ViewFinder: An Object Browser*

Alessandro D'Atri
Dipartimento di Ingegneria Elettrica
Università degli Studi di L'Aquila
I-67040 L'Aquila, Italy

Amihai Motro
Department of Information and Software Systems Engineering
George Mason University
Fairfax, VA 22030-4444

Laura Tarantino
Dipartimento di Ingegneria Elettrica
Università degli Studi di L'Aquila
I-67040 L'Aquila, Italy

Technical Report ISSE-TR-95-115
February 1995

Abstract

ViewFinder is a graphical tool for browsing in databases that provides a flexible, yet intuitive environment for exploratory searches. The design approach has been to provide maximum *functionality* and *generality* without sacrificing *simplicity*. The constructs of ViewFinder's external model are essentially object-oriented: class and token objects, membership relationships between tokens and classes, generalization relationships between classes, inheritance, and so on. This external model is based on an internal model which resembles a semantic network. Such a network may be extracted from a variety of data models, including object-oriented, entity-relationship and extended relational models. This architecture gives ViewFinder a large degree of model independence. The main construct of the external model are displays of objects (either classes or tokens), called *views*. Commands are available for efficient traversal of the database, displaying views of any class or token. To speed up repetitive searches, views may be synchronized: the user sets up several views, linked in a tree-like structure, so that when the information displayed in the root view is modified (e.g., scrolled by the user), the contents of the other views change automatically. Additional commands are available to search, order, aggregate and select the information displayed in a view, thus providing a simple query facility.

*The work of D'Atri and Tarantino was supported in part by the Commission of the European Communities under the projects ESPRIT 1117 (KIWI), ESPRIT 2424 (KIWIS) and AIM 2024 (MILORD). The work of Motro was supported in part by NSF grants No. IRI-8609912 and IRI-9007106

1 Introduction

To improve their usability and responsiveness most database systems offer their users a wide variety of interfaces, suitable for different levels of expertise and different types of applications. A particular kind of interface which is now commonly available are *browsers*. Browsers are intended for performing *exploratory searches*, often by *naive users*. Thus, they usually employ simple conceptual models and offer simple, intuitive commands. Ideally, browsing should not require familiarity with the particular database being accessed, or even preconceived retrieval targets. Browsing may be used to gain insight into the contents and organization of the searched environment, to arrive at specific retrieval targets, and even to satisfy such targets.

In this paper we describe a new browsing interface to databases, called ViewFinder. The design approach has been to provide maximum *functionality* and *generality* without sacrificing *simplicity*. These three design principles are explained below. A preliminary design of ViewFinder was described in [28].

Simplicity. As already mentioned, browsers must be simple to use. ViewFinder communicates with its users mostly by graphical means (e.g., menus and icons) and it requires very little typing. The ViewFinder model defines a small number of basic concepts and a small number of operations (altogether, the global menu has six options and the browsing menu has nine options). Indeed, it is possible to explore a database by using just two commands. There is emphasis on economy of style, and consistency and predictability of behavior; the authors believe that a graphical interface that employs a great number of different graphical constructs, can become just as complex and intimidating to use as the textual interface it is supposed to replace.

Functionality. Most database browsers define search processes that are indeed simple to perform, but simplicity is often achieved by providing low-level commands, and limited functionalities. Consequently, browsing sessions tend to be quite inefficient, and may become tedious after a while. ViewFinder incorporates several features that address these deficiencies. Additionally, where some browsers provide their users with graphical tools to browse in the database *scheme* and then construct simple queries, users of ViewFinder can browse in the database *extension* as well. For example, it is possible to enter the name of database value (or to "click" on this value if it appears somewhere on the screen) and receive a frameful of information on this value.

Generality. By and large, the approach to browsing has been ad-hoc: browsing is often understood only in terms of the specific user interface being constructed. Additionally, most browsers are designed to be used only with databases of a specific data model. The approach of ViewFinder is more formal and more general. Rather then define browsing in a given data model, ViewFinder defines a data model (with its associated operations) for browsing. The architecture of ViewFinder provides a large degree of model independence, which allows it to be used with databases of a variety of data models. This independence is achieved by a tiered architecture. ViewFinder employs two different models: the *external model* defines high level structures and operations; these are the structures that are displayed to users, and the operations with which users can manipulate these structures. This external model is mapped to an *internal model* whose structures are more primitive. These two generic models are realized in specific environments. At the user's end, the external model is interpreted with specific visualization structures and operations and implemented with a specific graphical windowing environment. At the database end, the internal model is interfaced to a specific database system; this database system can employ one of several possible data models. By reprogramming the presentation of the external model and the database interface of the internal model, the ViewFinder model can be implemented with different graphical user interfaces and with different database systems.

