FEATURES OF THE *TEDM* OBJECT MODEL

David Maier, Jianhua Zhu, Hitomi Ohkawa

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 89-018

December, 1989

FEATURES OF THE *TEDM* OBJECT MODEL†

David Maier
Jianhua Zhu‡
Hitomi Ohkawa

Department of Computer Science/Engineering
Oregon Graduate Center
Beaverton, Oregon 97006

Abstract: We present the object model of *TEDM*, a data model devised using concepts from both object-oriented systems and deductive reasoning systems. Each of of following topics is briefly discussed in this paper: 1) basic concepts of the model, 2) its complex object space, 3) its extension with abstract objects to the object space, 4) its data manipulation facilities, 5) its notion of rules and deductive reasoning on complex objects, 6) its approach to object representation of types and commands, and 7) its idea of dynamic construction of database programs.

Keywords: Object-oriented data models, Complex objects, Complex object logics, Rules, Deductive reasoning.

1. Introduction and Motivation

The need for complex objects in database applications has prompted new interests in searching for logic systems capable of automated deductions with complex objects. Two typical approaches are a) *transformational*, in which a complex object logic system is first mapped into a first-order logic system, or its weaker form the horn clause logic, and the method of the latter is then applied, and b) *axiomatic*, in which axioms and inference rules are established for a system of logic for complex objects and a proof theory within the logic system is then used [Beeri87, ChenW89, Kifer89, Kuper84, Maier86a]. This paper presents many interesting features of *TEDM*, a data model with a foundation in both the object-oriented paradigm and the logic programming paradigm. We take the axiomatic approach for formal investigation, while the transformational approach is employed as the primary implementation strategy.

The primary motivation behind our research is due to the growing need of applying database technologies to areas other than traditional business-type data processing. Many researchers proposed many data models during the past decade [Abiteboul87, Abriel74, Chen76, Codd79, Copeland84, Ecklund87, Hammer81, Katz83, Maier86,

† Work supported by NSF grant IST-83-51730, cosponsored by Tektronix Foundation, Intel, Mentor Graphics, DEC, Servio Logic Corp., Xerox and Beaverton Chamber of Commerce. The second author is also supported by a Graduate Research Fellowship from Apple Computer Corp.

‡ Current address: U S WEST Advanced Technologies, Advanced Software Technology, 6200 South Quebec Street, Suite 420, Englewood, Colorado 80122

Proceedings of DOOD '89,
Kyoto, Japan, December 1989

Mylopoulos80, Shipman81, Stonebraker87, Stonebraker88, Su83, Vbase86]. That clearly indicates large amount of effort spent in this area and its significance. But there is still no single data model that stands out and receives wide acceptance.

Nevertheless, there has been consensus on what it is that the conventional data models are lacking, and what it is that the new applications are seeking [Maier86, Maier87, Maier88, Sidle80, Rosenberg80]. In the mean time, different methodologies for attacking the problem have also been investigated, some of which have been shown to be very promising, especially in the area of integrating the logic paradigm and the object paradigm [Emden76, Gallaire84, Goldberg83, Kowalski78, Lloyd84, Maier84a, Stroustrup86]. The approach of *TEDM*, in particular, is based on prominent features from object-oriented systems and logic programming systems.

The organization of the paper is as follows. Basic concepts of *TEDM* are listed in the next section, and are elaborated in subsequent sections. In particular, Section 3 defines an object space and the notion of ***well-formed-object-terms***. Section 4 extends the object space with the notion of ***abstract objects***. Section 5 explains the model's data manipulation facilities and their core concept — ***pattern-matching***. Section 6 describes rules and deductive aspects of *TEDM*. Section 7 discusses two more extensions to the data model with the use of abstract objects. Section 8 illustrates the idea of dynamic program construction using an example. Finally, a summary and some concluding remarks are given in Section 9.

2. *TEDM* Overview

TEDM stands for *T*ektronix *E*ngineering *D*ata *M*odel, a data model originally described in [Maier85]. More recent work related to this data model can be found in [Anderson86, Anderson89, Ohkawa87, Zhu86, Zhu88]. We should point out that *TEDM* is a ***Structural*** object-oriented data model. Therefore, we are not concerned with issues related to methods, such as method inheritance, method combination and method overloading. However, it is not difficult to extend *TEDM* into a fully behavioral object-oriented model, and we have done some work along this direction [Zhu89].

The goal of *TEDM* is to provide flexible modeling tools and expressive data languages to engineering database applications. The model bears features whose origins can be distinctively traced to two important research areas of software systems and programming methodologies, namely, ***object-oriented*** systems and ***logic programming*** languages. In particular, the following adaptations are made in *TEDM*. From the object-oriented world, it acquires a notion of ***object identities***, ***complex objects***, a mechanism for object classification and a structure for property inheritance. From the logic programming world, it absorbs the concepts of ***unification*** (actually a special form called ***pattern-matching***) and ***answer substitution***, and a strategy for deductive query processing. *TEDM* also has a handful of innovative features that distinguishes it from other similar approaches, including the notion of ***abstract*** objects, object representation of types and commands, and support for dynamic command construction.

Much good work is being done in the area of object logic to provide a formal foundation for object-oriented systems, such as Ait-Kaci and Nasr's LOGIN [Ait-Kaci86], Bancilhon and Khoshafian's Complex Object Calculus [Bancilhon86], Chen and Warren's C-Logic [Chen89] and Kifer and Lausen's F-Logic [Kifer89], to just list a few. Our treatment of *TEDM* model has many similarities with the above-mentioned work, for example, our object language is an extension of the Ψ-terms of LOGIN, and the idea of using skolem functions as object identities is originated in [Chen89] and [Kifer89, 89a]. Nevertheless, the *TEDM* model has many distinctive features that the others do not have, such as the notion of abstract objects, the clear separation between intention and extension of database schemas, and the way its type system is formulated.

2.1. Object and Object Identities

Just like any other object-oriented systems, the basic modeling construct in *TEDM* is an *object*. Each object has an object *identity* and *may* also have internal structure, as a result of *composing* "smaller" objects. An object without internal structure is said to be a *simple* object; otherwise, it is a *complex* object. Notice that complex objects are always constructed from "smaller" objects by a *finite* number of compositions, starting from simple objects.

