ABSTRACT OBJECT IN AN OBJECT-ORIENTED DATA MODEL

Jianhua Zhu and David Maier

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

Technical Report No. CS/E 87-015

Abstract Object In An Object-Oriented Data Model†

Jianhua Zhu and *David Maier*
Department of Computer Science and Engineering
Oregon Graduate Center
Beaverton, OR 97006

Abstract

This paper introduces *abstract object* as an extension to an object-oriented data model. An abstract object is essentially a template for regular database objects. It can be used for various kinds of matching in database queries and for the structural portion of type definitions. In a sense, abstract objects are similar to nonground terms in some logic systems, but with the ability to put structural constraints on the binding of a variable. We present abstract objects in the context of the TEDM object-oriented data model. We show how to represent database commands using abstract objects and investigate the semantics of pattern-matching using abstract objects. We then cover implementation techniques, uses for abstract objects other than pattern matching and ideas for extensions to the model.

1. Introduction

As the desire to extend database technology to wider application domains (notably engineering design environments) increases, it becomes apparent that traditional database systems are no longer sufficient [MP84]. Consequently, more data models are being invented every day, addressing the deficiencies of the conventional models. Intended for use in engineering design and similar environments, many of this new breed of data models share one important characteristic, the capability of constructing complex data objects that accommodate hierarchical structures, shared subparts or even cyclic data, a major departure from

† 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 first author is also supported by a Graduate Research Fellowship from Apple Computer Corp.

portability: the semantics of a programming system is ported along with a program.

The next section presents a brief overview of the TEDM data model. In Section 3, abstract objects are introduced and are integrated into TEDM, and examples of their use are also provided. Section 4 investigates the formal meanings of abstract objects. Section 5 discusses implementation issues. Section 6 explores other uses for abstract objects and ideas for extensions to the model. Conclusion and summary are given in Section 7.

2. TEDM Overview

This section provides a brief overview of TEDM, an object-oriented database model. Databases under TEDM are collections of *objects*, the basic building blocks provided by the model. Objects in TEDM are either *simple* or *complex*. A simple object is a nondecomposable atomic value, such as a *string* or a *number*. A complex object has internal *state*, which is made up of a collection of *fields*. Fields are given labels and therefore their order in a complex object is immaterial. Field values in turn are either simple objects or complex objects. In this way, arbitrary nested data objects can be constructed.

A second basic notion in TEDM is that of *object identity*. Objects possess *object identifiers* (OBID's) internally. The OBID of an object is unique with respect to the entire database, and it

---

```
RS:RectSelect(rect → R:Rectangle
                       (origin → O:Point(x → #0,
                                         y → Y:Number #1),
                        corner → C:Point(x → X:Number #4,
                                         y → #3)),
              cursor → U:Point(x → X, y → Y))
```

Fig.1 A TEDM Object Expression

---

is made up of ten subobjects — one Rectangle object, three Point objects and six Number objects, where RectSelect, Rectangle, Point and Number are type names. (Also notice the substructure sharing.) The RectSelect object is depicted pictorially in Figure 2 along with a structured diagram of the objects involved.

Some comments on the notation — numbers are prefixed by a sharp sign (#), symbols preceding the right arrow (→) are field labels, symbols preceding the colon are object tags, symbols following the colon are type names. Our convention will be to have fields labels in lower case and object tags and type names capitalized. Sharing of substructure is indicated by using the same tag. The idea of substructure sharing is better illustrated in Figure 3,

---

RS:RectSelect(rect → R:Rectangle
(origin → O:Point(x → #0,
y → #1),
corner → C:Point(x → #4,
y → #3)),
cursor → C)

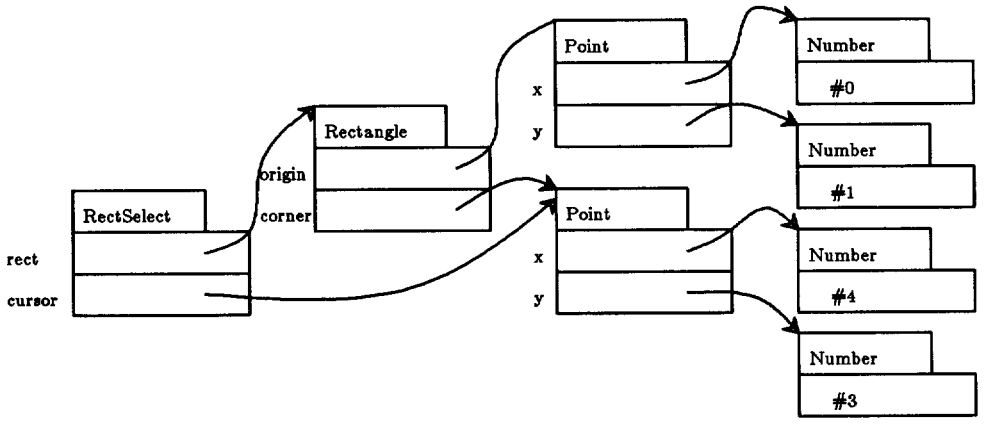

Fig.3 Another RectSelect Object

---

set of objects that belong to this type.

The types of the database form a *type hierarchy* used for organizing application data. This type hierarchy can be characterized as follows. The structural description is *prescriptive*: it gives the minimal structure for any object that is a member of the corresponding typeset. An object in a type may possess fields beyond those required by the intention. Membership in a typeset is not automatic upon an object fitting the structural description for the type. Rather, objects are explicitly added to or removed from types. Furthermore, an object may belong to multiple types. The *subtype* relation between types means the subset relation between typesets. The subtype relation is *declarational* rather than *structural*, meaning that a type is not treated as a subtype of another type unless such a subtype relationship has been explicitly declared or this relationship can be deduced by the transitivity of other declarations. However, this subset interpretation of the subtype relationship imposes constraints between the structural descriptions of a supertype and a subtype. A subtype must possess all fields of its immediate *supertype*, the type with respect to which the subtype declaration is made. A subtype may have more than one immediate supertype. Predefined types come with the system for simple values: for example String and Number. A comprehensive type ALL, which does not have any structural restrictions, is also provided as the root of the type hierarchy. The typical procedure for developing application databases is to extend this type hierarchy downwards, adding subtypes and adding objects to these types.

Point = (x ⟶ Number, y ⟶ Number)
Rectangle = (origin ⟶ Point, corner ⟶ Point)
RectSelect = (rect ⟶ Rectangle, cursor ⟶ Point)

Fig.4 Type Definitions

---

side describes virtual fields for objects that are assumed present for other queries. In Figure 5, there is a "leftside" field, Line object and Point object for each matching binding on the right. (The "*" stands for an arbitrary new object of the proper type.) For a command (<==), for each binding of the right side, the matching objects are modified to conform to the term in the head. In Figure 5, the "x" and "y" fields of the cursor point are modified. Some shorthand we use is to omit object variables when they aren't repeated (":Point") and omitting the type name for a variable in the head of a rule or command that appears on the right of the rule or command.

