A Graph-Oriented Object Model for Database End-User Interfaces

Marc Gyssens
Dept WNIF
Unıversıty of Lımburg
Unıversıtaıre Campus
B-3610 Dıepenbeek, Belgıum

Jan Paredaens
Dept of Math and Comp Scıence
Unıversıty of Antwerp
Unıversıteıtsplein 1
B-2610 Antwerpen, Belgıum

Dırk Van Gucht
Comp Scıence Department
Indıana Unıversıty
Lındley Hall 101
Bloomıngton, IN 47405

1 Introduction

The current database research trend ıs towards systems whıch can deal wıth advanced data applıcatıons that go beyond the standard "enterprıse" or "office" database applıcatıon Thıs trend ıs reflected ın the research on extensıon archıtectures [5,18,21] and object-orıented databases [2,3,12,21] Along wıth thıs trend, the need for better and easıer-to-use database end-user ınterfaces has been stressed [18,21] Therefore, we propose a graph-based data model, whıch shares many features wıth exıstıng data models, but whıch better facılıtates the rıgorous study of graphıcal database end-user ınterfaces

Graphs have been an ıntegral part of the database desıgn process ever sınce the ıntroductıon of semantıc data models [11,12] Theır usage ın data manıpulatıon languages, however, ıs far more sparse To deal wıth data manıpulatıon, typıcally, schemes ın semantıc data models are transformed ınto a conceptual data model such as the relatıonal model [19] The requıred database language features then become those of the conceptual model Object-orıented data models, on the other hand, often offer computatıonal complete, non-graphıcal data languages, usually ın the style of object-orıented programmıng languages such as Smalltalk Due to theır expressıveness, however, these language do not lend themselves easıly as hıgh-level data languages [2,21]

The first graphıcal database end-user ınterfaces were developed for the relatıonal model (for example Zloof's Query-By-Example (QBE) [22])

The earlıest graphıcal database end-user ınterfaces for semantıc models were assocıated wıth the Entıty-Relatıonshıp model [7,16,20] Subsequently, graphıcal ınterfaces were developed for more complex semantıc and object-orıented database models [4,8,13,14,15] These ınterfaces use *graphs* as theır central tool, but as far as data languages, they are usually lımıted ın expressıve power Graph-orıented end-user ınterfaces have also been developed for recursıve data objects and querıes [6,10,22]

In [9], we ıntroduced the *Graph-Orıented Object Database Model (GOOD)* Thıs model ıs buılt around a sıngle mathematıcal tool, namely graphs, to both model *and* manıpulate databases We belıeve that thıs ıs an ımportant step ın the dırectıon of rıgorously studyıng and developıng database end-user ınterfaces

In [9], we lımıted ourselves to descrıbıng a sımple yet powerful transformatıon language and dıscussıng ıts expressıveness In thıs paper, we further develop and ınvestıgate *GOOD* We show that ıt has many features generally present ın exıstıng semantıc, object-orıented and deductıve database models Specıfically, we demonstrate how the *GOOD* model ıs suıtable for *graphıcally* descrıbıng queryıng, browsıng, restructurıng and updatıng databases, and hence ıs ıdeally suıted for the study and development of graphıcally-orıented database end-user ınterfaces To demonstrate why *GOOD* ıs useful for advanced data applıcatıons, we descrıbe how ıt can be seen as an object-orıented data model

In Sectıon 2, we define the basıc *GOOD* model In Sectıon 3, we dıscuss queryıng, browsıng, restructurıng and updatıng, and show that they can all be expressed naturally ın a unıform, graphıcally-orıented and user-frıendly manner We also show how to use *GOOD* to manıpulate and query database schemes In Sectıon 4, we show how to adapt the *GOOD* model to ıncorporate the features of object-orıented database systems

2 Basic features of GOOD

A data model based on graphs offers an attractive possibility to graphically formulate database activities The *Graph-Oriented Object Model (GOOD)*, first introduced in [9], is such a model In this section, we present its basic features

2.1 Object base schemes and instances

We first turn to the representation of data [1] At the instance level, the data will be represented as a *directed labeled graph* The *nodes* of this graph represent the *objects* of the database We only distinguish between *non-printable nodes* (represented as squares) and *printable nodes* (represented as circles) As for the edges, we only distinguish between *functional edges* (shown as "$\rightarrow$") and *non-functional edges* (shown as "$\twoheadrightarrow$") A functional edge denotes a functional relationship between the objects connected by that edge, i e, for a given functional edge label and and a given (non-printable) node, there can be a most one edge leaving that node with the given functional edge label On the other hand, for a given non-functional edge label and a (non-printable) node there can be an unbounded (but finite) number of edges leaving that node all having the same given non-functional edge label

As an example, consider an object base representing vehicles together with their "physical" makeup Figure 1 shows a part of an (oversimplified) instance of such an object base, consisting of two similar cars owned by the same person

Of course, the graph representing a particular object base must obey certain structural constraints These are contained in the object base scheme which can be represented by a set of graph productions For the vehicle object base, which will be used as a running example throughout the paper, we assume the object base scheme of Figure 2 Observe that this scheme also allows for airplanes ("*A*") to be represented in the object base

More generally, we assume there are infinitely enumerable sets of *nodes*, *non-printable object labels*, *printable object labels*, *functional object labels* and *non-functional object labels* These four sets of labels are assumed to be pairwise disjoint We also assume there is a function π which associates to each printable object label a set of *constants* (e g , character, a string, a number, a boolean, , but also a drawing, a graph, a table, etc)

We then define an *object base scheme* as a five-tuple $\mathcal{S} = (NPOL, POL, FEL, NFEL, \mathcal{P})$ with *NPOL* a finite set of non-printable object labels, *POL* a finite set of printable object labels, *FEL* a finite set of functional edge labels, *NFEL* a finite set of non-functional edge labels, and $\mathcal{P}$ a set of *productions* (L, f) with $L \in NPOL$ and f $FEL \cup$

[1] It should be noted that the representation of data resembles that in the Functional Data Model [17]