The underlying internal model of ViewFinder may be characterized as a semantic network, consisting of objects connected by binary relationships. Objects are merely names. It is through their relationships with other objects that information about objects is expressed. For each object, a structure called *view* is defined, that presents the relationships and objects that are *adjacent* to the given object in the network. These structured displays of data are the main component of the external model.

The information in each view is sorted into several *frames*. For example, the view of the class EMPLOYEE will show its superclasses (e.g., PERSON), its subclasses (e.g., MANAGER), its properties (e.g., SALARY), and its member tokens (e.g., JOHN). By pointing to particular object names in a displayed view, users may request to display other views, in addition to or in place of views currently displayed. For example, while viewing the class EMPLOYEE, users may request to view the class MANAGER or the token JOHN. Thus, users may navigate in the class hierarchy, and move freely between class and token objects.

To speed up repetitive searches, which otherwise could become very tedious, views may be *synchronized*: the user sets up several views, linked in a tree-like structure, so that when the information displayed in the root view is modified (e.g., scrolled by the user), the contents of the other views change automatically. Additional commands are available to search, order, aggregate and select the information displayed in a view. Indeed, these commands amount to a flexible, yet simple, *query* facility.

Whereas the internal model of ViewFinder resembles a semantic network, its external model constructs are essentially object-oriented: class and token objects, membership relationships between tokens and classes, generalization relationships between classes, inheritance, and so on. Note that the external model of ViewFinder is not a complete object-oriented model, as this concept is commonly understood,[1] because ViewFinder is not a complete database management system. Rather, the external model of ViewFinder supports those elements of object-oriented models that are relevant to browsing and presentation. This *object-oriented presentation* can be provided for specific object-oriented models, as well as for other data models that support generalization hierarchies and complex objects (e.g., various flavors of the entity-relationship or extended relational models). In each case the low-level semantic network would be *extracted* from the given database.

[1]For example, it does not have the equivalent of *encapsulation*.

In the remainder of this section we provide a brief survey of database browsing and relate it to ViewFinder. The organization of the rest of this article follows closely the tiers in the ViewFinder architecture. Sections 2 and 3 describe the internal and external models. Section 4 describes in detail our implementation of the external model; the implementation described is for the SunView windowing environment. Section 5 is devoted to the mapping between the internal model and other data models; the mapping between the internal model and ODMG, a generic object-oriented data model, is discussed in detail. Section 6 concludes with a brief summary and discussion of additional research directions.

1.1 Background

Most browsers assume a conceptual model that interleaves the data in a *network* of some kind, and browsing is done by *navigation*: the user begins at an arbitrary point in the network (perhaps a standard initial position), examines the data in that "neighborhood", and then issues a new command to proceed in a new direction.

An early example of this approach is the interface designed and implemented by Cattell at Xerox [6]. The interface is to an entity-relationship database [9], and it features a set of directives for scanning a network of entities and relationships, and presenting each entity, together with its context, in a display called *frame*. A similar interface was later provided for the entity-relationship database management system Cypress [7].

Cattel's model was further employed in the Living in a Database (LID) system [13]. But whereas Cattel's system relied on textual interaction on a character-based display, the LID system featured graphical interaction on a bit-mapped display. Users of LID "live" inside an instance of an Entity-Relationship database. At each point in time they "reside" in a particular tuple, from which they can view the data tuples and schema structures that are associated with it.

BAROQUE [27] is a browsing interface to relational databases. BAROQUE establishes a view of the relational database that resembles a semantic network (integrating both schema and data), and provides several intuitive commands for scanning it. When a user "visits" a particular node in the network (a data value) the system constructs a "frame" of information on this value, showing its relationships to other data values. The user can then pick a value mentioned in this frame and request to display its frame instead. In effect, the user is traversing an edge in the network to visit an adjacent node. BAROQUE was recently ported to a graphical environment of windows, menus and icons.

More recently, Cattell, Rogers and Learmont describe a system that provides an entity-relationship view of relational databases [23], and a graphical tool that, like BAROQUE, displays entities assembled from the underlying relational database [32]. The interface also provides limited update capabilities, and it supports the display of pictures as well as text.

Browsing is offered as the principal retrieval method for loosely-structured databases [25]. Such databases are heaps of facts that do not adhere to any conceptual design. Facts are named binary relationships between data values, so the data may be regarded as a semantic network of values. This loosely-structured model of databases has been adopted by USD [20], a system that provides database capabilities often required in scientific research. The main retrieval and browsing paradigm of its graphical user interface is the matching of *patterns* constructed by users.