More detailed description of this data model is given in [Ma85]. Its formal logic is presented in [Ma86], where *O-Logic* is developed to provide formal semantics for the data model. Roughly, *O-Terms* correspond TEDM pattern terms described here, and *O-Formulas* are built up from O-Terms using usual logical connectives and quantifiers. Under that formalism, the models for O-Formulas turn out exactly to be those TEDM objects that make the O-Formulas true. A main memory prototype of TEDM using MProlog [Te85], which includes most of the features described here, has been carried out and its description can be found in [Zh86].

3. Adding Abstract Objects to TEDM

Given databases with collections of objects of the form described in the previous section, we naturally would want to retrieve information from them and make changes to them, with minimal human effort. Thus, we need a powerful query language. There have been various proposals for query languages on databases that support complex objects, such as Zaniolo's extension to the relational algebra [Za85], Kuper and Vardi's logic data model[KV84], Bancilhon and Khoshafian's object calculus [BK86] and Beeri's object logic[Be87], to just list a few.

The approach taken by TEDM is inspired by the computation mechanism used in logic programming systems, namely *unification.* The data retrieved from TEDM databases are presented in the workspace to the user as a set of *answer substitutions,* which are similar to pairs of variables and their bindings in programming languages. However, here the variables are bound to object

semantically. Instances of the former are a lexical means of defining concrete data objects. Instances of the latter specify patterns for matching against the data objects.

Traditionally, data objects are stored in the permanent databases, while queries are processed interactively during a work session, or preprocessed into DB calls when embedded in a program. As such, queries are separate from and are not described by the underlying data models. View definitions are usually supported by database systems as add-on features of the database languages and are processed outside the scope of the data model. From a practical point of view, a disadvantage of the traditional approach is that it is difficult to build systems that support a generalized (in the sense of more than some kind of command recall mechanism) query reuse scheme, nor is it easy to define a query procedure in one session and use it in another, or to combine queries.

TEDM tackles the problems by allowing query entities to be stored in the permanent databases as well. The approach is different from that taken by Stonebraker et al. [SAHR84] to extend INGRES to include QUEL commands as attribute values. That approach stores the textual form of a query in a field (and possibly caches a compiled form of a query). By contrast, we will have structured database objects that represent queries and commands. A key component of query objects will be a new flavor of objects to represent pattern terms. Such objects will be called *abstract* objects, as contrasted to the *concrete* objects introduced previously. Abstract objects will have physical representations in the database, just as concrete objects, but are given different interpretations by the database system. Having abstract objects in the database would be much like having the capability of storing a logic variable whose scope is multiple clauses in a logic database. In TOE's, we use a question mark (?) in place of a colon to indicate an abstract object of a given type.

As an example, Figure 7 presents a TOE for a stored abstract object and a diagram of its structural representation. Abstract objects are depicted there using double boxes. Such a stored abstract object can be used as a template in an operation. When so used, it will successfully match concrete objects of type RectSelect that have the specified internal structures, giving as result of the matching a tuple of abstract-concrete object pairs.

of type Command using TOE, as shown in Figure 8 and in Figure 9 respectively. We point out that it is possible for concrete objects to reference abstract objects, and vice versa. A Rule or Command object has three fields, for the head, the body and the bindings of variables to parts of the head or body. The values of the head and body fields are abstract objects. (The body could actually contain multiple abstract objects for pattern terms. TEDM allows multiple-occurrence fields.) The bindings field occurs once for each variable. Its value is an OVar object, which gives the name of the variable, the abstract object in the body it gets its binding from through matching, and the abstract object in the

---

```
:Rule(rulehead → ReObj?Rectangle
                 (leftside → :Line(p1 → PoObj?,
                                   p2 → :Point(x → Xcrd?,
                                               y → Ycrd?))),
      rulebody → DBRoot(rects →
                 (ReObj?Rectangle
                    (origin → PoObj?Point(x → Xcrd?Number),
                     corner → :Point(y → Ycrd?Number)))),
      bindings → :OVar(name → 'ReObj',
                       matches → Reobj?Rectangle,
                       makes   → Reobj?Rectangle)
                 & :OVar(name → 'PoObj',
                       matches → PoObj?Point,
                       makes   → PoObj?Point)
                 & :OVar(name → 'Xcrd',
                       matches → Xcrd?Number,
                       makes   → Xcrd?Number)
                 & :OVar(name → 'Ycrd',
                       matches → Ycrd?Number,
                       makes   → Ycrd?Number))
```

Fig.8 Rule expressed as TOE

RS?RectSelect(rect → R?Rectangle
(origin → OR?Point(x → X!, y → Y!),
corner → CO?Point),
cursor → CU?Point(x → X!, y → Y!))

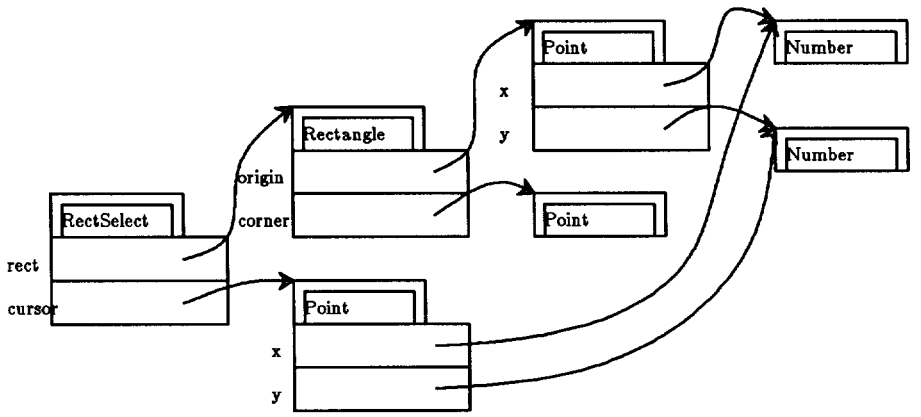

Fig.10 Indicating Identity in TCL

but O2 as a template does not match O1. For purposes of top-down evaluation of rules, we are looking at a unification operation, under which, for example, O1 above would match with

P3?Point(x → #4, y → M?Number)

Since we have abstract objects in the database, we are faced with the interesting problem of how to specify pattern matching against abstract objects and how to make use of this kind of matching. We will discuss the formal semantics of abstract-concrete matching in later sections. Here we discuss how to specify it in a command. We could introduce "meta-abstract" objects that serve as templates for abstract objects, but we would then need "meta-meta-abstract" objects as templates for those, and so on, ad naseum. Instead we include an "absMatches" field, holding an abstract object O1 in OVar objects to indicate that the variable should be bound to an abstract object O2 that matches

the universe. Our definitions will impose further structure on this universe.

Definition 2 (see [Ma86] for a similar definition): **TSU**, the TEDM structured universe, is a three-tuple (**U**, **g**, **t**), where $\mathbf{U} = \mathbf{D} \cup \mathbf{W}$ is the collections of entities, $\mathbf{g}: \mathbf{F} \rightarrow 2^{\mathbf{U} \times \mathbf{U}}$ is the label-interpreting function, and $\mathbf{t}: \mathbf{W} \rightarrow 2^{\mathbf{T}}$ is the type-assignment function. □

In this definition, **D** is the set of *atomic* entities (numbers and strings in our examples), and **W** is the set of *compound* entities. We further decompose **W** as $\mathbf{E} \cup \mathbf{X}$, where **E** is the set of *compound concrete* entities and **X** is the set of *abstract entities.* Recall F is the set of field labels.