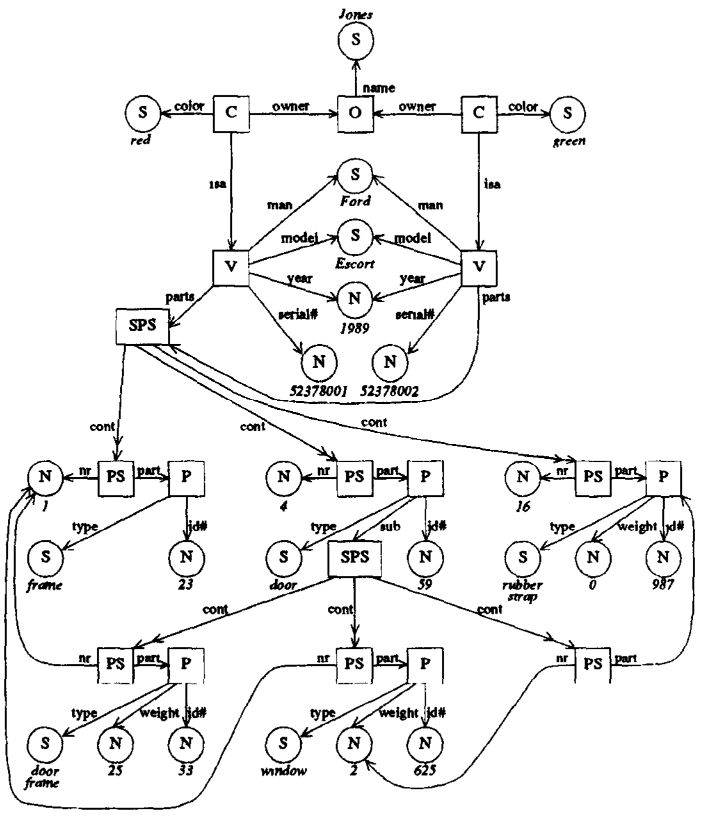

Figure 1 An example of an object base instance

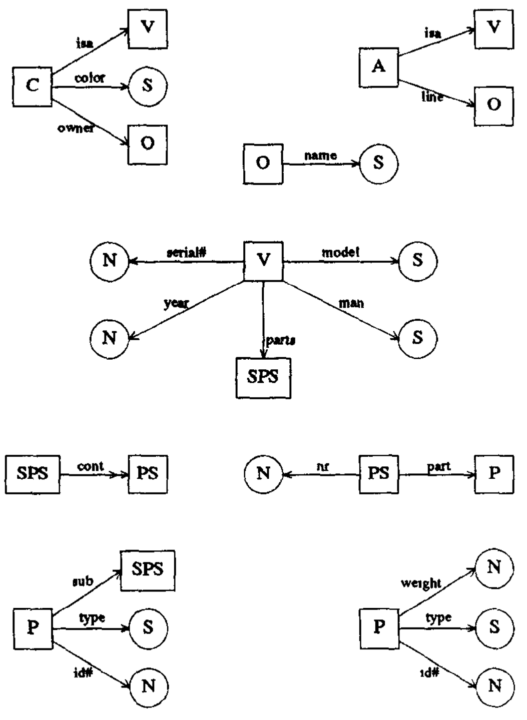

Figure 2 An example of an object base scheme

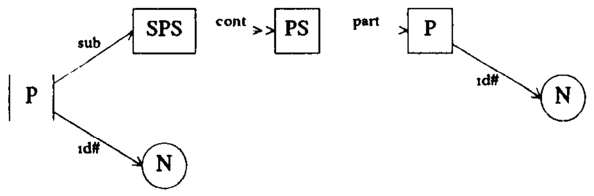

Figure 3 An example of a pattern

NFEL → *NPOL*∪*POL* a partial mapping from edge labels to object labels

We now turn to object base instances Thereto, let $\mathcal{S} = (NPOL, POL, FEL, NFEL, \mathcal{P})$ be an object base scheme Formally, an *object base instance* over $\mathcal{S}$ is a labeled graph $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ for which

- $\mathbf{N}$ is a set of labeled nodes, if $\mathbf{n}$ is a node in $\mathbf{N}$, then the label $\lambda(\mathbf{n})$ of $\mathbf{n}$ must be in $NPOL \cup POL$, if $\lambda(\mathbf{n})$ is in $NPOL$ (respectively in POL), then $\mathbf{n}$ is called a *non-printable node* (respectively a *printable node*),
- $\mathbf{E}$ is a set of labeled edges, if $\mathbf{e}$ is a labeled edge in $\mathbf{E}$, then $\mathbf{e} = (\mathbf{m}, \alpha, \mathbf{n})$ with $\mathbf{m}$ and $\mathbf{n}$ in $\mathbf{N}$, $\mathbf{m}$ a non-printable node, and the label $\alpha = \lambda(\mathbf{e})$ of $\mathbf{e}$ in $FEL \cup NFEL$, if $\lambda(\mathbf{e})$ is in FEL (respectively in $NFEL$), then $\mathbf{e}$ is called a *functional edge* (respectively a *non-functional edge*),
- each printable node $\mathbf{n}$ in $\mathbf{N}$ has an additional label print($\mathbf{n}$), called the *print label*, print($\mathbf{n}$) must be a constant in $\pi(\lambda(\mathbf{n}))$,
- each two functional edges $(\mathbf{m}, \alpha, \mathbf{n}_1)$ and $(\mathbf{m}, \alpha, \mathbf{n}_2)$ in $\mathbf{E}$ (leaving the same node and having the same label) are equal (i e $\mathbf{n}_1 = \mathbf{n}_2$), and
- for each non-printable node $\mathbf{m}$ in $\mathbf{N}$ there exists a production $(\lambda(\mathbf{m}), f)$ in $\mathcal{P}$ such that for each edge $(\mathbf{m}, \alpha, \mathbf{n})$ in $\mathbf{E}$, $f(\alpha) = \lambda(\mathbf{n})$

2.2 The transformation language

The *GOOD* data transformation language consists of five operators, four of which correspond to elementary manipulations of graphs addition of nodes, addition of edges, deletion of nodes and deletion of edges The fifth operator, called abstraction, is used to group nodes on the basis of common functional or non-functional properties

The specification of all five operators relies on the notion of *pattern* A pattern is a graph used to describe subgraphs in an object base instance

As an example, consider the graph in Figure 3 This graph represents a pattern over the vehicle object base scheme Intuitively, it describes an occurrence of a part, together with its identification number and one of its (immediate) subparts with its identification number In order to actually obtain such occurrences in a particular instance, we have to "match" the pattern with subgraphs of the instance under consideration E g the pattern in Figure 3 can be matched with a subgraph of the instance in Figure 1 in three different ways, corresponding with the part-subpart id#-pairs (59, 33), (59, 625) and (59, 987) respectively

In order to formalize this notion of "matching", we introduce so-called *embeddings* Let $\mathcal{S}$ be an object base scheme, let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance over $\mathcal{S}$ and let $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ be a pattern over $\mathcal{S}$ An *embedding* of $\mathcal{J}$ in $\mathcal{I}$ is a total mapping $\imath$ $\mathbf{M} \to \mathbf{N}$ preserving all labels, i e node labels, edge labels as well as print labels (where defined) For the pattern in Figure 3, there are three embeddings into the object base instance in Figure 1, each of which corresponds with one of the "matchings" described above