Browsers have been developed for various relational systems, for example, SDMS [19], INGRES [34] and DBASE-III [3]. These are primarily tools for scanning relations (including relations that are results of formal queries), and therefore have only limited exploration capabilities. Browsing is confined to a single relation at a time, and it is not possible to browse across relation boundaries. If a user encounters a value while browsing, and wants to know more about it, he must determine first in which other relations this value may appear (quite difficult), then formulate a formal query, and resume browsing in the new relation.

OdeView is a graphical interface to Ode, an object-oriented database system and environment [2]. OdeView provides facilities for examining the class hierarchy, for navigational browsing, for displaying parts of objects, and for selecting objects that possess specific characteristics. One of OdeView's most important features is synchronized browsing. This feature in particular, and OdeView's navigation model in general, were inspired by KIVIEW [28], the predecessor of ViewFinder.

In the network structure assumed by most browsers, *adjacency* reflects *relationship.* For example, JOHN is linked to MANUFACTURING because John works in the manufacturing department, and MANUFACTURING is linked to DEPARTMENT because the former is an instance of the latter. A different approach is taken by Pintado and Tsichritzis [31] who propose to place objects in a space, so that their mutual *proximity* reflects their *affinity* (as computed from some measure). Users can then navigate among objects with the help of a two dimensional "map" that visualizes the distances between objects. By changing the measure, different maps can be derived for the same set of objects. Navigating from an object directly to "similar" objects is also a feature of the BAROQUE browser, discussed earlier. When viewing an object, a user may ask to display similar objects, where similarity is defined as having identical values for a set of properties specified by the user. Hence, similarity establishes an alternative network of objects, in which objects are adjacent if they are similar.

A database browser should allow its users to navigate freely in the database *extension*; e.g., its users should be able to examine data items and follow their relationships to other data items, or to name a data item and receive information about it. The next three systems are in a somewhat different class: they are essentially *schema browsers*, with built-in query facilities.

GUIDE [38] is a graphical user interface which is based an Entity-Relationship model extended with *isa* relationships. GUIDE's primary focus is to assist its users in the formulation of queries. Typically, a user identifies the relevant schema components, and the system constructs a textual query. GUIDE provides many facilities for panning, zooming, and hiding schema components, yet the layout of the database schema is essentially fixed (i.e., individual schema components cannot be repositioned relative to each other).

SKI [22] is a user interface to the Sembase database system, a system that implements a semantically-rich data model. Sembase's constructs include entity sets, single- or multi-valued functional relationships, and *isa* relationships [21]. SKI provides powerful graphical tools for browsing in the schema. Its schema layout is much more flexible than GUIDE's, enabling users to focus on the parts of the schema that are relevant to their retrieval goals. Simple querying is provided by allowing users to define ad-hoc subtypes and explore their contents (the "answers") in scrollable windows. However, SKI has no facilities for browsing through the data.

Closely related to SKI with respect to functionality and expressive power, SNAP [5] is a general purpose schema manager which uses a coherent paradigm to support schema design, schema browsing, and selection-type queries. SNAP is based on the IFO data model [1], an object-based semantic data model which provides features for the representation of simple and complex objects, functional relationships, and *isa* relationships. As in SKI, users operate on the schema of the database, which is displayed diagrammatically using different kinds of graphical shapes. For large database schemes, SNAP provides commands to reposition objects, hide and redisplay objects, pan and zoom over the schema, and reformat *isa* hierarchies and complex object representations. Users can also specify selection-type queries graphically. All answers returned by the system have the structure of non-first normal form relations, and are displayed as nested tables. Again, SNAP has no facilities for browsing in the data.

Schema design, schema browsing, and query formulation are offered also by ISIS [15], a graphical interface to a subset of the Semantic Data Model [17]. ISIS provides two alternative views on the schema, depending on whether a user wishes to focus on the *isa* hierarchy or on the attribute network. The presentation is again diagrammatic. However, unlike the systems previously discussed, structured graphical symbols are used to visualize complex objects; furthermore, patterns assigned to classes allow representation of semantic links in iconic format. Browsing is mostly at the schema level, though a limited form of navigation can be accomplished at the data level: scrollable windows list the instances of classes, and individual instances can be selected to be analyzed in more detail. However, the visualization of the entire state of an object is somewhat inconvenient, requiring users to follow the semantic connections one at the time.

Database browsing has often been described as *complementary* to database querying. A survey of the advantages and limitations of these two retrieval methods is given in [10], which also suggests several ways to bridge these two methods. The automatic inference of formal queries from information obtained in browsing sessions is the purpose of the Query-by-Browsing system [12]. Users of Query-by-Browsing are presented with listings of database records and mark them as relevant or irrelevant to their retrieval objectives. The system then infers the pattern underlying the selections by means of an inductive learning-based algorithm.

