*, Proceedings of the Eighth ACM Symposium on Principles of Database Systems (PODS), 1989.

[J 82] B. Jacobs. *On Database Logic* JACM 29:2, 310-332 (1982).

[LDM 87] G. Kuper, The Logical Data Model, Ph. D. dissertation, Stanford University, 1985.

[LRV 88] C.Lecluse, P.Richard and F.Velez, "O2 an Object oriented data models" ACM SIGMOD 88.

[Vardi 82] M. Vardi, *The Complexity of Relational Query Languages*, Proc. of 14th STOC, 137-146, 1982.

[VERSO 87] J. Verso, Verso: A database machine based on non-1NF relations, INRIA report 523, 1986.