We are now ready to define the five operations of the *GOOD* transformation language

2.2.1 Node addition

Let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance and $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ a pattern over object base scheme $\mathcal{S}$ Let $\mathbf{m}_1, \ldots, \mathbf{m}_n$ be nodes in $\mathbf{M}$ Let K be a non-printable object label that is *not* the label of a node in $\mathbf{M}$ and let $\alpha_1, \ldots, \alpha_n$ be functional edge labels The *node addition* $\mathrm{NA}[\mathcal{J}, \mathcal{S}, \mathcal{I}, K, \{(\alpha_1, \mathbf{m}_1), \ldots, (\alpha_n, \mathbf{m}_n)\}] = (\mathcal{J}', \mathcal{S}', \mathcal{I}')$ where (1) $\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ where $\mathbf{M}'$ is obtained by adding to $\mathbf{M}$ a new node $\mathbf{m}$ with label K, $\mathbf{F}'$ is then obtained by adding to $\mathbf{F}$ the labeled functional edges $(\mathbf{m}, \alpha_1, \mathbf{m}_1), \ldots, (\mathbf{m}, \alpha_n, \mathbf{m}_n)$, (2) $\mathcal{S}'$ is the minimal scheme of which $\mathcal{S}$ is a subscheme [2] and over which $\mathcal{J}'$ is a pattern, and (3) $\mathcal{I}'$ is the minimal instance over $\mathcal{S}'$ for which $\mathcal{I}$ is a subinstance of $\mathcal{I}'$, for each embedding $\imath$ of $\mathcal{J}$ in $\mathcal{I}$, there exists a K-labeled node $\mathbf{n}$ in $\mathcal{I}'$ such that $(\mathbf{n}, \alpha_1, \imath(\mathbf{m}_1)), \ldots, (\mathbf{n}, \alpha_n, \imath(\mathbf{m}_n))$ are functional edges in $\mathcal{I}'$, and each edge in $\mathcal{I}'$ leaving a node of $\mathcal{I}$ is also an edge of $\mathcal{I}$

A node addition will be represented by drawing the pattern P' and mark in bold the node and edges not in $\mathcal{J}$ Suppose for example we want to effectively create nodes representing the pairs of identification numbers of parts and their (immediate) subparts, occurring in the object base instance of Figure 1 Using the pattern in Figure 3, this operation can be accomplished by performing the node-addition represented in Figure 4 The resulting object base instance is obtained by adding to the instance in Figure 1 three *PN*-labeled nodes, one for each of the part-subpart id#-pairs (59, 33), (59, 625) and (59, 987), together with the associated edges

2.2.2 Edge addition

Let $\mathcal{I} = (\mathbf{N}, \mathbf{E})$ be an object base instance and $\mathcal{J} = (\mathbf{M}, \mathbf{F})$ a pattern over object base scheme $\mathcal{S}$ Let

[2] Subscheme and subinstance are defined with respect to set inclusion

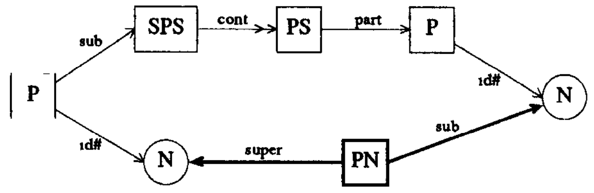

Figure 4 Example of a node addition

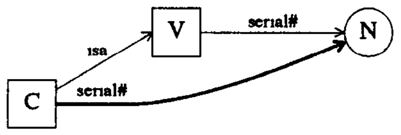

Figure 5 An example of an edge addition

$\mathbf{m}_1, \ldots, \mathbf{m}_n, \mathbf{m}'_1, \ldots, \mathbf{m}'_n$ be nodes in M and let $\alpha_1, \ldots, \alpha_n$ be arbitrary edge labels

The *edge addition*

$EA[\mathcal{J}, \mathcal{S}, \mathcal{I}, \{(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_n, \alpha_n, \mathbf{m}'_n)\}]$ $= (\mathcal{J}', \mathcal{S}', \mathcal{I}')$ where (1) $\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ where $\mathbf{M}'$ equals M and $\mathbf{F}'$ is obtained by adding to F the labeled edges $(\mathbf{m}_1, \alpha_1, \mathbf{m}'_1), \ldots, (\mathbf{m}_n, \alpha_n, \mathbf{m}'_n)$, (2) $\mathcal{S}'$ is the minimal scheme of which $\mathcal{S}$ is a subscheme and over which $\mathcal{J}'$ is a pattern, and (3) $\mathcal{I}'$ is the minimal instance over $\mathcal{S}'$ for which $\mathcal{I}$ is a subinstance of $\mathcal{I}'$, and such that for each embedding $\imath$ of $\mathcal{J}$ in $\mathcal{I}'$, $(\imath(\mathbf{m}_1), \alpha_1, \imath(\mathbf{m}'_1)), \ldots, (\imath(\mathbf{m}_n), \alpha_n, \imath(\mathbf{m}'_n))$ are labeled edges in $\mathcal{I}'$

As for node addition, we will denote an edge addition by drawing the graph $\mathcal{J}'$ and marking in bold the edges not in $\mathcal{J}$ As an example, reconsider the vehicle object base Suppose we want to see the serial number as a property of cars too, rather than of vehicles alone This transformation can be accomplished by the edge addition shown in Figure 5

Note that the edge addition can have a *recursive* effect (it is the only such operator), since we allow some of the labels $\alpha_1, \ldots, \alpha_n$ to occur in the pattern $\mathcal{J}$ (for examples, see [9])

2.2.3 Node deletion

To the node addition corresponds a *node deletion*, which in the object base instance removes nodes in all subgraphs described by a pattern The node to be deleted is marked in outline

As an example, consider the very simple node deletion of Figure 6 It removes the information about owners from the vehicle object base

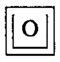

Figure 6 An example of a node deletion

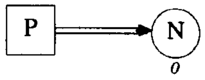

Figure 7 An example of an edge deletion

2.2.4 Edge deletion

To the edge addition corresponds the *edge deletion* As for node deletion, the edges to be removed are marked in outline

Reconsider the vehicle object base Suppose we want to remove the weight from all elementary parts for which the weight is negligible (i e , equal to zero in Figure 1) This removal can be accomplished with the edge deletion in Figure 7

2 2 5 Abstraction

In the *GOOD* model, different nodes represent different objects, even if they cannot be distinguished on the basis of their properties Therefore, we have introduced an *abstraction* operator that allows to define new nodes *in terms of functional or non-functional properties represented in the object base*