Intuitively, simple objects are the formal counterparts of well understood primitive abstract notions, such as *integers* and *character strings*. Complex objects model real world entities that inherently have internal structure and properties. For example, associated with a *person* there are properties such as the name of the person and his or her date of birth, etc. For the most part, the distinction between simple objects and complex objects is a conceptual one. Thus, simple objects are structureless, atomic and therefore *immutable*. On the other hand, complex objects are composite, decomposable and modifiable. Nevertheless, there is still an intimate connection between the simple/complex distinction and the objects' physical representations in hardware. Typically, simple objects are directly representable by the underlying hardware. The physical representation of complex objects, on the other hand, must rely on certain types of encoding. We point out that the distinction that we make between simple objects and complex objects is based on objects' conceptual immutability, which happens to coincide with the direct representability of the underlying hardware. An alternative way to draw the line is to view simple objects as those that don't have any associated properties, for example, a car about which we have nothing to say. However, there is nothing to prevent simple objects of this kind from evolving into complex ones. We prefer the former taxonomy because whether an object is simple or complex, in our view, is a static property.

The notion of object identity has its value both in conceptual modeling and in physical implementation. Each object has a unique object identity, which distinguishes the object itself from any other objects. The identity of an object is independent of the structure of the object. Thus it is possible to discern two objects that would otherwise be identical, which is useful, say, in a typical design of an electronic device, where several IC chips with identical physical and electrical parameters may be needed. In this case, any two chips can be interchanged without affecting the behavior the circuit, none of them is distinguishable from the other by its own properties. But when the design is stored by a database, it is necessary to distinguish the ICs for purposes of simulation and manufacturing. From the viewpoint of physical representation, the fact that object identities are not disk pointers but rather logical surrogate values makes it easy to reorganize databases, where large numbers of objects need be moved around.

2.2. Types and Object Conformity

Types are the classification mechanism in *TEDM*. Each type has two aspects, an *intentional* aspect and an *extensional* aspect. The extension of a type is a collection of objects of the type, and the intention of a type is a structure prescription that it expects its members to satisfy. The system actually does not explicitly maintain extensions of types. There may be multiple collections of objects to materialize a type extension.

The interaction between types and objects is modeled using two relations (in the mathematical sense). First, a *conformsTo* relation states that if an object possesses the structure that a type expects its elements to have, then the object conforms to the type. The condition for conformity only bounds the object structure from below. It is *prescriptive*: an object can have more structure than the type specifies, and still conforms to the type.

The second relation, *hasType*, captures the *declarational* imperative nature of *TEDM*'s type system. It states a stronger condition that not only an object conforms to a type, but also the fact that the object is *explicitly* declared to be a member of the type's extension. Thus the following relationship (with o being an object, and A being a type):

o *hasType* A $\Longrightarrow$ o *conformsTo* A

Notice that the *conformsTo* relation is a structural characterization of objects. For example, an object with an "x" field and a "y" field *conformsTo* the type Point (defined shortly), because it has all the fields required by the type definition. On the other hand, the object may not necessarily have the type Point: It may well be an object constructed as a solution to a system of linear equations with variables "x" and "y", namely, it is of type LinearEquationSolution2 that happens to have the same structure as the type Point. These choices in conceptual modeling is quite arbitrary. The system has no way to control nor should the system have control over these alternatives. The relation *hasType* is intended to give this high level control to the user: It is up to the user to decide on the intended conceptual constraints. On the other hand, the system is equipped with the *conformsTo* relation to guard against obvious inconsistencies, for example, assigning types to objects that are incompatible with the type definitions.

Notice also that an object can conform to and have multiple types. Types are defined using type definitions. A few examples are:

1) Point = (x $\rightarrow$ Integer, y $\rightarrow$ Integer)
2) Rectangle = (origin $\rightarrow$ Point, corner $\rightarrow$ Point)
3) RectSelect = (rect $\rightarrow$ Rectangle, cursor $\rightarrow$ Point)

Thus we define a type, Point, to model two-dimensional points, by an "x" coordinate and a "y" coordinate, as in 1). The type definition in 2), Rectangle, captures a rectangle by its upper-left "origin" point and its lower-right "corner" point. Similarly, 3) defines a RectSelect type as having a rectangle and a point, presumably modeling a rectangular region on a screen with a mouse point.

TEDM type system has benefited from known work on types in database programming languages. In particular, the *conformsTo* relation of this section and the *specializes* relation of the next section are very similar to the type system of Galileo [Albano85] and of Amber [Cardelli86]. However, the decision to have separate relations for capturing the structural compatibility and the declarational aspects is a novel feature and the separation is useful for conceptual modeling and is practical for implementation.

2.3. Type Hierarchy and Inheritance

Types are also involved in relations among themselves. A type *hierarchy* (actually a *semilattice*) is used to bind all the types together. Based on relative positions of types in this type hierarchy, *supertypes* and *subtypes* are recognized.

This type hierarchy affects both intentional as well as extensional aspects of types. The former is manifested in the form of *inheritance*: subtypes inherits structures from their supertypes. (*TEDM* is a structural model, namely, objects do not have associated behaviors. Hence, the term inheritance only refers to structural inheritance.) The latter takes the form of subset *inclusion*, the collection of objects having a certain type is included by the collection of objects for its supertypes.

Two relations are defined to formally back up this type hierarchy. First, a *specializes* relation holds from type A to type B, A *specializes* B, if the intentional structure of B is a included by that of A. Or equivalently, A *specializes* B, if for any object o,

o *conformsTo* A $\Longrightarrow$ o *conformsTo* B.

Similarly, a stronger relation, *isSubtypeOf*, forms the formal counterpart of the type hierarchy, type A is situated below type B in the type hierarchy if A *isSubtypeOf* B. That is, A *isSubtypeOf* B, if for any object o,

o *hasType* A $\Longrightarrow$ o *hasType* B.

This relation, again, must be explicitly declared. We also require

A *isSubtypeOf* B $\Longrightarrow$ A *specializes* B

Subtypes are also defined using type definitions. A type definition of the form

Point3D = Point:(z $\rightarrow$ Integer)

declares a new type, Point3D, as a subtype of the Point type. The same effect is also achieved by the following two definitions.

1) Point3D = (x $\rightarrow$ Integer, y $\rightarrow$ Integer, z $\rightarrow$ Integer)
2) Point3D < Point

Notice that, with the first type definition by itself, the type Point3D only specializes the type Point, but would not be a subtype.

2.4. Commands, Rules and Query Processing

Like traditional database management systems, *TEDM* provides data languages for accessing and manipulating database objects. In *TEDM*, *commands* are the primary means by which databases are accessed and manipulated. A command consists of a *pattern*, the body of the command, and an *action*, the head of the command, as is depicted using the following general form.

$$\text{Action}[Y_1, ..., Y_m] \Longleftarrow \text{Pattern}[X_1, ..., X_n]$$

The pattern denotes a matching function with an abstraction on X_1, ..., X_n. The action denotes an imperative operation with an abstraction on Y_1, ..., Y_m. For now, we assume

$$\{ Y_1, ..., Y_m \} \subseteq \{ X_1, ..., X_n \}$$

The semantics of such a command is realized by a two-phase procedure:

(1) applying a matching function using the pattern on the database, to yield a set of bindings for X_1, ..., X_n, and

(2) applying the imperative operation on the bound objects, passed to it via Y_1, ..., Y_m. Notice that the parameters, Y_1, ..., Y_m, obtain their values from the active set of bindings for X_1, ..., X_n.