Whereas the subject of database querying has been approached formally, the approach to database browsing has been mostly informal. Query languages are often based on formal languages (e.g., relational algebra and calculus), and issues such as query language completeness or query language equivalence are often addressed. In contrast, database browsing is usually defined by means of a particular interface that has been constructed. A more theoretical study of browsing is given in [11], which proposes a formal environment for defining several browsing processes with increasing level of complexity.

More generally, the subject of user interfaces to databases, and browsers in particular, have been recognized recently as an area of research and development within the field of databases that faces important challenges and that deserves increased attention [29, 33].

Finally, it should be noted that the applicability of browsing is not limited to databases, and the method has been applied successfully in other environments. Browsers have been constructed for programming environments to support the development of software [14, 16, 18, 4], and in electronic reference and hypertext systems [37, 24, 30, 35].

2 The Internal Model

In this section we define the underlying internal model of ViewFinder. This model is not apparent to the users of ViewFinder, but is necessary for defining the structures and operations of the external model.

2.1 Objects, Relationships and Facts

The underlying internal model is a semantic network, consisting of *objects* connected by *binary relationships*. In many aspects it is similar to the model behind loosely-structured databases [25, 26]. Such a network may be defined by a set of triplets (m, r, n), where m and n are objects and r is a relationship. These triplets will be called *facts*; the first and third objects of a fact will be called, respectively, the *source* and the *target* objects of the fact.

There are two kinds of objects: *class* objects and *token objects*. An object is either a class or a token. Examples of class objects are EMPLOYEE and DEPARTMENT; examples of token objects are JOHN and MANUFACTURING. In addition to the classes that are specific to the application, there are several *type* classes: a class called DATABASE, and a set of classes that correspond to the common data types, such as INTEGER and STRING. In the following definitions the symbols a, b, c denote any class, t denotes a type class, x, y, z denote tokens, m, n denote objects (either tokens or classes), and r denotes a relationship.

There are four kinds of facts: *generalization* facts, *membership* facts, *intensional* facts, and *extensional* facts. These different kinds of facts are described below, together with nine requirements that must be satisfied by any set of facts.

2.2 Generalization and Membership Facts

A frequent relationship between classes is *generalization*: one class is more general than another class (the latter class is then a *specialization* of the former). The generalization relationship will be denoted $\prec$. Examples of generalization facts are (EMPLOYEE,$\prec$,PERSON) and (DEPARTMENT,$\prec$,UNIT).

A frequent relationship between tokens and classes is *membership*: a token is a member of a class. The membership relationship will be denoted $\in$. Examples of membership facts are (JOHN,$\in$,EMPLOYEE) and (MANUFACTURING,$\in$,DEPARTMENT).

We require that generalization and membership satisfy five requirements:

1. Root of generalization hierarchy:
   For every database class a other than DATABASE, $(a, \prec, \text{DATABASE})$ is a generalization fact, and for every database class a which is not a type class, there exists a single type class t other than DATABASE, such that $(a, \prec, t)$ is a generalization fact.

2. Required membership for tokens:
   For every token x there exists a class a which is not a type class and a membership fact $(x, \in, a)$. If $(x, \in, a)$ and $(x, \in, b)$ and $a \neq b$, then there exists a type t, such that $(a, \prec, t)$ and $(b, \prec, t)$.

3. Irreflexivity of generalizations:
   If $(a, \prec, b)$ is a generalization fact then $a \neq b$.

4. Transitivity of generalizations:
   If $(a, \prec, b)$ and $(b, \prec, c)$ are generalization facts, then $(a, \prec, c)$ is also a generalization fact.

5. Inheritance of membership over generalization:
   If $(x, \in, a)$ is a membership fact and $(a, \prec, b)$ is a generalization fact, then $(x, \in, b)$ is also a membership fact.

The transitivity requirement models the accepted real-world semantics of this relationship. For example, if PERSON is a generalization of EMPLOYEE, and EMPLOYEE is a generalization of MANAGER, then PERSON is also a generalization of EMPLOYEE. Together, transitivity and irreflexivity guarantee *acyclicity*, so that the generalization relationship imposes a *hierarchy* on the classes of the database. The first requirement guarantees that this hierarchy is connected and has a single root class, called DATABASE.[2] Immediately below the root there is a level of classes that correspond to types. Each type class is then the root of a separate generalization hierarchy. Types are used to restrict operations on each object to those that have been defined for its type. The second requirement states that every token should be a member of at least one non-type class, but a token may not be a member of

[2]In practice, the name of this root class would be different for each database.

classes of different types. Finally, the inheritance requirement guarantees that the set of member objects of each class is contained in the set of member objects of every more general class.