By saying a binary relation is (not) defined at x, we mean there is (no) y such that the tuple (x, y) is in the relation. The function **g** takes a label and produces a relation. To prevent

---

Let $\mathbf{D} = \mathbf{N} \cup \mathbf{S}$,
$\mathbf{W} = \mathbf{E} \cup \mathbf{X} = \{e_1, e_2, ...\} \cup \{x_1, x_2, ...\}$,
$\mathbf{U} = \mathbf{D} \cup \mathbf{W}$,
$\mathbf{F} = \{$rect, cursor, origin, corner, x, y$\}$,
$\mathbf{T} = \{$Number, String, RectSelect, Rectangle, Line, Point$\}$,
$\mathbf{g} = \{$(rect, $\{(x_1, x_2), (e_2, e_3)\}$), (cursor, $\{(x_1, x_3), (e_2, e_6)\}$),
(origin, $\{(e_3, e_4)\}$), (corner, $\{(x_2, e_1), (e_3, e_5)\}$),
(x, $\{(e_1, \#4), (e_4, \#0), (e_5, \#4), (e_6, \#4)\}$), (y, $\{(e_1, \#3)$,
$(e_4, \#1), (e_5, \#3), (e_6, \#3)\}$)$\}$,
$\mathbf{t} = \{$(i, Number)$\} \cup \{$(s, String)$\} \cup \{(e_1$, Point), $(x_1$, RectSelect),
$(x_2$, Rectangle), $(x_3$, Point), $(e_2$, RectSelect), $(e_3$, Rectangle),
$(e_4$, Point), $(e_5$, Point), $(e_6$, Point)$\}$,
for all $i \in \mathbf{N}$ and $s \in \mathbf{S}$,
then (**U**, **g**, **t**) is a TSU

Fig.11 An Example TSU

$$o_e = (e, \{y \mid y \in \mathbf{U} \text{ and } (e, y) \in \mathring{\mathbf{R}}_{\mathbf{F}}\}).$$

The entity e is said to be the *root entity* of the object, and all other entities are said to be *subentities* of the object. We write an object as "root entity: set of subentities." □

Thus an object includes both a root entity that provides the identity of the object and a number of subentities that are properties or subcomponents of the object. Atomic objects have atomic root entities and no subentities. We will often confuse the root identities with the objects themselves. Two examples of TEDM object (see Figure 11) are given in Figure 12.

Definition 5: Given an $e \in \mathbf{U}$, the *incident set* of e is defined as

$$i_e = \{l \mid l \in \mathbf{F} \text{ and } \mathbf{g}(l) \text{ is defined at } e\};$$

the *reference set* of e is defined as

$$r_e = \{y \mid y \in \mathbf{U} \text{ and } (e, y) \in \mathbf{R}_{i_e}\}.$$

□

Namely, the incident set of an entity consists of all labels that the entity uses to reference other entities, and the reference set consists of all entities referenced. For example, $i_{x_1} = \{\text{rect}, \text{cursor}\}$ and $r_{x_1} = \{x_2, x_3\}$ (see Figure 11). This definition allows us to discuss the most closely related properties of objects while ignoring the remote ones.

Definition 6: Given an $e \in \mathbf{U}$, we say that o_e is an *abstract* object if $e \in \mathbf{X}$; otherwise it is a *concrete* object. □

For example, of the two objects we gave previously in Figure 12, o_{e_3} is a concrete object and o_{x_2} is an abstract object. Collectively, we will use **O** to denote all objects, **C** to denote all concrete objects and **A** to denote all abstract objects.

4.2. Object Isomorphism

The concept of object isomorphism is the theoretical basis for various kinds of matching. The idea is that two objects match if one can be mapped into another according to certain property-preserving criteria.

3). if $(c_1, d_1) \in \mathbf{g}(l)$, for some $l \in \mathbf{F}$, then $(c_2, d_2) \in \mathbf{g}(l)$

□

In other words, isomorphic mappings preserve the root, the type hierarchy and the labels. Notice we allow atomic entities to map to abstract entities and vice versa, but disallow this mapping between atomic entities and concrete entities. We say that an object o_1 is *embedded* in an object o_2 if o_1 is isomorphic to some initial segment of o_2. For example (see Figure 13 and Figure 12), i_1 is isomorphic to i_2 under an obvious isomorphism

$$\alpha = \{(x_2, e_3), (e_1, e_5), (\#3, \#3), (\#4, \#4)\},$$

and therefore o_{x_2} is embedded in o_{e_3}.

It can easily be shown that the system of initial segments and isomorphisms form an algebraic structure that possesses some nice properties, such as identities and compositions; but we do not go into details here.

Definition 9: Given two isomorphisms α and β, we define the *combination isomorphism* of α and β as $\alpha \cup \beta$, provided $\alpha \cup \beta$ is also a mapping; otherwise their combination is undefined. □

The concept of combination isomorphism can be extended to the case where more than two isomorphisms are involved in an obvious way. The combination isomorphism always exists for a collection of isomorphisms whose domains are pairwise disjoint.

4.3. Object Matching

Of all possible isomorphic mappings, we are mostly interested in those that go from abstract objects to initial segments of objects. Namely, we are interested in the situation where abstract objects can be embedded into objects.

Definition 10: Given an abstract object $o_1 = e_1{:}E_1$ embedded in an object $o_2 = e_2{:}E_2$ under α, we say that o_1 *matches with* o_2 if the image of o_1 under α contains no abstract objects and

$$\forall d_1 \in E_1\ d_1 \in \mathbf{D} \cup \mathbf{E} \Longrightarrow \alpha(d_1) = d_1.$$

□

```
rel(o2, origin, o4)
rel(o2, corner, o5)
rel(o3, $in, 'Point')
rel(o3, x, 4)
rel(o3, y, 3)
rel(o4, $in, 'Point')
rel(o4, x, 0)
rel(o4, y, 1)
rel(o5, $in 'Point')
rel(o5, x, 4)
rel(o5, y, 3)
```

Note that "$in" is a predefined field label. The values for "$in" fields are object identifiers for proper type defining objects. (But we will use type names in examples to illustrate ideas.)

Interactive user queries are translated into Datalog (i.e. Prolog without functors) queries and are then executed. For example, the pattern term

RS:RectSelect(rect → R:Rectangle, cursor → P:Point)

would be translated into the following query:

```
?- rel(RS, $in, 'RectSelect'),
   rel(RS, rect, R),
   rel(R, $in, 'Rectangle'),
   rel(RS, cursor, P),
   rel(P, $in, 'Point').
```

The object handler is being implemented in MProlog and is being interfaced to a secondary storage facility (namely the storage manager). The object handler is made up a number of smaller modules. The central module is an *execution module* that controls the data flow between the object handler and the storage manager and carries out appropriate operations, as dictated by the current execution state, on the workspace objects (either permanent data or temporary data). Under this execution module, there are other modules that coordinate communications between the object handler and the storage manager. On top of the execution module, there will be processing-modules for TEDM query expressions. And at the highest level there is an input module that parses interactive user command, and directs them to processing-modules for further translation.

type cacculus has been devised for describing these requirements. We will take the approach in which abstract objects play a major role for devising the meanings of type definition expressions. In what follows, we assume the existing types are already represented as abstract objects, and show how new types can be defined and represented as abstract objects. This kind of reasoning is acceptable since we can always bottom out at system predefined types.