Rules almost have the same form as that of commands, as shown below.

VirtualData[Y_1, ..., Y_m] ← Pattern[X_1, ..., X_n]

Essentially, rules assert logical consequences based on known structures, or introduce virtual data based on stored data.

Query processing in the presence of rules becomes a deductive process. Pattern-matching has to take into consideration of virtual data. In other words, a database should now be viewed as a closure of physical data plus virtual data derivable by database rules, or logical consequences. Alternatively, database rules can be viewed as dormant database commands with transient results. They execute on demand, produce temporary data, and go back to sleep afterwards.

3. Object Terms and An Object Space

This section defines object terms and an intended object space of interpretation. We construct the object space in such a way that it admits encoded complex objects. We use an extended first-order language to describe objects. We start off by introducing the notion of *well-formed-object-terms* (WFOTs). WFOT is constructed using symbols from the following denumerable sets:

1) constant symbols DS
2) placeholder symbols PS
3) type symbols TS
4) field label symbols LS

In addition, we also use, among others, the following auxiliary symbols:

1) the colon ":" to indicate type symbols
2) the caret "^" to indicate placeholder symbols
3) the question mark "?" to indicate object tag symbols (in next section)

The set WFOT is defined as follows.

1) if ds $\in$ DS, ts $\in$ TS, then ts:ds $\in$ WFOT
2) if ps $\in$ PS, ts $\in$ TS, $ls_i \in$ LS and $f_i \in$ WFOT,
   then ts:ps^($ls_1 \rightarrow f_1$, ..., $ls_n \rightarrow f_n$) $\in$ WFOT
3) if ts $\in$ TS and f $\in$ WFOT, then ts:f $\in$ WFOT

We construct an object space, OS, in which members of WFOT are interpreted. The following semantic entities are assumed:

1) a set of constants D
2) a set of object identities I
3) a set of binary relations R, one r_i ($\in$ R) $\subseteq$ I $\times$ (I $\cup$ D) for each field label l_i
4) a set of subsets T, one t_i ($\in$ T) $\subseteq$ (I $\cup$ D) for each type symbol ts_i

The construction of OS proceeds as follows:

1) if d $\in$ D, then d $\in$ OS
2) if id $\in$ I, $os_i \in$ OS, and (id, os_i) $\in r_i$, then (id, { id r_1 os_1, ..., id r_n os_n }) $\in$ OS

The cryptic form of (id, { id r_1 os_1, ..., id r_n os_n }) is an encoding for a complex object with identity id that is related to "smaller" objects os_i via r_i. An element of OS is said to be an *object*. The type of an object is determined by the type of its identities. (Constants are a special kind of identities.) In implementation, it is acceptable to equate the set of constant symbols, DS, with the set of constants themselves, D. The object identities for complex objects are uniquely generated by the system. The users do not know of their existence, nor do they have access to such identities.

The interpretation of a term in WFOT, s, into an object in OS, $\mu[\![s]\!]$, under a given mapping m from PS to I, is as follows.

1) $\mu[\![s]\!] = d$, if $s \equiv$ ts:ds, ds $\in$ DS and d $\in$ t for ts (meaning t is a set denoted by symbol ts),

2) $\mu[\![s]\!] = (\text{id}, \{ \text{id } r_1\ os_1, \ldots, \text{id } r_n\ os_n \})$, if $s \equiv$ ts:ps^(ls$_1 \rightarrow f_1, \ldots, ls_n \rightarrow f_n$), $\mu[\![f_i]\!] = os_i$, and id $\in$ t for ts

3) $\mu[\![s]\!] = \mu[\![f]\!]$, if $s \equiv$ ts:f and $\mu[\![f]\!] \in$ t for ts

Item 2) above assumes that the placeholder symbol ps is mapped (*bound*) to the object identity id: m(ps) = id.

Not all objects in OS are *true* objects. Two possible sources of problems come from 1) type violation, where an object may be in a type to which it does not conform; and 2) unique identity violation, where two different objects are assigned the same object id. We constrain OS not to have these violations, but do not give formal semantics here.

A valuation function (with a range of { t, f }, say) on WFOT can be established from the notion of *true* objects, in conjunction with the interpretation, μ, in a straightforward way. Neither is it difficult to define *well-formed-object-formulas* (WFOFs), and obtain soundness and completeness results relating the *provability* and *validity* of WFOFs similar to those in first-order logic. We will touch this issue again later in the context of query processing.

From a system viewpoint, terms in WFOT are typically used as object constructors, each generating a new instance in the database, or object space. Some examples are:

```
1) Point:P^(x → Integer:0, y → Integer:0)

2) Rectangle:R^(origin → Point:P1^(x → Integer:0, y → Integer:0),
                corner → Point:P2^(x → Integer:5, y → Integer:5))

3) RectSelect:S^(rect → Rectangle:R^(origin →
                           Point:P1^(x → Integer:0, y → Integer:0),
                                  corner →
                          Point:P2^(x → Integer:6, y → Integer:6)),
                cursor → Point:P^(x → Integer:3, y → Integer:3))
```

Figure 1(a) provides an illustration for the object created by the last example. Figure 1(b) shows that subobject sharing can be described by duplicating placeholders:

```
4) RectSelect:S^(rect → Rectangle:R^(origin →
                           Point:P1^(x → Integer:0, y → Integer:0),
                                  corner →
                           Point:P2^(x → Integer:6, y → Integer:6)),
                cursor → Point:P1^(x → Integer:0, y → Integer:0))
```