2.3 Intensional and Extensional Facts

An *intensional* fact associates two non-type classes with a relationship other that $\prec$; for example, (PERSON,AGE,YEARS). An *extensional* fact associates two tokens; for example (JOHN,AGE,32). The semantics of an intensional fact is that any extensional fact with source and target tokens that are members of its source and target classes, respectively, is admissible. Thus, an intensional fact defines a *domain* of extensional facts.

Given a fact (m, r, n), which is either intensional or extensional, the pair (r, n) will be referred to as a *property* of m. Thus, the class PERSON has the (intensional) property (AGE,YEARS), and the token JOHN has the (extensional) property (AGE,32).

Extensional facts must belong to the domain of at least one intensional fact; that is, for each extensional fact there must be an intensional fact and two membership facts that associate the tokens of the extensional fact with the respective classes of the intensional fact. Formally,

6. Required membership for extensional facts:
   If (x, r, y) is an extensional fact, then there exist non-type classes a and b, membership facts $(x, \in, a)$ and $(y, \in, b)$, and an intensional fact (a, r, b).

Intensional facts may possess two characteristics. Some intensional facts are designated as *mandatory*: there *must* be an extensional fact that associates *each* of the members of the source class with a member of the target class. Some intensional fact are designated as *single-valued*: for each member of the source class there may be at most one extensional fact that associates it with a member of the target class. Formally,

7. Instantiation of extensional facts for mandatory intensional facts:
   If (a, r, b) is a mandatory intensional fact, then for every membership fact $(x, \in, a)$ there exist a membership fact $(y, \in, b)$ and an extensional fact (x, r, y).

8. Instantiation of extensional facts for single-valed intensional facts:
   If (a, r, b) is a single-valued intensional fact, then for each membership fact $(x, \in, a)$ there exists at most one pair of membership fact $(y, \in, b)$ and extensional fact (x, r, y).

An important characterization of intensional properties is that they are inherited over generalizations. Formally,

9. Inheritance of intensional properties over generalizations:
   If (a, r, b) is an intensional fact and $(c, \prec, a)$ is a generalization fact, then (c, r, b) is

also an intensional fact. Similarly, if (a, r, b) is an intensional fact and $(c, \prec, b)$ is a generalization fact, then (a, r, c) is also an intensional fact.

As an example, consider these three generalization relationships: MANAGER$\prec$EMPLOYEE, TEMPORARY$\prec$EMPLOYEE, and PRIVATE-OFFICE$\prec$OFFICE-SPACE. The inheritance rule guarantees that if office space is available to employees, then it is available to managers; and if employees may have office spaces, then they may have private offices as well.

These definitions lead to various conclusions. When an intensional property is propagated to the subclass of the source, it *preserves* the characteristics of being mandatory or single-valued. For example, if office space is available to every employee, then it is available to every manager; and if employees may not have more than one office space, then this restriction also applies to managers. When an intensional property is propagated to the subclass of the target, it preserves the characteristic of being single-valued, but not necessarily the characteristic of being mandatory. For example, if employees may not have more than one office space, then they may not have more than one private office; but when employees are guaranteed office spaces, they are not guaranteed private offices.

When an intensional property is propagated to the subclass of the source, it may *acquire* the characteristics of being mandatory or single-valued. For example, office space may be available to employees, and guaranteed for managers; and several office spaces may be available to employees in general, but only a single office space to temporaries. When an intensional property is propagated to the subclass of the target, it could acquire the characteristic of being single-valued, but never the characteristic of being mandatory. For example, multiple office spaces may be available to employees, but at most one private office; however, if office space is not guaranteed to every employee, then private office could not be guaranteed.

Rules 4–7 and 9 are illustrated graphically in Figure 1. The following graphical conventions are used: Class objects are indicated with squares, token objects with circles. Relationships are indicated with arrows directed from the source object to the target object. Mandatory relationships are indicated with thick arrows, all other relationships with thin arrows. Each of these rules was phrased as an implication, stating that if certain facts exist, then other facts are implied: existing facts are indicated with solid relationship arrows, implied facts with dashed relationship arrows.

2.4 Immediate and Distant Objects and Properties

Rules 4, 5 and 9 (the transitivity of generalizations, the inheritance of membership over generalizations, and the inheritance of intensional properties over generalizations) may be regarded as *closure* properties of the generalization relationship. In these cases, it is useful to distinguish between *immediate* and *distant* relationships.