Let $\mathcal{S}$ be an object base scheme Let $\mathcal{I} = (\mathrm{N}, \mathrm{E})$ be an object base instance and $\mathcal{J} = (\mathrm{M}, \mathrm{F})$ a pattern over $\mathcal{S}$ Let $\mathbf{n}$ be a non-printable node in M Let K be a non-printable object label that is *not* the label of a node in M, let $\alpha_1, \ldots, \alpha_n$ be edge labels and let β be a non-functional edge label not occurring in $\mathcal{S}$ Intuitively, the abstraction creates sets (labeled K) Each set contains all the objects $\mathbf{n}$ that match the pattern $\mathcal{J}$ and that have the same $\alpha_1, \ldots, \alpha_n$-properties

More formally, the *abstraction*

$AB[\mathcal{J}, \mathcal{S}, \mathcal{I}, \mathbf{n}, K, \{\alpha_1, \ldots, \alpha_n\}, \beta] = (\mathcal{J}', \mathcal{S}', \mathcal{I}')$ where (1) $\mathcal{J}' = (\mathbf{M}', \mathbf{F}')$ where $\mathbf{M}'$ is obtained by adding to M a new node $\mathbf{m}$ with label K, $\mathbf{F}'$ is then obtained by adding to F the labeled non-functional edge $(\mathbf{m}, \beta, \mathbf{n})$, (2) $\mathcal{S}'$ is the minimal scheme of which $\mathcal{S}$ is a subscheme and over which $\mathcal{J}'$ is a pattern, (3) $\mathcal{I}'$ is the minimal instance over $\mathcal{S}'$ for which $\mathcal{I}$ is a subinstance of $\mathcal{I}'$, and (4) for each embedding $\imath$ of $\mathcal{J}$ in $\mathcal{I}$, there exists a K-labeled node $\mathbf{p}$ in $\mathcal{I}'$ such that $(\mathbf{p}, \beta, \imath(\mathbf{n}))$ is a non-functional edge of $\mathcal{I}'$, if $(\mathbf{p}, \beta, \mathbf{q}_1)$ and $(\mathbf{p}, \beta, \mathbf{q}_2)$ are both in $\mathcal{I}'$ then for each $\imath = 1, \ldots, n$ and for each node $\mathbf{r}$ in $\mathcal{I}'$, $(\mathbf{q}_1, \alpha_\imath, \mathbf{r}) \in \mathrm{E}' \Leftrightarrow (\mathbf{q}_2, \alpha_\imath, \mathbf{r}) \in \mathrm{E}'$, and each edge in $\mathcal{I}'$ leaving a node of $\mathcal{I}$ is also an edge of $\mathcal{I}$

As for node addition, we will denote an abstraction by drawing the graph $\mathcal{J}'$ and marking in bold the node and edges not in $\mathcal{J}$ The edges $\alpha_1, \ldots, \alpha_n$ will be marked dot-dashed, if these edges do not occur in $\mathcal{J}$, they will be added without drawing the nodes in which they arrive

To conclude this section, we present an example of an abstraction Suppose that in the vehicle object base we want to "abstract" over vehicles, independent of their se-

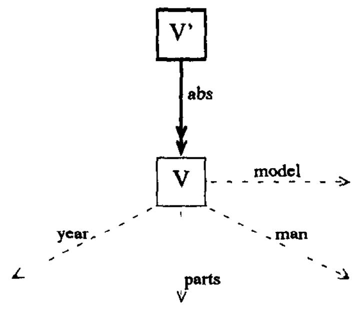

Figure 8 An example of an abstraction

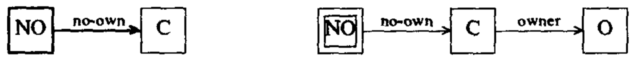

Figure 9 An example of a query involving node deletion

rial numbers This can be done by performing the abstraction of Figure 8 At the instance level, the abstraction in Figure 8 results in the creation of one new non-printable node labeled V' from which two non-functional edges labeled "*abs*" leave, pointing to both vehicle nodes, respectively

3 GOOD as a database interface

The *GOOD* model is equipped with the necessary tools to handle typical database operations To validate this, we will give examples of how to browse, query, restructure, and update *GOOD* object bases The graph-orientation of the *GOOD* model will then imply that such database operations can be formulated graphically

3.1 Querying

Querying is retrieving information from a database without affecting its information contents We have already, on several occasions, discussed the *GOOD* data transformation language as a tool to query databases (see Section 2 2 1 and Section 2 2 2) In this section, we will consider some additional queries about the vehicle object base We will begin with a query using the node deletion operator Consider the query "*Find all the cars which do not have an owner* " We start with marking all the cars, using the node addition in Figure 9 Using node deletion, we then remove the *NO*-nodes associated with owned cars The remaining *NO*-nodes now point to cars without owners

Now, consider the range query "*Find all elementary parts (parts without subparts) with weight strictly between 10 and 100* " As such, we can not solve this query, because it involves the *larger than* relation over the natural numbers However, since considering *larger than*, within the context of *GOOD*, yields some interesting insights, we will make a small digression at this point

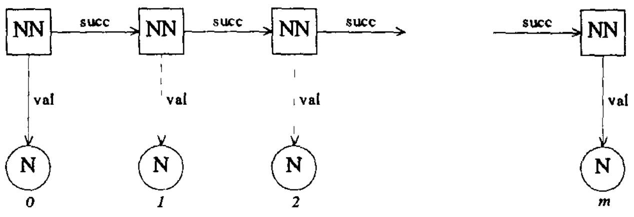

Figure 10 A representation of natural numbers in *GOOD*

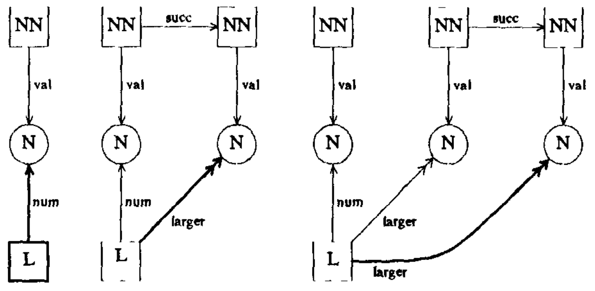

Figure 11 Computing the *larger than* relation

Assume that we are given an arbitrary but finite sequence of successive natural numbers starting at zero Since it is the most characteristic aspect of the natural numbers, we assume the successor function to be known Figure 10 illustrates how to represent such a sequence in *GOOD* at the instance level, the corresponding scheme is obvious

It is now a straightforward exercise to construct the *larger than* relation over the numbers in this sequence This can be done by a node addition, followed by two edge additions, as shown in Figure 11 The first edge addition indicates that the direct successor of a number is larger than that number The second edge addition then basically computes the transitive closure of that relation