6.1.1. Cartesian Product Types

In this form of type definition, a new type is defined by explicitly giving its structures, and an abstract object is created to specify the structures. That abstract object is a part of a *type-defining* object, which gives the name and position in the type hierarchy, as well as the minimum structure on its instances. What we really have here is another language (type definition language or TDL) for describing type-defining objects. For example, the following expression evaluates to a type object representing a type named Rectangle,

```
Rectangle = (origin → Point,
             corner → Point)
```

The resulting type-defining object can be described in TOL as

```
:TypeDef(typeName → 'Rectangle',
         supertype → 'All',
         structure → R?Rectangle(origin → P1?Point,
                                 corner → P2?Point))
```

6.1.2. Equivalent Types

The simplest way to define a new type is by saying it is structurally equivalent to an existing type. For example, to define a new type, Box, such that it has the same structural constraints as the type Rectangle, we use "Box = Rectangle". The effect of such a definition is to set the "structure" field of the type-defining object for Box to be a copy of the "structure" field of the type-defining object for Rectangle. Thus the result of this type expression is

```
:TypeDef(typeName → 'Box',
         supertype → 'All',
         structure → B?Box(origin → P1?Point,
```

The result of this type definition is again an abstract object:

```
:TypeDef(typeName → 'RectSelect',
         supertype → 'Rectangle',
         structure →
           RS?RectSelect(origin → P1?Point,
                         corner → P2?Point,
                         cursor → P3?Point))
```

6.1.5. Multiple Inheritance

In a type definition, more than one type can be specified as supertypes of the type being defined, and the new type will inherit the union of the structures of each individual supertype. (We assume no name clash can occur.) For example, the following definition defines a type called DesignDoc, which presumably models documenting digital system design project,

```
DesignDoc = ArchitectDoc, FunctionDoc,
            LogicDoc, CircuitDoc
              (designDocNo → Number)
```

That is, The DesignDoc type has structures of those of ArchitectDoc, of FunctionDoc, of LogicDesign and of CircuitDoc, plus an additional field "designDocNo". The resulting "structure" field in the type-defining object is, schematically, the following:

```
DD?DesignDoc(<fields-from-ArchitectDoc>,
             <fields-from-FunctionDoc>,
             <fields-from-LogicDoc>,
             <fields-from-CircuitDoc>,
             designDocNo → N?Number,
```

6.1.6. General Type Calculus

The type definition constructs outlined above can be composed in a very flexible way, yet still maintaining clean semantics. For instance, a type for modeling mailing addresses can be defined as follows:

```
MailAddress = (stNumber → Number,
               stName → String,
               cityName → String,
               state → (stateName → String,
                        postalCode → String),
```

A serial composition of commands, C_1, C_2, ... , C_n takes the form of C_1; C_2; ... ; C_n. Each command in this composition is executed in sequence. If any of the individual commands fails in execution (due to pattern matching failure), the execution of the composition fails.

A parallel composition of commands takes the form of C_1 || C_2 || ... || C_n. Command in this composition is executed one by one until a successful execution is obtained. The result of the first command that is executed without failure is taken to be the result of the parallel composition. If none of commands participating in a parallel composition succeeds, the execution of the composition fails.

6.3. Computational Objects

Computational objects are an effective means for dealing with infinite types and infinite objects. Some examples are:

```
AddType = (addend1 → N1:Number,
           addend2 → N2:Number,)
        ⟹ add[N1, N2]

InfSequence = (start → N:Number)
            ⟹ cons[N, InfSequence[succ[N]]]
```

Some explanations are needed. First, we use the reduction symbol ⟹ to indicate a computational type. Expressions following ⟹ specify reduction rules for computational types. Second, some reduction symbols, such as add, succ and cons, are predefined. Third, member objects of a computational type are those derivable via the reduction rules of the type.

The G-machine architecture [Jo83] for graph reduction from functional programming research area seem to be a good candidate for evaluating computational objects. The way in which the graph reduction approach represents computations closely resembles how object structures are represented, namely using directed graphs. For example, the application graph for expression InfSequence[1] can be described as the following TEDM structure

```
:Apply(function → 'InfSequence'),
       argument → #1)
```

5. As an intermediate step towards a completely "objectified" language, a simple interface between a programming language and the database system allows construction and evaluation of command objects, without major modification to the programming language itself. Adding new query functionality—range selection, equality vs. identity comparisons, aggregates—involves creating new flavors of abstract objects, but not changes to the programming language parser. Ultimately, however, we want to do away with any "surface" language, and have entire programs represented as objects, such as in the Garden system [Re86].

6. With commands as objects, we can consider interpretations of those objects other than just their execution value. Abstract interpretations can be defined for them to give valuations in domains such as execution time or result size. As Atkinson et al. point out, the compilation or optimization of such an object is just a particular view on the object [AMP87].

There are also a number of extensions and refinements to explore:

1. Is it possible to develop an abstract object notion based on protocol rather than structure? Such an object might be viewed as a computation graph to be evaluated via graph reduction techniques, with an added reduction rule for database matching. The result of a reduction sequence would be nondeterministic, because an abstract object can match the database in multiple ways.

2. For a command object, what are strategies for evaluating portions of it on different processors? For example, the structural access could be done on a central storage server, and the computational part on a local workstation.

3. We don't think abstract objects are quite equivalent to logical variables. We think objects with logical variables would be useful for expressing and constraining partially defined objects and for representing alternative configurations or versions of an object. Perhaps the ability to store a name from a binding environment in place of a value would give equivalent power [AM85].

Principles of Database Systems, 1984.

[MP84] *Data Model Requirements for Engineering Applications*, D. Maier. and D. Price, IEEE 1st International Workshop on Expert Database Systems, 1984.

[Ma85] *The TEDM Data Model*, D. Maier. Working Paper, Oregon Graduate Center, 1985.

[Ma86] *A Logic for Objects.* D. Maier. Proceedings of the Workshop on Deductive Databases and Logic Programming, 1986.

[Ma87] *Why Database Languages Are A Bad Idea?.* D. Maier. Workshop on Database Programming Languages, Roscoff, France, September 1987.

[MSOP86] *Development of an Object-Oriented DBMS.* D. Maier, J. Stein, A. Otis and A. Purdy. OOPSLA-86 Conference Proceedings, 1986.

[Oh87] *Mapping an Engineering Data Model to a Distributed Storage System.* H. Ohkawa. Ph.D. Research Proficiency Paper, Oregon Graduate Center, 1987.

[Re86] *An Object-Oriented Framework for Graphical Programming.* S.P. Reiss. Department of Computer Science, Brown University, 1986.

[RS85] *The MR Diagram — A Model for Conceptual Database Design.* R. Ramakrishnan and A. Silberschatz. Proceedings of VLDB, 1985.

[SAHR84] *QUEL as a Data Type.* M. Stonebraker, E. Anderson, E. Hanson and B. Rubenstein. Proceedings of SIGMOD, 1984.

[Te85] *4400P31 Prolog Programmer's Manual*, Tektronix Incorporation, 1985.