A class a is an *immediate generalization* of a class b, if $(b, \prec, a)$ and there is no class c such that $(b, \prec, c)$ and $(c, \prec, a)$. For example, EMPLOYEE is an immediate generalization of

$$\begin{tikzcd}
\boxed{b} \arrow[r, "\prec"] & \boxed{c} \\
\boxed{a} \arrow[u, "\prec"'] \arrow[ur, dashed, "\prec"'] &
\end{tikzcd}$$

Rule 4

$$\begin{tikzcd}
\boxed{a} \arrow[r, "\prec"] & \boxed{b} \\
x \arrow[u, "\in"'] \arrow[ur, dashed, "\in"'] &
\end{tikzcd}$$

Rule 5

$$\begin{tikzcd}
\boxed{a} \arrow[r, "r"] & \boxed{b} \\
\boxed{c} \arrow[u, "\prec"'] \arrow[ur, dashed, "r"'] &
\end{tikzcd}$$

Rule 9a

$$\begin{tikzcd}
\boxed{a} \arrow[r, "r"] \arrow[dr, dashed, "r"] & \boxed{b} \\
& \boxed{c} \arrow[u, "\prec"']
\end{tikzcd}$$

Rule 9b

$$\begin{tikzcd}
\boxed{a} \arrow[r, dashed, "r"] & \boxed{b} \\
x \arrow[u, dashed, "\in"'] \arrow[r, "r"] & y \arrow[u, dashed, "\in"']
\end{tikzcd}$$

Rule 6

$$\begin{tikzcd}
\boxed{a} \arrow[r, "r"] & \boxed{b} \\
x \arrow[u, "\in"'] \arrow[r, dashed, "r"] & y \arrow[u, dashed, "\in"']
\end{tikzcd}$$

Rule 7

Figure 1: Rules 4–9

MANAGER, if it is a generalization of MANAGER, but not a generalization of any other class which is more general than MANAGER.

A token x is an *immediate instance* of a class a, if $(x, \in, a)$ and there is no class b such that $(b, \prec, a)$ and $(x, \in, b)$. For example, JOHN is an immediate member of EMPLOYEE, if it is a member of EMPLOYEE, but not a member of any class which is more specific that EMPLOYEE.

A property (r, b) is an *immediate intensional property* of a class a, if (a, r, b) is an intensional fact and there is no class c such that $(a, \prec, c)$ and (c, r, b) is also an intensional fact, and there is no class d such that $(b, \prec, d)$ and (a, r, d) is also an intensional fact. For example, (WORK-FOR,DEPARTMENT) is an immediate intensional property of EMPLOYEE, if it is an intensional property of EMPLOYEE, but not an intensional property of any class which is more general than EMPLOYEE, and EMPLOYEE does not have an intensional property with relationship WORK-FOR and a target class which is more general than DEPARTMENT.

Since the characteristics of intensional properties may be acquired after propagation, we need to define separately the immediate intensional properties which are mandatory or single-valued.

A property (r, b) is an *immediate mandatory intensional property* of a class a, if (a, r, b) is a mandatory intensional fact and there is no class c such that $(a, \prec, c)$ and (c, r, b) is also a mandatory intensional fact. For example, (SALARY,AMOUNT) is an immediate mandatory property of EMPLOYEE, if it is a mandatory property of EMPLOYEE, but not a mandatory property of any class which is more general than EMPLOYEE. A property (r, b) is an *immediate single-valued intensional property* of a class a, if (a, r, b) is a single-valued intensional fact and there is no class c such that $(a, \prec, c)$ and (c, r, b) is also a single-valued intensional fact, and there is no class d such that $(b, \prec, d)$ and (a, r, d) is also a single-valued intensional fact. For example, (OFFICE,PRIVATE-OFFICE) is an immediate single-valued property of MANAGER, if it is a single-valued property of MANAGER, but not a single-valued property of any class which is more general than MANAGER, and MANAGER does not have a single-valued property with relationship OFFICE and a target class which is more general than PRIVATE-OFFICE.

Objects and properties that are related to a given object, but are not immediate, will be referred to as *distant*.

Finally, a *database* is a set of generalization facts, membership facts, intensional facts (some of which are mandatory and/or single-valued) and extensional facts, over a set of tokens, a set of classes and a set of relationships, that satisfy the nine requirements described above.

3 The External Model

The *external model* (the user model) defines higher level structures and operations. These are the structures that are displayed to users and the operations with which users can manipulate these structures.

3.1 Views, Frames and Windows

For every database object, we define a structure called *view*. This structure is assembled from facts in which this object participates, and it has several independent components called *frames*. The definition of views is different for class objects and token objects.

Let a be a class object. The view of a is defined as four frames:

1. *members*: the tokens that are immediate members of a.
2. *superclasses*: the classes that are immediate generalizations of a.
3. *subclasses*: the classes that are immediate specifications of a.
4. *properties*: the immediate intensional properties of a (properties that are mandatory or single-valued are marked as such).