Let us now return to the above range query To denote its result, we first introduce a new node, R, via a simple node addition over an empty pattern The edge addition in Figure 12 specifies the solution to the query

3.2 Browsing

Browsing is traversing a database according to its underlying scheme Obviously, browsing does not alter the information contents of the database Browsing can be accomplished as a succession of node additions wherein the newly created nodes serve as markers for the object(s) of interest

3.3 Restructuring

Restructuring a database is transforming its structure (scheme) without altering its information contents This can be necessary to accommodate a different view of the

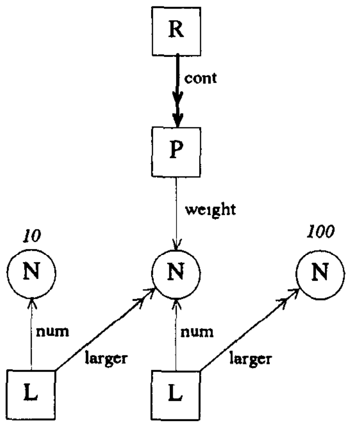

Figure 12 An example of a query involving the *larger than* relation

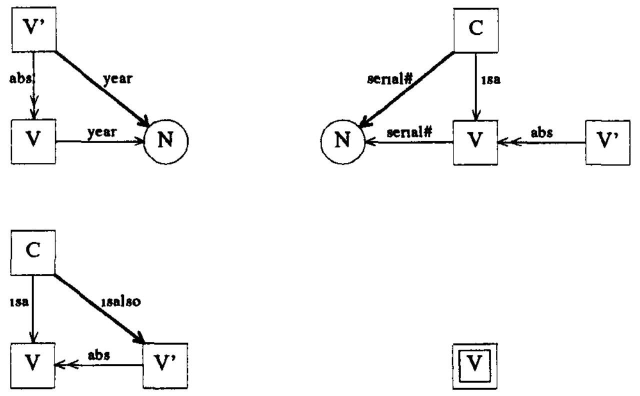

Figure 13 Restructuring the vehicle object base

database, to remove redundancy, or to allow for more efficient access to the data

Suppose we want to restructure the vehicle object base so that serial numbers are no longer associated with vehicle objects but rather are directly associated with the appropriate car and plane objects Reconsider Figure 8, representing an abstraction operation Each added V'-object is associated with all the vehicles (V-objects) sharing the same year, parts, model, and manufacturer Using four edge additions, we can attach the year, parts, model, and manufacturer information to the V'-objects Figure 13 (upper left) shows the edge addition involving the year information, similar edge additions can be done for parts, model, and manufacturer information Next, we can associate the serial numbers with the car and plane objects (see Figure 13, upper right drawing, for the car objects) An additional edge addition is needed to associate the appropriate V'-objects with the car (plane) objects (see Figure 13, lower left drawing, for the car objects) Finally, the node deletion shown in the lower right corner of Figure 13 removes the redundant V-objects and completes the restructuring

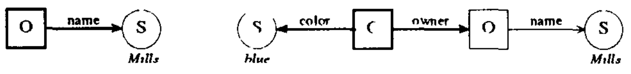

Figure 14 An example of an insertion

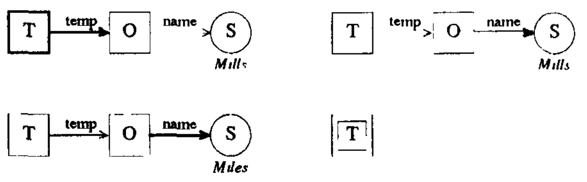

Figure 15 An example of a modification

3.4 Updating

Updating a database involves changing its information contents, without affecting its scheme Updates are typically the result of insertions, deletions, and modifications of data

As an example, suppose, we want to insert a (new) *blue* car, with owner *Mills*, into the vehicle object base This insertion can be expressed by two subsequent node additions, shown in Figure 14 First, we add an owner with name *Mills* Then, we insert a *blue* car with owner *Mills*

Assuming that the owner's name of the just inserted car is really *Miles*, we can do the following *GOOD* operations to reflect this modification, shown in Figure 15 First, we use a node addition to mark the owner with name *Mills* (If the "*name*"-edge uniquely identifies an owner, we will have marked one node) Using an edge deletion, we disassociate the printable node with print label *Mills* from the marked owner object An additional edge addition associates the marked owner with the name *Miles*, and finally use a node deletion to remove the marking object

3.5 Meta modeling

Many data models have the ability to specify a scheme which has as instances the valid database schemes of that model This technique is commonly referred to as *meta modeling* Meta modeling allows for the application of the data model operations to database schemes

In Figure 16 we show a meta scheme for valid *GOOD* object base schemes The node labels "N", "E", "NT", "ET", and "S" stand for node, edge, type of node (nonprintable or printable), type of edge (functional or not

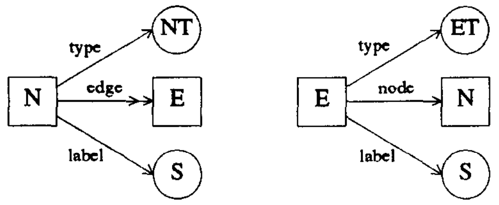

Figure 16 The meta scheme of the *GOOD* model

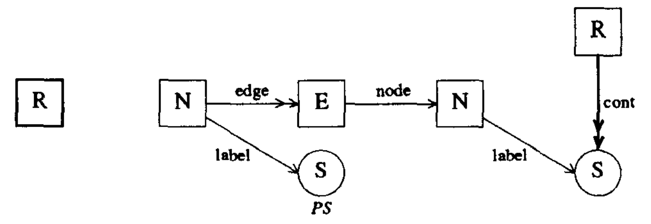

Figure 17 Browsing through the scheme using the meta object base

functional), and string, respectively The edge labels "*edge*", "*type*" and "*label*" are self explanatory The edge label "*node*" indicates the node in which the edge arrives

If we are now interested in the node labels pointed at by edge labels emanating from the "*PS*"-nodes in the object base scheme of Figure 2, we can obtain these labels by applying the *GOOD* transformation language to the meta object base This *GOOD* transformation, consisting of a node addition and an edge addition, is shown in Figure 17

4 Object-orientedness of GOOD

In Section 2 and Section 3 we discussed *GOOD* as a simple model to rigorously study graphical database end-user interfaces Here, we will concentrate on the *GOOD* model as a data model More specifically, we will consider the *GOOD* model in the context of object-oriented database models [2,3,12,21] This analysis will yield two insights first, it will illustrate the modeling power of the *GOOD* model, and second, it will propose an explicit tool to study graphical end-user interfaces for object-oriented database systems Our analysis was guided by the *Object-Oriented Database System Manifesto* [2]