[Za85] *The Representation and Deductive Retrieval of Complex Objects.* C. Zaniolo. Proceedings of VLDB, 1985.

[Zh86] *Prototype Implementation and Storage Design for An Engineering Data Model*, J. Zhu. Ph.D. Research Proficiency Paper, Oregon Graduate Center, 1986.

[Zh87] *The Notion of Abstract Object in an Engineering Data Model*, J. Zhu. Ph.D. Thesis Proposal, Oregon Graduate Center, 1987.

8. References

[AM85] *Types, Bindings and Parameters in a Persistent Environment.* M.P. Atkinson and R. Morrison. Proceedings of Data Types and Persistence Workshop, Appin, 1985.

[AMP87] *Persistent Information Architectures.* M.P. Atkinson, R. Morrison and G.D. Pratten. Persistent Programming Research Report 36, University of Glasgow/University of St. Andrews, 1987.

[AN85] *LOGIN: A Logic Programming Language With Built-In Inheritance*, H. Ait-Kaci and R. Nasr. MCC Technical Report, AI-068-85, 1985.

[AEM86] *PROTEUS: Objectifying the DBMS User Interface*, T.L. Anderson, E.F. Ecklund, Jr and D. Maier. Proceedings of the International Workshop on Object-Oriented Database Systems, 1986.

[BanK85] *A Model of CAD Transaction*, F. Bancilhon and W. Kim. Proceedings of VLDB, 1985.

[BatK85] *Modeling Concepts for VLSI CAD Objects*, D. Batory and W. Kim. ACM Transactions on Database Systems, September 1985.

[BBDMR84] *CAD/CAM Database Management.* M.L. Brodie, B. Blaustein, U. Dayal, F. Manola and A. Rosenthal. Database Engineering, June 1984.

[Be87] *On Combining Object Orientation and Logic Programming.* C. Beeri. XP8.5i Workshop, Oregon Graduate Center, 1987.

[BK86] *A Calculus for Complex Objects.* F. Bancilhon and F.S. Khoshafian. Proceedings of ACM Symposium on Principles of Database Systems, 1986.

[EEET87] *DVSS: A Distributed Version Storage Server for CAD Applications.* D.J. Ecklund, E.F. Ecklund, Jr. B.O. Eifrig and F.M. Tonge. Proceedings of VLDB, 1987.

[Jo83] *The G-machine: an Abstract Machine for Graph Reduction.* T. Johnsson. Proceedings of the Declarative Programming Workshop, 1983.

[Ka83] *Managing the Chip Design Database.* R.H. Katz. Computer, December 1983.

[KV84] *A New Approach to Database Logic.* G.M. Kuper and M.Y. Vardi. Proceedings of the 3rd ACM Symposium on

where Apply is a predefined computational type.

Type objects for computational types are represented as a sequence of G-machine instructions obtained through compiling computational types that, when executed, would carry out desired reduction. The following design offers one possible solution:

Operator = {push, pushfun, pushint, pushbool,
pushnil, update, ...}
Instruction = (operator → Operator,
operand → All)

A more powerful TEDM command execution engine is needed. In particular, a reduction engine is necessary to evaluate computational objects, which could be a G-machine. Command execution strategy should also be modified accordingly, as outlined in [Ma87]. Instead of a matching phase followed by an action phase, we need a reduction phase that comes before the action phase. The task of the reduction phase is to pick up all pending computations, and conceivably create temporary structures to hold the intermediate results.

7. Concluding Remarks

We have presented abstract object as an extension to an object-oriented database model, and discussed their uses, semantics and implementation. We list some advantages that accrue from using abstract objects as the building blocks of database commands:

1. Commands can be stored in the database, making them easy to catalog and accessible from multiple applications. Moreover, we now have the possibility that two commands could share the same abstract object as a subpart.

2. Queries can have arbitrary literals, not just those with lexical conventions for representation. We can write a query that looks for a Rectangle containing a certain Point, without having to describe the Point by its state.

3. Expressing cyclic query structures is possible.

4. Commands can be viewed and edited with whatever mechanism exists for manipulating regular database objects.

```
                    zipCode ⟶ Number)
```

which results in the following "structure" field in its type-defining object:

```
MA?MailAddress(stNumber ⟶ N1?Number,
               stName ⟶ S1?String,
               cityName ⟶ S2?String,
               state ⟶ I1?InternalTypeX
                        (stateName ⟶ S3?String,
                         postalCode ⟶ S4?String),
               zipCode ⟶ N2?Number)
```

Notice an internal type InternalTypeX has been created to help describe the MailAddress type object.

6.2. Compound Commands

We described simple commands in TCL previously. We show in this subsection that simple commands can be composed, and we present TCL extensions to describe compound commands.

A compound command is a user defined TEDM command procedure that manipulates database objects based on somewhat higher level semantics. Compound commands are composed of a number of simple commands. Command composition can be done in two ways — *serial composition* and *parallel composition.* Besides being a mechanism for grouping individual commands to perform meaningful operation, compound command also provides the opportunity of sharing variable bindings across individual commands. For example, one could come up with a compound command that consistently carries out necessary changes to a Person object to hire the person:

```
HireEmployee[N:PersonName, D:Department, S:Salary] =
{
  LocalVar[P:Person];
  P:Person(department ⟶ D, salary ⟶ S) <==
     persons ⟶ P:Person(personName ⟶ N);
  emps ⟶ P <==
}
```

where LocalVar is a system provided command that introduces local variables.

corner → P2?Point))

It should be noted that two equivalent types are guaranteed to have the same structures only at the time of definition. After the definition, any one of them can change freely without affecting the other.

6.1.3. Maximum Subtypes

Another simple way of defining a new type is to declare it as a subtype of an existing type. For example, "Square < Box" defines a new type Square that is to have the same structure requirements of type Box and is to be treated as a subtype of Box to start with. Types defined this way are maximum subtypes of corresponding supertypes in the sense that if any of the structures is dropped from a new type, the subtyping discipline of TEDM is violated. Such type definitions create type-defining objects in the same way as equivalent type definitions, with the created abstract objects having one additional meta structure that takes the supertype objects as values, capturing the subtyping information. Therefore, the resulting type object is

```
:TypeDef(typeName → 'Square',
         supertype → 'Box',
         structure → B?Box(origin → P1?Point,
                           corner → P2?Point))
```

If Square is also an existing type, the effect of the definition "Square < Box" is to make Box as its supertype, provided typing system constraint is not violated.

6.1.4. General Subtypes

In general, a new type can be defined as having the structure requirements of an existing type plus certain additional requirements. Types defined this way are subtypes of corresponding supertypes. An example of where this kind of definition is useful would be an alternative RectSelect type that has three Point components, modeling the origin point and corner point of a rectangle, and a cursor point selecting regions inside the rectangle. The RectSelect type can be defined as follows:

RectSelect = Rectangle(cursor → Point)

An critical performance issue is efficient pattern matching against large database that resides on secondary storage. Our current design incorporates many storage structuring techniques such as clustering and duplicating, that should reduce the number of disk accesses needed for processing queries. In order to reduce the need for in-memory transformations, we use *fragments*, which are main memory chunks shared by the object handler and the storage manager. Fragments provide storage space for initial segments of objects. All initial segments in a fragment have the same format, which is dynamically defined. In order to stretch the bandwidth of communication between the two layers, we use *bulk-load* as much as possible. In the bulk-load mode, the storage manager is given some loading criteria and a format, and it searches the database and loads appropriate triples to form fragments using the format provided. Since the format for each bulk-load is fixed and is known to the object handler, access to individual fields can be compiled to use a starting address and an offset directly, eliminating the overhead of matching against the field labels. In the case of multiple occurrence fields, a pointer is planted in place of the field value, leading to a separate region where the multiple occurrence values are located. Other relational query processing techniques, such as building optimized query plans, can also be applied to bulk-load operation. This prefiltering processing by the storage manager avoids much of the unnecessary traffic to the object handler, and should lead to a fairly efficient implementation.