Note that the members of the class in view are those shown in the *members* frame, as well as the members of any subclass. Similarly, the subclasses of the class in view are those shown in the *subclasses* frame, as well as their subclasses; and the superclasses are those shown in the *superclasses* frame, as well as their superclasses. Similarly, the intensional properties of the class in view are those shown in the *properties* frame, as well as the intensional properties of any superclass, and the intensional properties obtained from the intensional properties shown by substituting subclasses in the target.

For example, a class EMPLOYEE may have the following view (m means mandatory and s means single-valued):

EMPLOYEE

members	*superclasses*	*subclasses*	*properties*		m	s
ADAM	PERSON	INACTIVE	WORK-FOR	DEPARTMENT	$\surd$	$\surd$
BETTY		MANAGER	POSITION	TITLE	$\surd$	$\surd$
FRANK		TECHNICAL	SALARY	AMOUNT	$\surd$	$\surd$
MARY		TEMPORARY	OFFICE	ROOM		$\surd$
TOM			SECRETARY	EMPLOYEE		

Let x be a token object. The view of x is defined as two frames:

1. *classes*: the classes of which x is immediate member.
2. *properties*: the extensional properties of x.

For example, a token JOHN may have the following view:

JOHN

classes	*properties*	
EMPLOYEE	WORK-FOR	MANUFACTURING
PARENT	POSITION	SUPERVISOR
	SALARY	46,000
	OFFICE	MB475
	AGE	32
	CHILD	JEFFREY
	CHILD	JULIE

Note that different real-world objects should be modeled by database objects with different names, or else information about different real-world objects would be combined into a single view. Using the same name for different real-world objects may cause other modeling problems as well. For example, consider the intensional facts (CITY,POPULATION,CITIZENS) and (COUNTY,POPULATION,CITIZENS). If the object FAIRFAX is chosen to designate both the city of Fairfax and the county of Fairfax, then it would be impossible to associate the two extensional facts that state the actual populations of the city and the county of Fairfax with their corresponding intensional facts (also, the single-valuedness of these intensional facts would be violated).

Finally, a *window* is a display structure for presenting the contents of a frame. A window presents only a fixed interval of elements of the frame; a specific position in this interval (e.g., the first, middle or last position) is established as the *distinguished position.* A frame may be presented in several independent windows simultaneously. Each window displays a "version" of the frame, which may be subjected to useful manipulations, such as ordering and selection. These and other manipulations are described next.

3.2 Basic View Operations

The basic operations that may be applied to views are: *display*, *scroll*, *search*, *order*, *aggregate*, *selection*, and *closure.*[3]

The most elementary browsing operation is to *display* a frame of an object in a window (to "open" a window). The particular object must be specified by the user, either by typing

[3]Although we shall refer to an operation as being applied to window or a frame, it is always applied to a *copy* of a frame which is associated with a particular window.

its name, or, more often, by selecting it from some window which is already open. When an object is thus specified, the system selects a default frame and presents it in a window. The user can then replace the contents of this window with any other frame of the same object, or he can open another window with any other frame of the same object.

The process of repeatedly opening new windows on objects that are referenced in current windows (and closing some windows as well) corresponds to *navigation* in the underlying semantic network. When a node m of the network is visited, a portion of its immediate neighborhood is displayed in window. Opening a new window on an object n which is referenced in a window on m, corresponds to crossing an edge from m to visit its adjacent node n.

The number of elements displayed in each window is limited by the size of the window. The user may control the frame interval that is currently being displayed, either by the sequential process of *scrolling* the frame in the window, or by the direct process of *searching* a particular element and displaying it in the distinguished position.

The default *order* of the elements (tokens, classes, or properties) is implied by the order in which they have been arranged in the database, but each frame may be reordered by the user. Frames of classes (the *superclasses* and *subclasses* frames in the view of a class, and the *classes* frame in the view of a token) may be ordered lexicographically by the name of the class. Frames of properties (the *intensional properties* in the view of a class, and the *extensional properties* in the view of a token) may be ordered lexicographically by the name of the relationship. If several properties have the same relationship, they are ordered by the target object. In the case of intensional properties, the targets are classes and a lexicographical order is used; in the case of extensional properties, the targets are tokens, and the order is determined by the type class to which these tokens belong (e.g., numerical for the class NUMBER, lexicographical for the class STRING). Frames of tokens (the *members* frame in the view of a class) may be ordered by an order which is determined by the type class to which the tokens belong.

In addition, frames of tokens may also be ordered with respect to the target token of any single-valued property of that class.[4] For example, the members of PERSON may be ordered by AGE. This kind of order is extensible in two ways. By using aggregate functions, multi-valued properties may be used as well. For example, the members of PERSON may be ordered by the total number of occurrences of the property (CHILD, x). Also, orders on several properties may be combined in sequence to resolve the ordering of members that have the same value by the previous orders. For example, the members of EMPLOYEE can be ordered by POSITION and LEVEL (within a position).