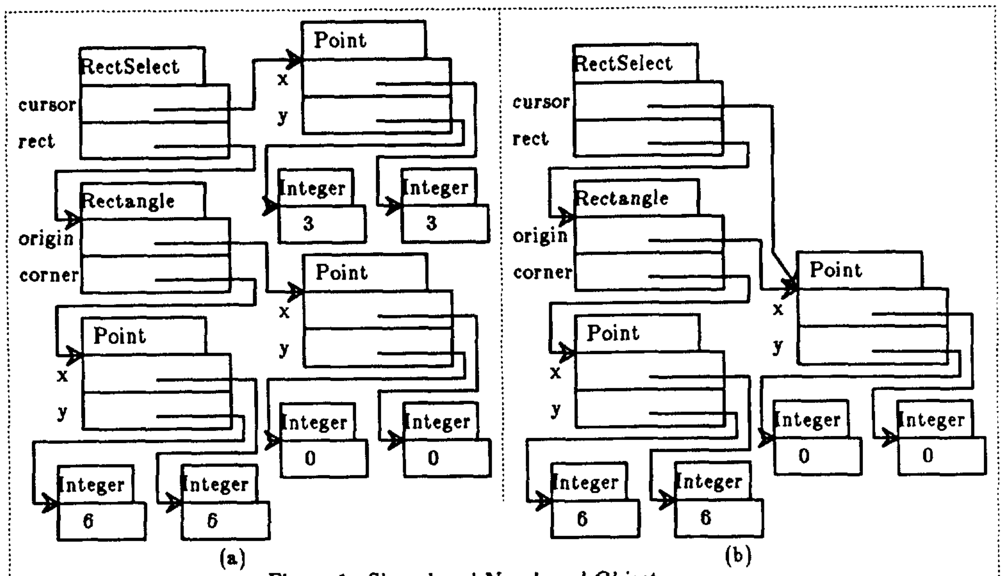

Figure 1. Shared and Nonshared Objects

4. An Extended Object Space

An extended object space, EOS, is formed by adding *abstract* objects, a class of objects with special interpretation requirements. The construction of EOS is similar to that of OS. We postulate a new set of semantic entities, A, the set of *abstract* object identities. In addition, the meaning of the set T is changed accordingly to take elements of A into consideration, namely, t_i ($\in$ T) $\subseteq$ (I $\cup$ A $\cup$ D) for each type symbol ts_i.

1) if d $\in$ D, then d $\in$ EOS
2) if a $\in$ A, then a $\in$ EOS
3) if id $\in$ (I $\cup$ A), $os_i \in$ EOS, and (id, os_i) $\in r_i$,
   then (id, { id r_1 os_1, ..., id r_n os_n }) $\in$ EOS

Assuming an additional symbol set, SS, the set of object *tag* symbols, the following modified version of WFOT, EWFOT, makes abstract objects accessible at the syntax level.

1) if ss $\in$ SS, then ss? $\in$ EWFOT
2) if ds $\in$ DS, ts $\in$ TS, then ts:ds $\in$ EWFOT
3) if ps $\in$ PS, ts $\in$ TS, $ls_i \in$ LS and $f_i \in$ EWFOT,
   then ts:ps^($ls_1 \rightarrow f_1$, ..., $ls_n \rightarrow f_n$) $\in$ EWFOT
4) if ps $\in$ PS, ss $\in$ SS, ts $\in$ TS, $ls_i \in$ LS and $f_i \in$ EWFOT,
   then ts:ss?ps^($ls_1 \rightarrow f_1$, ..., $ls_n \rightarrow f_n$) $\in$ EWFOT
5) if ts $\in$ TS and f $\in$ EWFOT, then ts:f $\in$ EWFOT

The extension to the interpretation function, μ, is to make sure that object tags get mapped to abstract objects. We may assume that there is a 1-1 correspondence (denoted using ι) from the set of object tags, SS, to the set of abstract objects, A. The extended version of μ, $\hat{\mu}$, is the following:

1) $\hat{\mu}[\![s]\!]$ = a, if s $\equiv$ ss?, a $\in$ A and ι(ss) = a
2) $\hat{\mu}[\![s]\!]$ = d, if s $\equiv$ ts:ds, ds $\in$ DS and d $\in$ t for ts
3) $\hat{\mu}[\![s]\!]$ = (id, { id r_1 os_1, ..., id r_n os_n }), if s $\equiv$ ts:ps^($ls_1 \rightarrow f_1$, ..., $ls_n \rightarrow f_n$),
   $\hat{\mu}[\![f_i]\!]$ = os_i, and id $\in$ t for ts

4) $\hat{\mu}[\![s]\!] = (a, \{ a\ r_1\ os_1, ..., a\ r_n\ os_n \})$, if $s \equiv$ ts:ss?ps^($ls_1 \rightarrow f_1, ..., ls_n \rightarrow f_n$),
$\hat{\mu}[\![f_i]\!] = os_i$, $\iota(ss) = a$ and $a \in t$ for ts
5) $\hat{\mu}[\![s]\!] = \hat{\mu}[\![f]\!]$, if $s \equiv$ ts:f and $\hat{\mu}[\![f]\!] \in t$ for ts

Several examples of object terms in EWFOT are provided below.

1) Point:P?Q^(x → 0)
2) Point:P?Q^(y → Y?)
3) Rectangle:R?S^(origin → Point:P?Q^(x → 0, y → Y?)

Item 3) describes an abstract "Rectangle" object, which consists of an abstract "Point" object, a constant "0" and another abstract object. Intuitively, we use this abstract object to represent a pattern that can match "Rectangle" objects whose "origin" point has an x-coordinate of "0". Moreover, the pattern would also return the "origin" point object and the y-coordinate of the "origin" point of the matched object.

Some remarks on the motivation of extending the object space with abstract objects before leaving this section. Our ultimate goal is to support dynamic database program construction, which is predicated on being able to store program pieces as data, thus manipulable using ordinary commands. As alluded to earlier, a database command is a juxtaposition of two pieces of code, a *pattern* routine and an *action* routine. Either the pattern or the action may contain "free" variable symbols. These "free" variables are not really free in the sense they cannot just be bound to any objects. Rather, the pattern code dictates how these variables can be bound and the action code dictates how the binding objects can be used. In other words, these variables carry with them a binding environment and a use environment, which more or less resemble internal structures in objects. Introducing abstract objects and associating them with "free" variables in patterns and actions make it possible to have *pattern objects* and *action objects*, and hence *command objects* even *program objects*. The comments above shall be substantiated throughout the rest of the paper.

Lomet [Lomet73] discussed similar ideas about program constructions in an operator driven model for program execution.

5. Data Manipulation Facilities

The command language provided by *TEDM* for data manipulation purposes is described in this section. In particular, the central idea of the command language, *pattern-matching*, is explained.

5.1. Patterns and Pattern-matching

The syntax for patterns is a slight variation of that of WFOT, and they are constructed using the same sets of symbols. The set of *well-formed-object-patterns*, WFOP, is obtained as follows. First, we need to define an auxiliary notion, *well-formed-object-literal*, or WFOL.