4.1 Modeling features

4.1.1 Complex objects and object-identity

Complex objects are typically built from primitive objects (natural numbers, booleans, strings etc) [1,2] according to certain object constructors, such as tuples, sets, and lists Clearly, the *GOOD* model supports such complex objects

The notion of *object-identity* refers to the existence of objects in the database independent of their associated properties As stressed from the outset, object-identity is a basic feature of the *GOOD* model

4.1.2 Classes, hierarchies and inheritance

All object-oriented databases support some form of *inheritance*, i e , it is customary to define new classes as subclasses of existing ones (e g [12,21]), therefore organizing the classes in a *class hierarchy*

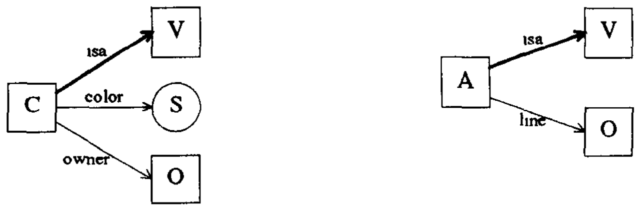

Figure 18 An object base scheme with subclasses

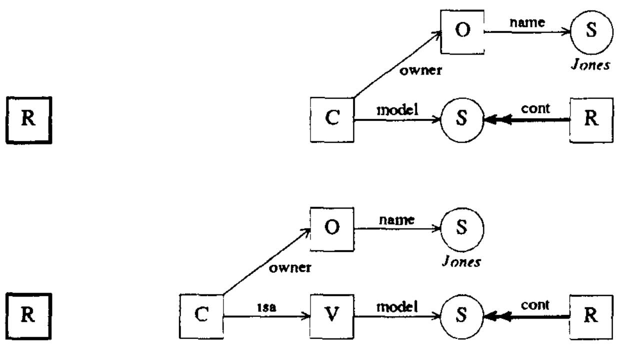

Figure 19 Specifying a query using inheritance

In the *GOOD* model, classes can be associated with node labels in schemes Functional edge labels can then support the notion of subclass However, it is clear that not *all* functional edge labels in an object base scheme can be interpreted as a subclass-relationship Therefore, we will mark in bold the functional edges in the scheme graph we wish to interpret as subclass edges (we will implicitly assume that the subclass edges do not form a "cycle" in the object base scheme)

For example, we can consider the "*isa*"-edges in the vehicle object base scheme shown in Figure 2 as subclass-edges The effect to the user is the same as if all properties of vehicle objects were attached to the corresponding car and plane objects (Clearly, this transformation can be simulated by a number of edge additions) The user can now apply the vehicle operations directly to cars E g , suppose we want to know the models of cars owned by *Jones* This query can now be specified as in Figure 19 (top)

4.2 Language features

4.2.1 Methods, encapsulation and ad-hoc query language

In this section, we define the concept of *method* in the *GOOD* model and discuss encapsulation As for the ad-hoc query language, we refer to Section 3 1

A *GOOD* method is a *named* procedure associated with a labeled node [3] It has *parameters*, a *method specification*, a *method body* and a *method interface* Throughout this section, let S be an object base scheme

[3]This node corresponds to a class, as described in Section 4 1 2

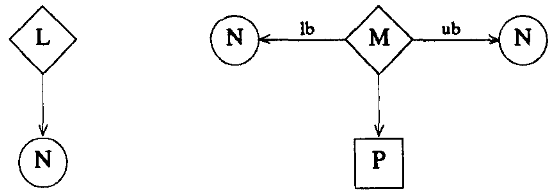

Figure 20 Examples of method specifications

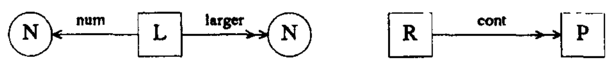

Figure 21 The interfaces of methods L and M

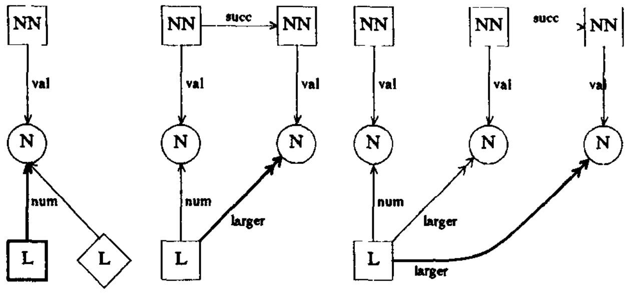

Figure 22 The body of method L

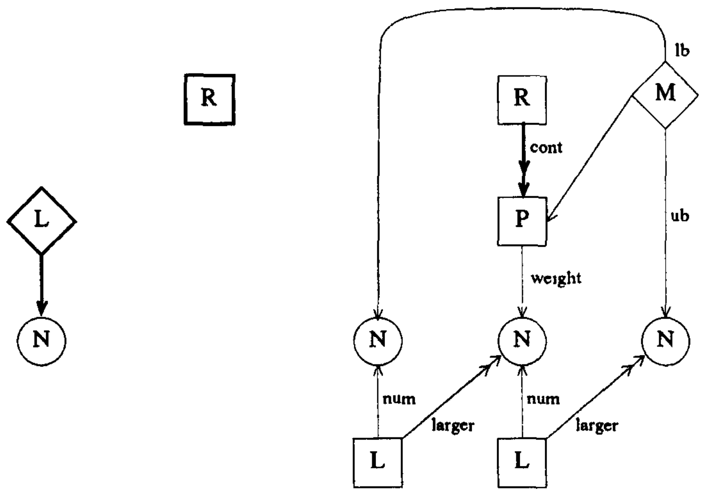

Figure 23 The body of method M

The method specification contains the method's name, its associated node label (class) and its parameters Formally, the *method specification* $S_{\mathcal{M}}$ of a method $\mathcal{M}$ is a couple $(O_{\mathcal{M}}, f_{\mathcal{M}})$, where $O_{\mathcal{M}} \in NPOL \cup POL$ is called the *owner* and $f_{\mathcal{M}}$ is a total function, $f_{\mathcal{M}}$ $L_{\mathcal{M}} \rightarrow NPOL \cup POL$, with $L_{\mathcal{M}}$ a finite (possibly empty) set of labels $f_{\mathcal{M}}$ associates with each of its labels a parameter Graphically $(O_{\mathcal{M}}, F_{\mathcal{M}})$ is represented by a diamond-shaped node that is labeled by $\mathcal{M}$, with one unlabeled outgoing edge to a node labeled by $O_{\mathcal{M}}$ and a labeled outgoing edge for each label $\lambda \in L_{\mathcal{M}}$ to a node labeled by $f_{\mathcal{M}}(\lambda)$ No two edges point to a same node