6. Type, Command and Computation Objects

We proposed earlier using abstract objects to store commands in databases — the results are databases with data objects and commands manipulating these data objects. This way we can build self-contained, application-specific databases. In this section, we show by examples how the rest of the data model proper can be captured nicely using abstract objects. We will also look into possibilities of extending the concepts to introduce computations into the data model.

6.1. Type Definitions

Typing in TEDM is such a mechanism that puts minimum requirements on structures of objects. A flexible yet expressive

That is, for two objects to match, the isomorphism has to be able to make their atomic data values and concrete objects correspond. Consequently, as a precondition, ones data value set has to be a subset of the other's data value set. Therefore, the term "pattern matching" we use in the syntactic domain to describe the query facility of the database language corresponds to a special kind of isomorphism in the semantic domain. In Figure 12, o_{x_2} matches with o_{e_3}.

Definition 11: Given object o_1 matches with object o_2 under α, then the *answer substitution* of the matching is defined as the maximal subset of α:

$$\mu = \{(e_1, e_2) \in \alpha \mid e_1 \in \mathbf{X}\}.$$

□

For example (see Figure 12 and Figure 13), the answer substitution for the matching of o_{x_2} with o_{e_3} is $\mu = \{(x_2, e_3)\}$. In general we are interested in the set of all answer substitutions.

5. Implementation

The plan is to implement the TEDM data model on top of a distributed object server DVSS [EEET87], which provides concurrency, recovery and distributing on variable-size byte objects. Architecturally, there are two functional blocks that implement the TEDM database system, an *object handler* and a *storage manager*, each implementing a mapping between layers of abstraction. The storage manager defines the mapping from the TEDM storage model (explained below) to the DVSS object model. More detailed discussion concerning this mapping can be found in [Oh87].

The object handler compiles TEDM objects into their internal representation using TEDM storage model, which can be described as rel(OBJECT_ID, FIELD_NAME, FIELD_VALUE)*, namely list of triples of the given format. For example, the object given in Figure 1 is translated into the following triple list:

```
rel(o1, $in, 'RectSelect')
rel(o1, rect, o2)
rel(o1, cursor, o3)
rel(o2, $in, 'Rectangle')
```

Definition 7: The set of *initial segments* of an object o_e = e: $\{e_1, e_2, \ldots\}$, I_e, is defined by:

1). e: $\varnothing \in I_e$;
2). e: $E \in I_e$ and $\exists\!\!\!-_{x \in E}\ (x, y) \in \mathbf{R_y}$, then e: $(E \cup \{y\}) \in I_e$
3). nothing else is in I_e.

□

Not every subset of an object is an initial segment of the object, (even though it is a necessary condition), since reachability from the root through entities is also needed. Also observe that any object is an initial segment of itself. Informally, initial segments are connected subcomponents that include the object root. Thus we also use the term "the root of an initial segment." We will use **I** to denote the collection of all initial segments, namely,

$$\mathbf{I} = \cup_{e \in U}\ I_e.$$

Three examples of initial segment (see Figure 12) are given in Figure 13.

Definition 8: Given $i_1, i_2 \in \mathbf{I}$, with $i_1 = e_1{:}E_1$ and $i_2 = e_2{:}E_2$, we say that i_1 is *isomorphic* to i_2 if there is a 1-1 onto mapping α: $i_1 \rightarrow i_2$, such that $\alpha(e_1) = e_2$ and if $c_1 \in i_1$ gets mapped to $d_1 \in i_2$ and $c_2 \in i_1$ gets mapped to $d_2 \in i_2$ by α, then the following conditions must be true:

1). $\mathbf{t}(c_2) \sqsubseteq \mathbf{t}(c_1)$;
2). if $c_1 \in \mathbf{D}$, then $c_2 \in \mathbf{D} \cup \mathbf{A}$, and vice versa;

$i_1 = o_{x_2}$ is an initial segment of o_{x_2}, and
$i_2 = e_3$: $\{e_5, \#3, \#4\}$ is an initial segment of o_{e_3}.
$i_3 = e_3$: $\varnothing$ is an initial segment of o_{e_3}.

Fig.13 Three Initial Segments

---

atomic entities from having substructures, we impose the restriction that the relations it produces not defined at any atomic entity. Namely,

$$\forall x \in \mathbf{D}\ \forall l \in \mathbf{F}\ \forall y \in \mathbf{U}\ ((x, y) \notin \mathbf{g}(l)).$$

Figure 11 provides an example of a TEDM universe. Notice **N** represents atomic integer values and **S** represents atomic string values and they are predefined.

Definition 3: $\mathbf{R}_F$, the *reference relation* relative to $F \subseteq \mathbf{F}$, is defined as $\bigcup_{l \in F} \mathbf{g}(l)$. $\mathring{\mathbf{R}}_F$, the *reachable relation* relative to F, is the reflexive transitive closure of $\mathbf{R}_F$. We also say that y is *F-reachable* from x if $(x, y) \in \mathring{\mathbf{R}}_F$, and use *reachable* to mean **F**-reachable. □

In other words, the reference relation specifies that one entity is referenced by the other, via a sequence of fields with labels from F. This reference relation is obtained from the union of certain relations produced by **g** by taking a closure. Normally, the subscript can be omitted when the set F is understood. For the example given in Figure 11, assuming F is **F**, **R** is

$$\{(x_1, x_2), (e_2, e_3), (x_1, x_3), (e_2, e_6), (e_3, e_4), (x_2, e_1),$$
$$(e_3, e_5), (e_1, \#4), (e_4, \#0), (e_5, \#4), (e_6, \#4), (e_1, \#3),$$
$$(e_4, \#1), (e_5, \#3), (e_6, \#3)\}.$$

Definition 4: Given an $e \in \mathbf{U}$, A TEDM *object*, o_e, is defined as a pair (e, the maximal set of entities that are reachable from e); namely,

---

$o_{e_3} = e_3$: $\{e_4, e_5, \#0, \#1, \#3, \#4\}$
$o_{x_2} = x_2$: $\{e_1, \#3, \#4\}$.