1) if ds $\in$ DS, then ds $\in$ WFOL
2) if $ls_i \in$ LS and $f_i \in$ WFOL, then ($ls_1 \rightarrow f_1, ..., ls_n \rightarrow f_n$) $\in$ WFOL

Now the set WFOP is:

1) if ps $\in$ PS and ts $\in$ TS, then ts:ps $\in$ WFOP
2) if ps $\in$ PS, ts $\in$ TS, $ls_i \in$ LS and $f_i \in$ WFOP or $f_i \in$ WFOL,
then ts:ps^($ls_1 \rightarrow f_1, ..., ls_n \rightarrow f_n$) $\in$ WFOP
3) if ts $\in$ TS and f $\in$ WFOP, then ts:f $\in$ WFOP

Notice that the only difference between this construction and the one for WFOT lies in the treatment on the roles of constants and placeholders. An element of WFOT must bottom-out at some constants. On the contrary, an element of WFOP must not be fully instantiated. *TEDM* supports the notion of ***negative*** patterns but we do not cover them in this paper.

These are some example instances of WFOP:

1) Point:Q^(x $\rightarrow$ 0)
2) Point:Q^(y $\rightarrow$ Integer:Y)
3) Rectangle:S^(origin $\rightarrow$ Point:Q^(x $\rightarrow$ 0, y $\rightarrow$ Integer:Y)

In *TEDM*, pattern-matching, referring to matching a formula in WFOP against an object in EOS, is an important concept. We distinguish two kinds of pattern-matching, that on *concrete* (non-abstract) objects and that on abstract objects, and discuss each of them in turn.

The meaning of ***concrete-match*** is stated as follows, given a formula f $\in$ WFOP and an object o $\in$ OS.

1) For f $\equiv$ ts:ps, f matches o if

a) o $\equiv$ d $\in$ D and d $\in$ t for ts, or
b) o $\equiv$ (id, { id r_1 os_1, ..., id r_n os_n }) $\in$ OS, and id $\in$ t for ts

2) For f $\equiv$ ts:ps^(ls_1 $\rightarrow$ f_1, ..., ls_n $\rightarrow$ f_n), f matches o if

a) o $\equiv$ (id, { id r_1 os_1, ..., id r_n os_n, ..., id r_m os_m }),
b) id $\in$ t for ts, and
c) for each i, f_i matches os_i

3) For f $\equiv$ ts:g, f matches o if

a) g matches o, and
b) o $\in$ t for ts

This definition needs extension in one case. In 2.c) above, an f_i can come from WFOL instead of WFOP, and the definition does not cover that case. We remedy the flaw by a supplementary notion of ***l-match***, which is stated as follows (assuming l $\in$ WFOL and o $\in$ OS).

4) For l $\equiv$ ds, l l-matches o if o $\equiv$ d $\in$ D and ds represents d

5) For l $\equiv$ (ls_1 $\rightarrow$ l_1, ..., ls_n $\rightarrow$ l_n), l l-matches o if

a). o $\equiv$ (id, { id r_1 os_1, ..., id r_n os_n, ..., id r_m os_m }), and
b). for each i, l_i l-matches os_i

Notice the difference between a match and an l-match. A match is a structure preserving mapping from a pattern formula into a database object, and retains a binding of placeholders to objects as its result. On the other hand, object literals do not contain placeholders, so an l-match is only a correspondence without bindings.

We also need a notion of the ***result*** of pattern-matching. The result of a match is a ***binding*** (or assignment) of placeholders to database objects. We use the following form to denote such a binding:

$$[\ P_1{:}o_1, \ldots, P_n{:}o_n\ ]$$

meaning placeholder P_i is bound to object o_i. Then, in case 1.a), the placeholder ps is bound to d, and the result of the match is [ps:d]. Similarly, in case 1.b), ps is bound to id, and the result is [ps:id]. In case 2), the placeholder ps is also bound to id. The result in this case [ps:id] + B_1 + ... + B_n, where B_i is the result of matching f_i on os_i, and

$$[\ P_1{:}o_1, \ldots, P_n{:}o_n\ ] + [\ Q_1{:}b_1, \ldots, Q_m{:}b_m\ ] = [\ P_1{:}o_1, \ldots, P_n{:}o_n, Q_1{:}b_1, \ldots, Q_m{:}b_m\ ]$$

Each successful match of a formula on a database object produces a binding:

$$[\ P_1{:}o_1, \ldots, P_n{:}o_n\ ]$$

Since the formula can match more than one object in the database, the general form for the result of pattern-matching is a set of bindings:

$$\begin{array}{l}[\ P_1{:}o_{1,1}, \ldots, P_n{:}o_{1,n}\ ]\\ [\ P_1{:}o_{2,1}, \ldots, P_n{:}o_{2,n}\ ]\\ \ldots\\ [\ P_1{:}o_{m,1}, \ldots, P_n{:}o_{m,n}\ ]\end{array}$$

While in concrete-match the targets of match are restricted to concrete objects, *abstract-match* removes this restriction. Before discussing abstract-match, An extension to WFOP is in order, to obtain *extended-well-formed-object-patterns*, EWFOPs.

1) if ps $\in$ PS and ts $\in$ TS, then ts:ps $\in$ EWFOP
2) if ps $\in$ PS and ts $\in$ TS, then ts:ps? $\in$ EWFOP
3) if ps $\in$ PS, ts $\in$ TS, $ls_i \in$ LS and $f_i \in$ EWFOP or $f_i \in$ WFOL,
   then ts:ps^($ls_1 \rightarrow f_1, \ldots, ls_n \rightarrow f_n$) $\in$ EWFOP
4) if ps $\in$ PS, ts $\in$ TS, $ls_i \in$ LS and $f_i \in$ EWFOP or $f_i \in$ WFOL,
   then ts:ps?($ls_1 \rightarrow f_1, \ldots, ls_n \rightarrow f_n$) $\in$ EWFOP
5) if ts $\in$ TS and f $\in$ EWFOP, then ts:f $\in$ EWFOP

The definition simply extends WFOP with formulas with placeholders that must be bound to abstract objects in a successful match, which is indicated using a question mark "?". The meaning of abstract-match, then, is to match a formula f $\in$ EWFOP against an object o $\in$ EOS. The precise definition of this match operation parallels that of concrete-match. The difference is that placeholders with question marks must be bound to abstract objects in pattern-matching.

If we put together the two sets, WFOT and WFOP, we obtain essentially what are usually called *well-formed-formulas* (WFFs), or more appropriately in our case, *well-formed-object-formulas* (WFOF). Thus, we say a pattern is *satisfiable* if, after assignment of object identities to placeholders in the pattern, it gets interpreted into a true object or an *initial* portion [Zhu88] of a true object, by some interpretation function $\hat{\mu}$. This true object is said to be a *model* of the pattern. From this point of view, matching a pattern with a database is effectively a model discovery process for the pattern, which happens to keep a record of assignments of the placeholders leading to models.