As an example, consider Figure 20, in which two methods are specified The first method is $L = (N, \emptyset)$ L has as owner the non-printable node label N and has no parameters The second method is $M = (P, f_M)$ M has as owner the non-printable node label P and has two parameters, i e , f_M associates with the label *lb* the printable node label N and f_M associates with the label *ub* the printable node label N

In general, it may be anticipated that a method call has "side effects" in the form of objects and edges which are introduced to perform intermediate computations Some of these side effects will be desirable, some other unwanted Since a user needs to be protected from unwanted side effects, our methods have an associated interface which specifies only the desired side effects Formally, the *interface* $I_{\mathcal{M}}$ of a method $\mathcal{M}$ is an object base scheme (We will also require that if (L, f) is a production in the object base scheme S and (L, g) is a production $I_{\mathcal{M}}$ then $dom(f) \cap dom(g)$ must be empty)

The method interface for method L is shown in Figure 21 (left) and that for method M is shown in Figure 21 (right)

The *method body* specifies the implementation of the method Formally, the *method body* $B_{\mathcal{M}}$ of a method $\mathcal{M}$ is a sequence of parameterized operations *Parameterized operations* are normal operations (i e , NA, ND, EA, ED, AB or MC (method call, see further)) or normal operations where the source pattern $\mathcal{J}$ is augmented with one diamond-shaped node labeled by $\mathcal{M}$, called the $\mathcal{M}$-head-node, and with edges leaving that node At most one unlabeled edge can leave the $\mathcal{M}$-head-node It has to point to a node m_O labeled by $O_{\mathcal{M}}$ At most one edge for each label λ of $L_{\mathcal{M}}$ can leave the $\mathcal{M}$-head-node It has to point to a node m_λ labeled by $f_{\mathcal{M}}(\lambda)$ No other edges can leave the $\mathcal{M}$-head-node

As an example, suppose L is the method computing the *larger than* relation over the natural numbers as represented in Figure 10 In Section 3 1, we computed this relation as a query using three primitive $GOOD$ operations (see Figure 11) We can represent these three operations as consecutive steps in the body of the method L, as shown in Figure 22

As another example, suppose M is the method returning all parts in the vehicle object base having a weight strictly included between a given lower and upper bound (cfr. 12) Using the method L, the body of M can be drawn as in Figure 23 It consists of three steps The second and third are primitive $GOOD$ operations (Note the parameter binding in the third step) The first step is a *method call* of the method L, it is used to compute the *larger than* relation

The *method call* is the operation that invokes the execution of the method body in a context specified by a pattern and actual parameters Formally, let $\mathcal{I} = (N, E)$

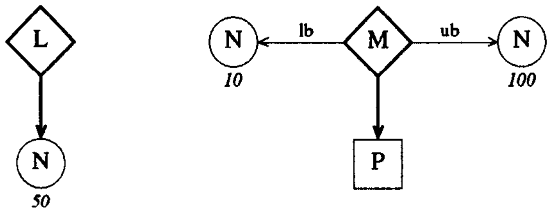

Figure 24 Examples of method calls

be an object base instance and $\mathcal{J} = (N', F)$ a pattern $\mathcal{S}$ Let M be a method of $\mathcal{S}$, m be a node of N' that is labeled by $O_{\mathcal{M}}$ and g be a total function, g $L_{\mathcal{M}} \rightarrow N'$ where $g(\lambda)$ must have the label $f_{\mathcal{M}}(\lambda)$ The *method call* is specified by $\mathrm{MC}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \mathcal{M}, m, g]$ and is represented by the pattern $\mathcal{J}$ augmented with a bold diamond shaped node, labeled $\mathcal{M}$, and from this node an unlabeled bold edge to m and a bold edge for each $\lambda \in L_{\mathcal{M}}$ to the node $g(\lambda)$

The semantics of the method call is then that the steps in the body of the method are executed consecutively, *but* only for these nodes in the instance under consideration that match the node in the pattern to which the method points, and only with the actual values of the parameters Formally, the method call $\mathrm{MC}[\mathcal{J}, \mathcal{S}, \mathcal{I}, \mathcal{M}, m, g] = (\mathcal{S}', \mathcal{I}')$ results in a new scheme $\mathcal{S}'$ and a new instance $\mathcal{I}'$ over $\mathcal{S}'$ defined as follows if there is no embedding of $\mathcal{J}$ in $\mathcal{I}$ then $(\mathcal{S}', \mathcal{I}') = (\mathcal{S}, \mathcal{I})$, otherwise, there are embeddings of $\mathcal{J}$ in $\mathcal{I}$, called *bindings embeddings* Let $(\mathcal{S}_1, \mathcal{I}_1)$ be the result of the execution of the parameterized operations in the body of method $\mathcal{M}$ The execution of a parametrized operation is equal to the execution of the associated normal operation, for each embedding e of this parameterized operation in $\mathcal{I}$ for which there is a binding embedding b with $e(m_O) = b(m)$ and $e(m_\lambda) = b(g(\lambda))$ [4] The new object scheme $\mathcal{S}'$ is obtained by augmenting the labels in $\mathcal{S}$ with the labels in $I_{\mathcal{M}}$ and replacing the set P of productions in $\mathcal{S}$ with the set of productions $\mathcal{P}' = \{(L, f)$ with (L, f) in P if there is no (L, f') in $I_{\mathcal{M}}$ (the interface of $\mathcal{M}$), or there is a (L, f_1) in $\mathcal{P}$ and (L, f_2) in $I_{\mathcal{M}}$ with $f = f_1 \cup f_2\}$ Finally, the new instance $\mathcal{I}'$ is defined as the *maximal* subinstance of $\mathcal{I}_1$, the scheme of which is $\mathcal{S}'$

For example, the method call of Figure 24, a method call of L *only*, computes the numbers larger than 50, subsequently the method M searches for *all* parts having weights strictly included between 10 and 100

Hence the result of the method call of M in Figure 24 is an object base over the scheme consisting of the productions in Figures 2 and 21, the object base is obtained by adding one R-labeled node to the object base in Figure 1 Two non-functional edges with label "*cont*" leave this R node, indicating the parts with weights between 10 and 100 Note that the L-nodes created to compute the *larger than* relation are *not* a part of the resulting instance