Fig.12 Two Objects

O1. In a TCL pattern term, we use "?" to indicate abstract matching. For example, for the pattern term

ptTemplates ⟶ P?Point(x ⟶ M?Number,
y ⟶ N:Number)

P will bind to abstract Point object in the ptTemplates set whose x coordinate is an abstract Number object and whose y coordinate is a concrete Number object. Such a matching object, expressed as TOE, might be

P1?Point(x ⟶ M1?Number, y ⟶ #6)

One further enhancement to TCL queries we want to support is to retrieve an abstract object O2 by matching, then use it immediately as a template itself, for matching other objects. The TCL syntax for such a pattern might be

ptTemplate ⟶ P?Point(x ⟶ M?Number,
y ⟶ N:Number),
points ⟶ PT:P

In the second pattern term, the binding for P from the first term becomes the template for binding for PT in the second term. We have yet to fix the format of a command object to specify this matching. Possibilities are letting the "matches" field in an OVar object reference another Ovar, or adding an "indirectMatches" field. We are also looking at ways to specify a query that selects an object that has a certain type of object as a subobject, at an unspecified level of nesting. Such a capability would be useful, for example, for indexing all commands that included a Point abstract object somewhere in a Pattern term.

4. Object Semantics

Having described TEDM concrete and abstract objects, we are in a position to investigate their semantics more formally, and we will do this from more or less an algebraic viewpoint, using the notion of *isomorphism* to describe the meanings of various kinds of matching.

4.1. Basic Definitions

Object expressions denote entities residing in permanent storage media. Thus, we take everything in permanent storage as

head to which we want make modifications. If an object variable occurs multiple places in the body, it does not represent identity among abstract objects. Rather, each occurrence gives rise to another pattern the variable must match, which are captured as multiple occurrences of the "matches" field. In inspecting Figure 8 and Figure 9, note that the named sets in pattern terms are really just fields in a special DBRoot object. We assume any database has only one concrete DBRoot object.

If it is desired that a repeated object variable in a command body represent just one abstract object, then we use a special "!" notation in TCL. Figure 10 shows such an example, where only relevant portion of a "cmdbody" field is shown along with its structural representation.

Note that we are doing matching here in the strict sense, and not unification. For example, the object O1 (In TOL)

P?Point(x → M?Number, y → #0)

matches object O2

P2:Point(x → #4, y → #0)

```
:Command(cmdhead → PoObj?Point(x → #2,
                               y → #2)),
         cmdbody → DBROOT(rectss →
                          RSObj?RectSelect
                            (rect → ReObj?Rectangle
                                      (corner → :Point(x → #4,
                                                       y → #3)),
                                    cursor → PoObj?Point)),
         bindings → :OVar(name → 'PoObj',
                          matches → PoObj?Point,
                          makes   → PoObj?Point))
```

Fig.9 Command Expressed as TOE

RS?RectSelect(rect → R?Rectangle
(origin → OR?Point(x → X?, y → Y?),
corner → CO?Point),
cursor → CU?Point(x → X1?, y → #1))

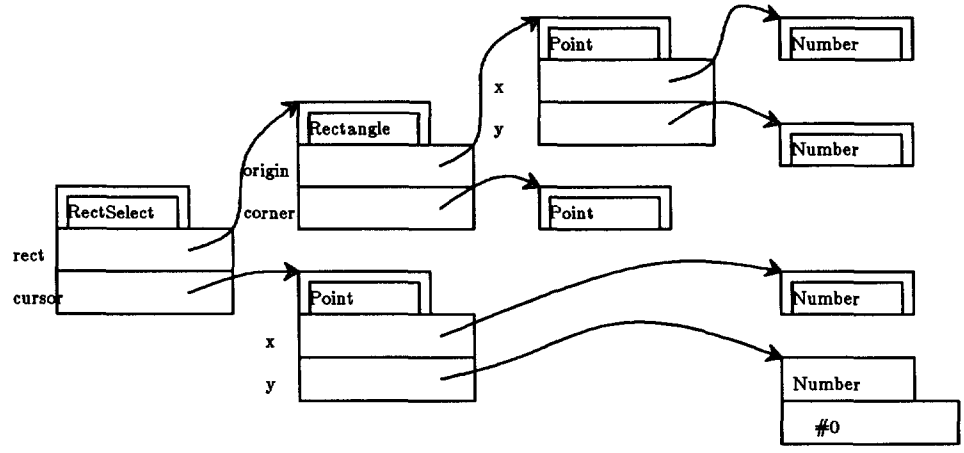

Fig.7 An Abstract Object

Actually, the use of abstract objects is much richer than merely specifying pattern matching templates. We can now easily represent database commands, rules and other language constructs as TEDM objects and store them along with application data, as was claimed earlier. We can view TEDM command language (TCL) as a succinct alternative to TOE for expressing command objects. (However TEDM command language is not as general as TEDM object language, or TOL, the language in which TOE's are written.) In TOE, object tags merely indicate interconnectivity of the object described. In TCL, we need to preserve the variables as actual objects, in order to express the propogation of bindings from matching concrete objects with the right side to making modifications to those concrete objects as specified on the left. For example, the rule and the update statement given using TCL in Figure 5 can be represented as TEDM objects of type Rule and

identifiers instead. Using the identifiers obtained from answer substitutions, objects in the database can be viewed and modified and then reinstated. In this context, object identifiers function much as l-values in programming languages. Figure 6 contains a TEDM query that asks for all RectSelect objects in the set "rects" whose cursor point coincides with their origin, plus their corner points. (If we wanted RectSelect objects with their cursor identical to their origin, the second part of the term would be "cursor → OR".) The answers to this query will be returned as 6-tuples of bindings of objects to the variables RS, OR, X, Y, CO and CU.

This example also illustrates several important differences between TEDM query terms and first-order terms (FOT's) as seen in, say, PROLOG. First, typing information is attached to the query, which can be used to restrict the range of the search and improve the processing efficiency. Second, fields are labeled, which in turn eliminates the stiff requirements on arguments of FOT's (namely, fixed arity and fixed position). Third, object variables are more general than variables of FOT's, as they permit retrieval of complex objects by partially specifying their internal structures, whereas a variable in a FOT is unrestricted in the substructure of what it matches.

Another point shown in this example is that syntactically query terms are similar to TOE's, except that query terms are allowed to partially specify the structure of what they match (whereas a TOE defines the entire states of an object.) Nevertheless, object expressions and query terms are quite different

---

```
rects → RS:RectSelect(rect → :Rectangle
                              (origin → OR:Point(x → X, y → Y),
                               corner → CO:Point),
                      cursor → CU:Point(x → X, y → Y))
```

Fig.6 A TEDM Query Expression

```
R(leftside ⟶ *:Line
                    (p1 ⟶ :P, p2 ⟶ *:Point( x ⟶ M, y ⟶ N)))
  ←
  rects ⟶ R:Rectangle(origin ⟶ P:Point(x ⟶ M),
                              corner ⟶ :Point(y ⟶ N))


P(x ⟶ #2, y ⟶ #2))
  <=
  rects ⟶ RS:RectSelect
                    (rect ⟶ :Rectangle
                                  (corner ⟶ :Point(x ⟶ #4, y ⟶ #3)),
                     cursor ⟶ P:Point)
```

Fig.5 TEDM Rule and Update Statements

Other features of this data model include deductive elements for virtual data derivation, a rule-like non-relational data language, and the capability of creating arbitrarily complex structures in a single data language statement. In Figure 5, we present an example of a rule statement and an example of an update statement. The rule defines a virtual field "leftside" for rectangles stored as origin and corner points. The update command moves the cursor point of RectSelect objects with a corner of (4, 3) to (2, 2), in particular, it moves the cursor of the RectSelect object given in Figure 1 to its center, by modifying the coordinates of the existing cursor point. A few words of explanation. The right side of a rule or command consists of one or more terms. A term in TEDM is syntactically similar to a TOE, except object tags are construed as object variables. These variables range over the objects in the database for purposes of matching. The term must be "anchored" by a named set of objects. Here, the "rects ⟶" prefix on the right side supposes a set named "rects" of objects against which the rest of the term is matched. The meaning of the whole rule or command is that for every assignment of objects to object variables, the left side must hold. In the case of a rule (←), the left

where the cursor and the corner of a RectSelect object happen to have the same Point object as their value.