Abstract-match is a vehicle for accessing and manipulating abstract objects with commands. Such an example will be provided later in the paper.

5.2. Actions in Commands

The second component of a command, the head of the command, denotes an operation, with an abstraction on free variables occurring in the head. Thus, we had the following general abstract form (Section 2):

$\mathrm{Action}[Y_1, ..., Y_m]$,

which is effectively an n-nary operator, except we treat it as an operator over a set of n-tuples. The operator operates on sets and is highly polymorphic. It accepts the results of pattern-matching, a set of bindings, each providing an operand for each of the m variables.

The next a few subsections elaborate this abstract operator.

5.2.1. Object Evolution: I (Changing Type Membership)

One aspect of object evolution is that objects may gain or lose type memberships during their lifetime. The syntax form for denoting such changes is the simplest among all the actions.

The following syntax scheme expresses an membership-adding operation:

$ts{:}ps$, $ts \in TS$ and $ps \in PS$

Notice, ps is a placeholder whose instantiation relies on bindings of pattern-matching. Thus, for command

Employee:P $\Longleftarrow$ Person:P^(name $\rightarrow$ "John"),

suppose that the result of pattern-matching is

$[P{:}p_1]$
$[P{:}p_2]$
$[P{:}p_3]$

where p_1, p_2 and p_3 are objects of type "Person", then the effect of this command is that p_1, p_2 and p_3 each acquire a membership in the type "Employee" (and its super-types).

The syntax scheme for objects to be removed from a type is the following:

$ts{\sim}ps$, $ts \in TS$ and $ps \in PS$

5.2.2. Object Evolution: II (Changing Field Structure)

Objects can also evolve along another dimension — they can acquire new fields, lose old fields, as well as update existing fields.

The following is the syntax scheme for adding new fields to an object, given that $ps \in PS$, $ls_i \in LS$ and $f_i \in DS$ or $f_i \in PS$.

$ps\ (ls_1 \rightarrow f_1, ..., ls_n \rightarrow f_n)$

The effect of this operation is that if the object does not have field "ls_i, then field "ls_i" is added; otherwise the existing field is updated. *TEDM* also supports *multiple-occurrence* field. We need separate syntax for field-add operation and for field-update operation in the case of multiple-occurrence field, since there is a difference between changing an existing field and adding a new occurrence of the field.

The following syntax is used to denote objects losing fields:

$ps\sim(ls_1 \rightarrow f_1, ..., ls_n \rightarrow f_n)$

One way to achieve update operation is to use a delete operation followed by an add operation. Notice that objects do not lose their object identifiers when their fields are deleted, which is an important precondition that makes this alternative for update feasible. However, we do need to make sure that in the simulated update (delete followed by add), pattern matching provides the same set of objects to operate on, which can be guaranteed by either enclosing the delete operation and the add operation in a compound command (see below), or extending the command language to allow multiple heads.

5.2.3. Object Creation

The general syntax scheme for object creation is the following:

$ts*(ls_1 \rightarrow f_1, ..., ls_n \rightarrow f_n)$

where $ts \in TS$, $ls \in LS$, and f_i is either a constant, a placeholder or another object creation scheme.

The asterisk "*" in this form dictates a new object identity be generated for each of the bindings from pattern-matching.

5.2.4. Compound Commands

One last kind of command action is a *compound* command invocation. To describe it, we introduce an additional symbol set, CS, a set of command symbols. The general syntax form, then, is the following:

$cs[ps_1, ..., ps_k]$, $cs \in CS$ and $ps \in PS$

The operation performed by this command invocation is determined by the definition of the compound command, which will not be discussed in this paper. An example compound command definition is:

```
HireEmployee[arg1 → Person:P, arg2 → Dept:D]
{

  P(dept → Dept:D) <== HireEmployee(arg1 → P, arg2 → D);
  Employee:P <== HireEmployee(arg1 → P);
}
```

6. Rules and Deductions

Rules are a special kind of well-formed-object-formula. They are made up of an *antecedent* and a *consequent*. The abstract form of a rule is (Section 2):

$VirtualData[Y_1, ..., Y_m] \leftarrow Pattern[X_1, ..., X_n]$,

where the part to the right of $\leftarrow$ is the antecedent (an element of WFOP), and the part to the left of $\leftarrow$ is the consequent. The syntax schemes for the consequent are similar to those for actions in commands — not only is the former a subset of the latter, but also they have closely related semantics. The intuitive connection between the two is that rules can be viewed as deferred updates.

The deductive component is built into the query processor for pattern matching, which is similar to but more powerful than a Prolog interpreter, since it needs to deal with rules that cannot be translated into Horn clauses. In fact, we built the deduction component on top of C-Prolog in an earlier *TEDM* prototype implementation. The interaction between the deductive component and the imperative component is restricted in the following sense: Within a command, the imperative action cannot start until the deduction is completed.

6.1. Virtual Memberships

The following syntax scheme is used to express a virtual type membership for objects.

$$ts{:}ps,\ ts \in TS \text{ and } ps \in PS,$$

which is the same syntax as that for objects acquiring type memberships. This part of the rule states that, given the result of a pattern-matching, the objects bound to the placeholder ps are to be virtual members of the type denoted by ts. This is not a stored fact, but is always derivable as long as the relevant rule exists.

These virtual data asserted by rules affect the pattern-matching phase during a command execution. They are not substantiated with physical data nevertheless. Hence, the meaning of rules of this kind is just like that of membership-add operations in commands, except the former are executed only on demand, and the consequence of the execution has a rather short lifetime: after the command execution, no physical data are altered with respect to the virtual membership.

6.2. Virtual Fields

Similarly, rules for virtual fields are like operations for objects acquiring new fields. Their consequent uses the following syntax scheme:

$$ps\ (ls_1 \rightarrow f_1, \ldots, ls_n \rightarrow f_n),$$

where $ps \in PS$, $ls_i \in LS$ and $f_i \in DS$ or $f_i \in PS$.

The remarks made in the previous subsection also hold here. We also point out that since the rules presented so far only reference existing database objects, they are *safe* in the sense they will not introduce infinite loops into the deduction engine, even when recursions are involved. Alternatively, we can consider the closure of the database under the rules. It is clear that with the rules discussed, the closure of a finite database is always finite.

6.3. Virtual Objects

The general syntax form for rules asserting virtual objects resembles that for object creation, as shown below.

$$ts{*}(ls_1 \rightarrow f_1, \ldots, ls_n \rightarrow f_n)$$

where $ts \in TS$, $ls \in LS$, and f_i is either a constant, a placeholder or another virtual object scheme. Similarly, the asterisk "*" in this form indicates a ***temporary*** object identity is to be used for each virtual object derived.