[4] m_O and m_λ were defined in the method specification of method $\mathcal{M}$

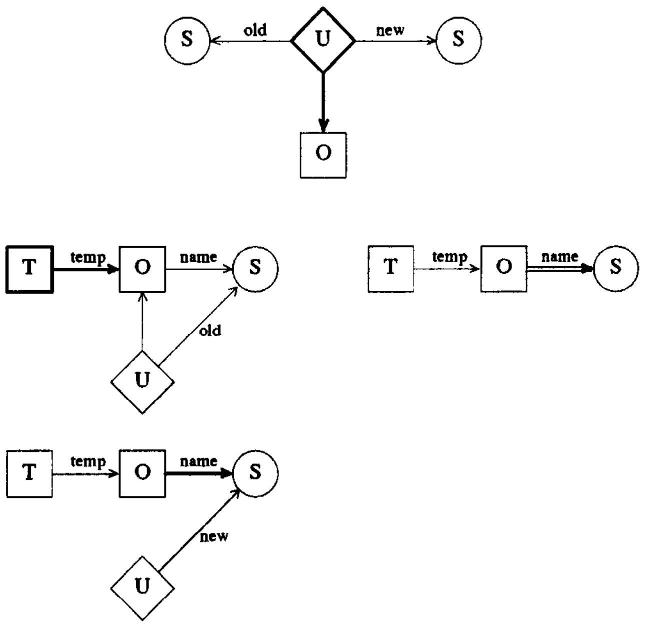

Figure 25 An example of method used for updating

An important property of object-oriented methods is that it provides *encapsulation* the result of a method should not depend on the actual implementation of that method, i e , methods should not have side effects Clearly, *GOOD* methods provide encapsulation, in the sense that the scheme of the result only depends on the interface of the method, i e , the user does not have to know the body of the method If the user knows the method specification and its interface, he can apply the method and know the structure of the result, no unwanted side effects will occur

As a final example of methods, we consider a method U for updating owner names in the vehicle object base (cfr Figure 15) The specification of the method U is shown in Figure 25, top part, the method body is shown in the bottom part of Figure 25, and the method interface is empty

4.2.2 Other Language Features

Most often, *computational completeness* in a database system is achieved by embedding the data manipulation language into a complete programming language such as Pascal or C The awkwardness of this process is commonly referred to as the *impedance match* problem Research on object-oriented database systems have therefore advocated to either support a computationally complete database language, or to design the data language so that it can be easily integrated into a complete language [2,21] We view the *GOOD* data transformation language augmented with the notion of methods as such a data language Indeed, throughout the paper, we argued that the *GOOD* data transformation language is suitable as a data manipulation language Furthermore, in [9], we have shown that the *GOOD* data transformation language can express the recursive functions, thus establishing its expressiveness Finally, although not shown in this paper, the notion of method and encapsulation should facilitate the integration of the *GOOD* data transformation language into a complete (preferably object-oriented) programming language

The notion of *extensibility* refers to the facility of freely adding new data types, so that these types have the same status as system defined types We believe that our treatment of encapsulation and methods is broad enough to support the extensibility feature of object-oriented data models

The concepts of *overriding*, *overloading* , and *late binding* refers to the usage of the same name for different operations and the consequent run time binding We do not see any inherent difficulty to support these concepts in an implementation of the *GOOD* model

Acknowledgments

The first author wishes to acknowledge the Belgian Fund of Scientific Research, which enabled him to visit Indiana University, where part of this research was performed The third author wishes to thank Latha S Colby and Ed Robertson for their insightful comments

References

[1] S Abiteboul and Beeri C On the power of languages for the manipulation of complex objects Technical report, INRIA, 1988

[2] M Atkinson, F Bancilhon, D DeWitt, K Dittrich, D Maier, and S Zdonik The object-oriented database system manifesto To be published

[3] F Bancilhon Object-oriented database systems In *Proc 7th ACM SIGACT-SIGMOD-SIGART Symp on Princ Database Systems, Austin*, pages 152–162, 1987

[4] D Bryce and R Hull SNAP A graphics-based schema manager In *Proc of the Int'l Conf on Data Engineering*, pages 151–164, 1986

[5] M J Carey, (Ed) Special issue on extensible database systems *Database Eng* , June 1987

[6] K F Cruz, A O Mendelzon, and P T Wood A graphical query language supporting recursion In *Proc SIGMOD Annual Conf , San Francisco*, pages 323–330, 1987

[7] D Fogg Living in a database In *Proceedings of ACM-SIGMOD 1983 International Conference on Management of Data, Boston*, pages 100–106, 1984

[8] K J Goldman, S A Goldman, P C Kanellakis, and S B Zdonik ISIS Interface for a semantic information system In *Proc of SIGMOD Conf , Austin*, pages 328–342, 1985

[9] M Gyssens, J Paredaens, and D Van Gucht A graph-oriented object database model To appear in *Proc 9th ACM SIGACT-SIGMOD-SIGART Symp on Princ of Database Systems*, 1990

[10] S Heiler and A Rosenthal G-WHIZ, a visual interface for the functional model with recursion In *Proc 11th Int'l Conference on VLDB, Stockholm*, pages 209–218, 1985

[11] R Hull and R King Semantic database modeling Survey, applications, and research issues *ACM Computing Surveys*, 19(3) 201–260, 1987

[12] W Kim and F H Lochovsky, editors *Object-Oriented Concepts, Databases, and Applications* ACM Press (Frontier Series), 1989

[13] R King Sembase A semantic DBMS In *Proc of the First Intl Workshop on Expert Database Systems*, pages 151–171, 1984

[14] R King and S Melville The semantics-knowledgeable interface In *Proc 11th Int'l Conference on VLDB, Singapore*, pages 30–37, 1984

[15] A Motro, A D'Atri, and L Tarentino The design of KIVIEW an object-oriented browser In *Proc 2nd Int'l Conf on Expert DB Systems*, pages 73–106, 1988

[16] D Reiner, M Brodie, G Brown, M Chilenskas, D Kramlich, J Lehman, and A Rosenthal The database design and evaluation workbench (DDEW) *IEEE Data Eng* , 7(4), 1984

[17] D Shipman The functional data model and the data language DAPLEX *ACM Transactions on Database Systems*, 6(1) 140–173, 1981

[18] M Stonebraker, editor *Readings in Database Systems* Morgan Kaufmann Publisher, Inc , 1988

[19] J D Ullman *Principles of Database and Knowledge-Base Systems, Vol 1,* Comp Science Press, 1989

[20] H K Wong and I Kuo GUIDE A graphical user interface for database exploration In *Proc 11th Int'l Conference on VLDB, Mexico City*, pages 22–32, 1982

[21] S B Zdonik and D Maier, editors *Readings in Object-Oriented Database Systems* Morgan Kaufmann, 1989

[22] M Zloof Query-By-Example *IBM Syst Journal*, 16 324–343, 1983