We provide a formal definition to describe TEDM object expressions. The existence of several denumerable sets, **F**, **S**, **N**, **H** and **T** is assumed: where **F** is a set of symbols for field labels, **S** is the set of string values, **N** is the set of natural numbers (for the moment we only consider strings and natural numbers as atomic values), **H** is the set of object tags, and **T** is the set of symbols for types. We also define a partial order $\sqsubseteq$ on **T** based on a subtype/supertype relationship.

Definition 1: TOE, the collection of well formed TEDM object expressions(TOE's), is defined by the following rules:

1). S:String d $\in$ **TOE** for d $\in$ **S** and S $\in$ **H**;
2). N:Number d $\in$ **TOE** for d $\in$ **N** and N $\in$ **H**;
3). given $e_1, ..., e_n \in$ **TOE**, $f_1, ..., f_n \in$ **F**, T $\in$ **T** and H $\in$ **H**, then $H{:}T(f_1 \rightarrow e_1, ..., f_n \rightarrow e_n) \in$ **TOE**;
4). nothing else is in **TOE**.

□

The first two items in the definition describe simple objects, and the third describes complex objects. We point out that TOE's resemble Ait-Kaci's *Ψ-terms* [AN85], but the two differ in semantics. TOE's represent single, ground objects, while Ψ-terms denote types. There are some other consistency conditions on TOE's that we do not formalize here. The *type conformity condition* requires that there be no conflicts between types denoted by type symbols and objects tagged by object tags. The *tag repetition condition* disallows mismatches between object labels and object tags, e.g., had we used R instead of C to describe the value of the cursor point in Figure 3. Certain object tags and type symbols can be omitted when writing object expressions. For example, one need not include type symbols String or Number in an expression (as in the previous example), as these atomic values are self-describing for types.

Types in TEDM have *intentional* and *extensional* aspects. The intentional aspect is a structural description, i.e., a list of fields and their respective types. Examples of such descriptions are shown in Figure 4. The extensional ingredient is *typeset*, the

will not change during the lifetime of the object. When an object dies, its OBID also dies and will not be reused again. The OBID of an object and the state of the object are orthogonal — while the state may change as the database evolves, the OBID always stays the same. With this notion of object identity, each object is distinguishable and therefore the system can discriminate any two objects without depending on their states. Also, two or more fields can have the same object as their value.

Figure 1 shows a textual form (object expression) that represents a RectSelect object, which models a rectangle that has an associated cursor point. (This particular example has the cursor in the upper-right corner.) Structurally, this RectSelect object

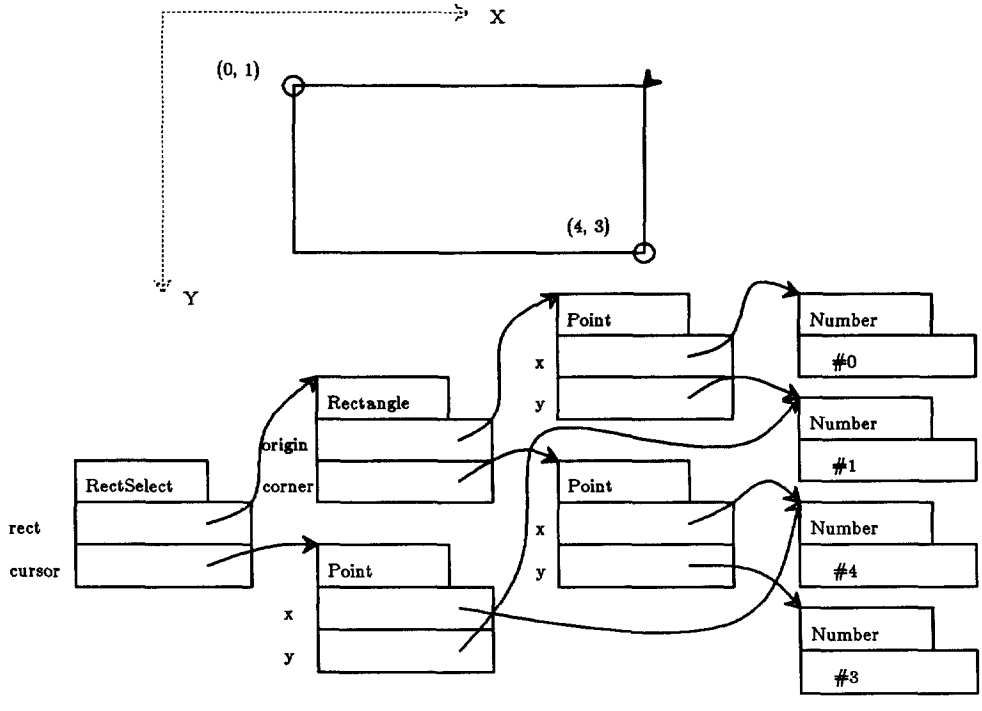

Fig.2 A RectSelect Object

relational database theory [BBDMR84, BanK85, BatK85, Ka83, MSOP86, RS85]. A data model of this kind is variously termed an *object-oriented data model* (OODM), a *non-first normal form data model* (NF^2DM), an *engineering data model* (EDM), or a *semantic data model* (SDM).

Maier et al. [Ma85, AEM86] proposed an object-oriented data model — the Tektronix Engineering Data Model (TEDM). Among other features, TEDM supports *object identities*, *complex objects*, a *type hierarchy* and a rule-like command language. We provide an overview of this data model shortly.

In this paper, we introduce *abstract objects* as an attempt at providing a unified framework for the database language and a means for making the query language entities persistent. (By query language, we really mean the complete database language.) Abstract objects are complex data structures with a declarative semantics, but which also support operational interpretations, much like literals in logic languages. Strategically, we adopt the idea from logic databases that an application database is some kind of closure of application data plus meaningful queries on that data. We go one step further, storing query entities as abstract objects, which have object identities and subcomponents. These abstract objects are very structural in nature, but they are quite rich in operational interpretations. As objects, they can be created, stored, manipulated and viewed just as normal data objects. As operators, they can be used as matchers and makers for other objects, specifying operations such as creating objects, retrieving objects and modifying subcomponents of existing objects.

We expect the next generation of database systems would be constructed based on a layered architecture — a data language processor sits on top of an intelligent storage server that has its own internal execution model. In that respect, what we propose here may be more of an intermediate construct that is suitable for communication between the storage server and the semantic processor. With this architectural consideration, the advantage of having abstract objects as proposed here is obvious. Using these abstract objects as building blocks, we can construct database program objects. Ultimately, the semantics of program objects should be describable as yet other objects, giving the ultimate in