There are two technical difficulties related to rules for virtual objects. First, they might lead to *unsafe* computations, since virtual objects amounts to temporary object creation. Therefore, a recursive rule might potentially lead to creation of infinitely many temporary objects. To avoid this problem, we restrict our rules for virtual objects to nonrecursive ones. Note that other types of rules can be recursive, so we can do, for example, transitive closures.

A second problem is that virtual object identities may violate the assumption that each object is uniquely identifiable by its identity, since virtual objects use temporary identities. In particular, given the same set of bindings from pattern-matching, different invocation of the same rule should conceptually derive the same set of virtual objects. But unless we cache these virtual objects, it is a difficult task to guard against using different object identities for different rule invocations.

One way to get around this identity consistency maintenance problem on virtual objects is to encapsulate the temporary object identities using *skolem* functions. In other words, a new style is adopted for temporary object identities, or function object identities. Thus, each virtual object has an identity that is functionally dependent on a specific binding that leads to the derivation of the virtual object, which remains fixed as long as the binding does not change. The idea of using skolem functions as object identities is originated in the work by Chen and Warren [ChenW89] and in the work by Kifer and Wu [Kifer89].

7. Object Representation of Types and Commands

This section demonstrate the uses of abstract objects. We describe two extensions to the data model itself with the use of abstract objects. In the first case, type definitions are represented as database objects. Due to limited space, our discussion will be restricted to an interesting aspect of this representation scheme, where abstract objects are used to capture the structure information of type definitions. In the second case, we discuss object representation of database commands. Again, the exposition will be restricted, to a subproblem — pattern representations.

7.1. Type Defining Objects

In *TEDM*, type definitions are stored as database objects of type "Typedef". Objects of type "Typedef" are called *type-defining-objects*. In the center of a type-defining-object is an abstract object of the type that is being defined by the type definition itself. The abstract object represents the structure of the type. We use a few examples to explain the idea. Consider the type definition for type "Point":

Point = (x → Integer, y → Integer)

In the type-defining-object for "Point", an abstract object is described by

Point:P1?P2^(x → Integer:X?, y → Integer:Y?)

where "P1", "X" and "Y" are object tag symbols, and "P2" is a placeholder symbol. The presence of "P1" dictates that the object constructed is an abstract "Point" object. The placeholder symbol "P2" is only a syntactical requirement and does not have semantic significance. Similarly, for the type definition

Rectangle = (origin → Point, corner → Point)

the following abstract object can be used as its structure representation:

Rectangle:R1?R2^(origin → Point:P1?, corner → Point:P2?)

7.2. Command Defining Objects

We provide an abstract object representation scheme for patterns, according to the construction of WFOP. First, for a pattern of the form

ts:ps,

its representation is denoted by the following object term

ts:ps?

Second, a pattern of the form

$ts{:}ps\hat{}(ls_1 \rightarrow f_1, \ldots, ls_n \rightarrow f_n)$

can be represented using the object described by

$ts{:}o?ps\hat{}(ls_1 \rightarrow o_1, \ldots, ls_n \rightarrow o_n)$

where o_i is an object representation for f_i. Third, an object denoted using

ts:o

can be used to represent a pattern of the form

ts:f

where o is an object representation of f.

8. Dynamic Constructions of Database Programs

We illustrate the idea of dynamic construction of database programs by showing how we can construct a more complex pattern from two simpler ones. Consider the following two patterns:

1). Point:Q^(x → 0)
2). Rectangle:R

From the previous section, we can use the two abstract objects denoted by

1). Point:P?Q^(x → 0)
2). Rectangle:R?

to represent the two patterns respectively. The following command

R(origin → P) <== Rectangle:R?, Point:P?(x → 0)

then, will among others create an object described by

Rectangle:S?R^(origin → Point:Q?P^(x → 0))

which in turn is a representation for the following pattern:

Rectangle:R^(origin → Point:P^(x → 0))

9. Summary and Concluding Remarks

We have shown in this paper a number of interesting ideas in our approach to combining the power of object-oriented systems and logic programming systems. An overview is provided for a data model (*TEDM*) that is designed based on those concepts. A key extension in our approach is the notion of abstract objects, which breaks up variables in traditional logic systems into two parts. The first part in this partition preserves the meaning of variables as pure placeholders, while the second part contains "free" symbols that can directly be interpreted by a semantical function as abstract objects. We have shown several uses of abstract objects, and the possibility of constructing dynamic database programs with them.

There are other features of *TEDM* left uncovered by this paper. One of them is the notion of *ordered* field, a special kind of multiple-occurrence field in which the order of field values is significant. Ordered fields are needed in design applications. For example, in VLSI design, a larger module may consist of several smaller ones. This fact can be stored using a multiple-occurrence field. For the purpose of simulation, we need to determine an order to excite the modules, according to its relative data path from the input (levelization). Ordered fields would be most ideal in this situation.

Finally, object representation of commands is an important concept. With command objects, command syntax is not an important issue any more, and we can always define different kinds of surface syntax for users' convenience.

Our effort is an indication that the deductive object-oriented approach to database systems is a promising direction for extending the database technologies to wider application domains. Many applications such as CAD/CAM require data models to have sophisticated modeling tools as well as powerful and flexible manipulation tools. Object-oriented systems are at their best in terms of modeling capabilities; and logic deductions are a natural extension to currently available query languages, such as relational algebra or calculus.

References

[Abiteboul87] "IFO: A Formal Semantic Database Model," Abiteboul, S. and Hull, R., *ACM Transactions On Database Systems*, Vol.12, No.4, 1987.

[Abriel74] "Data Semantics," Abriel, J. R., *Data Base Management Systems*, 1974.

[Ait-Kaci86] "LOGIN: A A Logic Programming Language with Built-in Inheritance," Ait-Kaci, H. and Nasr R., *Journal of Logic Programming*, Vol.3, PP185-215, 1986.

[Albano85] "Galileo: A Strongly-Typed Interactive Conceptual Language," Albano, A. and Cardelli L., *ACM Transactions on Database Systems*, Vol.10, No.2, 1985.

[Anderson86] "PROTEUS: Objectifying the DBMS User Interface," Anderson, T. L., Ecklund, E. F. Jr. and Maier, D., *Proceedings of the International Workshop on Object-Oriented Database Systems*, 1986.

[Anderson89] "Representing CSG Solids Using a Logic-Based Object Data Model," Anderson, T. L., Ohkawa, H., Gjovaag, J., Maier, D. and Shulman, S., *Proceedings of the International Workshop on Object-Oriented Database Systems*, 1989.

[Bancilhon86] "A Calculus for Complex Objects," Bancilhon, F and Khoshafian, S. N., *Proc. of the ACM Symposium on Principles of Database Systems*, 1986.

[Beeri87] "On Combining Object Orientation and Logic Programming," Beeri, C., XP8.5i Workshop, Oregon Graduate Center, 1987.

[Cardelli86] "Amber," Cardelli, L., *Combinators and Functional Programming Languages*, Cousineau, G., Curien, P. and Robinet, B. (eds.), Springer-Verlag, NY, 1986.

[Chen76]	"The Entity-Relationship Model: Toward a Unified View of Data," Chen, P., *ACM Transactions On Database Systems*, Vol.1, No.1, 1976.
[ChenW89]	"C-Logic of Complex Objects," Chen, W. D. and Warren, D. S., *Proc. of the ACM Symp. on Principles of Database Systems*, 1989.
[Codd79]	"Extending the Database Relational Model to Capture More Meanings," Codd, E. F., *ACM Transactions on Database Systems*, Vol.4, No.4, 1979.
[Copeland84]	"Making Smalltalk a Database System," Copeland, G. and Maier, D., *Proceedings of the ACM SIGMOD*, 1984.
[Ecklund87]	"DVSS: A Distributed Version Storage Server for CAD Applications," Ecklund, D. J., Ecklund, E. F. Jr., Eifrig, B. O. and Tonge, F. M., *Proceedings of International Conference on VLDB*, 1987.
[Emden76]	"The Semantics of Predicate Logic as a Programming Language," van Emden, M. and Kowalski, R., *Journal of the ACM*, Vol.23, No.4, 1976.
[Gallaire84]	"Logic and Databases: a Deductive Approach," Gallaire, H., Minker, J. and Nicolas, J. M., *ACM Computing Surveys*, Vol.16, No.2, 1984.
[Goldberg83]	*Smalltalk-80, The Language and its Implementation*, Goldberg, A. and Robson, D., Addison-Wesley, 1983.
[Hammer81]	"Database Description with SDM: A Semantic Database Model," Hammer, M. and Mcleod, D., *ACM Transaction on Database Systems*, Vol.6, No.3, 1981.
[Katz83]	"Managing the Chip Design Database," Katz, R. H., *IEEE Computer*, Vol16, No.12, 1983.
[Kifer89]	"Maier's Logic Revisited," Kifer, M. and Wu, J., *Proc. of the ACM Symp. on Principles of Database Systems*, 1989.
[Kifer89a]	"F-Logic: A Higher-Order Languages for Reasoning about Objects, Inheritance and Scheme," Kifer, M. and Lausen, G., *Proc. of the ACM SIGMOD International Conference on the Management of Data*, 1989.
[Kowalski78]	"Logic for Data Description," Kowalski, R., *Logic and Databases*, Nicolas, J. M., Gallaire, H. and Minker, J. (eds.), Plenum Press, New York, 1978.
[Kuper84]	"A New Approach to Database Logic," Kuper, G.M. and Vardi, M.Y., *Proceedings of the 3rd ACM Symposium on Principles of Database Systems*, 1984.

[Lloyd84] *Foundation of Logic Programming*, Lloyd, J. W., Springer-Verlag, Berlin, 1984.

[Lomet73] "An Operator Driven Model of Program Execution," Lomet, D. B., RC 4444, IBM Thomas J. Watson Research Center, 1973.

[Maier84] "Data Model Requirements for Engineering Applications," D. Maier and D. Price., *Proceedings of IEEE 1st International Workshop on Expert Database Systems*, 1984.

[Maier84a] "Databases in the Fifth Generation: Is PROLOG a Database Language?" Maier, D., *Proceedings of NYU Symposium on New Directions for Database Systems*, 1984.

[Maier85] "TEDM Data Model," Maier, D., *Unpublished Manuscript*, 1985.

[Maier86] "Development of an Object-Oriented DBMS," Maier, D., Stein, J., Otis A. and Purdy, A., *Proceedings of OOPSLA-86 Conference, 1986.*

[Maier86a] "A Logic for Objects," Maier, D., *Proceedings of the Workshop on Deductive Databases and Logic Programming*, 1986.

[Maier87] "Why Database Languages Are A Bad Idea?" Maier, D., *Workshop on Database Programming Languages*, Roscoff, France, 1987.

[Maier88] "Making Database System Fast Enough to Support CAD Applications," Maier, D., *Object-Oriented Concepts, Applications and Databases*, Kim, W. and Lochovsky, F. (eds.), 1988 (to appear).

[Mylopoulos80] "A language Facility for Designing Database-Intensive Applications," Mylopoulos, J., Bernstein, P. A. and Wong, C. K. T., *ACM Transactions on Database Systems*, Vol.5, No.2, 1980.

[Ohkawa87] "Mapping an Engineering Data Model to a Distributed Storage System," Ohkawa, H., *Research Paper*, Oregon Graduate Center, 1987.

[Rosenberg80] "The Evolution of Design Automation to Meet the Challenges of VLSI," Rosenberg, L. M., *Proceedings of 17th Design Automation Conference*, 1980.

[Shipman81] "The Functional Data Model and the Data Language DAPLEX," Shipman, D. W., *Transactions On Database Systems*, Vol.6, No.1, 1981.

[Sidle80] "Weakness of Commercial Database Management Systems in Engineering Applications," Sidle, T. W., *Proceedings of 17th Design Automation Conference*, 1980.

[Stonebraker86] "Inclusion of New Types In Relational Data Base Systems," Stonebraker, M., *Proceedings of IEEE Data Engineering*, 1986.

[Stonebraker87] "The Design of POSTGRES Rules System," Stonebraker, M., *Proceedings of IEEE Data Engineering*, 1987.

[Stroustrup86] *The C++ Programming Language*, Stroustrup, B., Addison-Wesley, New York, 1986.

[Su83] "SAM*: A Semantic Association Model for Corporate and Scientific-Statistical Databases," Su, S. Y. W., *Information Sciences*, 1983.

[Vbase86] *Vbase Users Manual*, Ontologic Inc., 1986.

[Zhu86] "Prototype Implementation and Storage Design for An Engineering Data Model," Zhu, J., *Research Paper*, Oregon Graduate Center, 1986.

[Zhu88] "Abstract Objects In An Object-Oriented Data Model," Zhu, J. and Maier, D., *Proceedings of 2nd International Conference of Expert Database Systems*, 1988.

[Zhu89] "Computational Objects In Object-Oriented Data Models," Zhu, J. and Maier, D., *Proceedings of 2nd International Workshop on Database Programming Languages*, 1989.