Each of these orders may be specified as either ascending or descending. Note that the same frame may be displayed in multiple windows, each using a different order.

[4]If the property is not mandatory, the elements for which this property is not defined are grouped at the end and marked appropriately.

Several standard *aggregate* functions may be computed on the elements of a frame. A *count* function that computes the total number of elements is defined for all frames, *maximum* and *minimum* functions that compute, respectively, the element with the lowest and highest value (with respect to the prevailing order) are available for frames of tokens or classes, and *sum* and *average* functions that compute, respectively, the sum and average of all the elements are available for the *members* frame of classes that are subclasses of the class NUMBER.

Frames of tokens allow users to apply the numerical functions (i.e., *maximum*, *minimum*, *sum*, and *average*) also to any numerical property of the class.[5] For example, when viewing the *members* frame in the view of PERSON, users may request the average AGE.

The set of elements of a frame may be restricted temporarily to a selected subset, and future operations on this frame will ignore all other elements. There are two basic *selection* methods. In the first method, which is available for all frames, the user *enumerates* the subset, either by specifying individual elements, or by specifying the end-points of intervals of elements. In the second method, which is available only for *members* frames, the user specifies a *property* and a *condition*, selecting all the tokens whose values for the property satisfies the condition. A variation of this method does not require a condition, and selects all the tokens for which this property is defined. Evidently, this feature resembles a simple querying tool. The effect of selection can be canceled.

As defined, the various frames include only immediate objects and properties. At times, it may be desirable to view distant objects and properties as well. A *closure* operation is available for augmenting the contents of the window with distant objects and properties. The effect of closure can be canceled.

3.3 Synchronization

In the process of browsing, the different windows, once opened, are independent: the information displayed in any window may be modified without any effect on the information displayed in the other windows. *Synchronization* allows users to set up several windows, connected in a tree-like structure, so that when the information displayed in any window of this tree is modified (by scrolling or searching), the contents of its descendent windows change automatically.

Let W_1 be an open window, displaying the object name p_1 in its distinguished position. Assume that the user opens a second window W_2 by selecting p_1 in W_1. W_1 and W_2 are then called *parent* and *child* windows, respectively.[6] The operation $sync(W_1, W_2)$ establishes a synchronization link between the child window and its parent window: W_2 will always display a frame of the object appearing in the distinguished position of W_1.

[5]An intensional property whose target class is a subclass of NUMBER. This property need not be mandatory or single-valued.

[6]When W_2 is used for opening another window W_3 on the same object, W_1 would still be the parent of W_3.

Such synchronization links may be repeated to form a *tree* of windows (note that the children of a window are always synchronized with the *same* object in the parent window). When the information in one of these windows is modified, changes will be made to all its *descendent* windows.

Synchronization is terminated with the operation *break*(W), where W is an open window. This operation breaks the synchronization between W and its parent. Synchronization is broken automatically when the parent window is closed or is changed to display another frame, or when the child window is closed.

The precise effect of synchronization depends on the specifics of the child window W_2. When synchronization is established W_2 has specific *window attributes*. As much as possible, these attributes should be maintained when W_2 is refreshed to display new objects. When the same attributes cannot be maintained, a policy should establish the new attributes. There are five such window attributes: *frame* (the specific frame which appears in the window), *closure* (whether the window shows the closured set of items or only the immediate items), *ordering* (the ordering of the frame), *interval* (the specific interval of the frame which is displayed in the window), and *selection* (the selection criterion applied to the frame).

The policy for determining the attributes of a refreshed window could be quite complex. Consider, for example, the interval attribute. A window of *members* will be refreshed to show the interval occupying the same relative position in the new frame; e.g., if the original frame has 200 objects, with objects 41 through 60 displayed in the window, and the new frame has only 100 objects, the refreshed window will now display objects 16 through 35. A window of *properties* will be refreshed to show the same property in its distinguished position. If this property does not exist for the new object, the next property in the present ordering would be shown; unless this window itself has a child window synchronized with it, in which case this window will be scrolled until its distinguished position is empty (and its child window will display the window NULL).

We illustrate the use of synchronization with a description of a browsing session that involves three windows.

First the user opens a window with the *subclasses* frame of EMPLOYEE and scrolls it until the object MANAGER appears in the distinguished position. Next, the user opens a window with the *members* frame of MANAGER, and scrolls it until the object HARRY appears in the distinguished position. Finally, the user opens the *properties* frame of HARRY and scrolls it until the property (OFFICE,AV678) appears.

The user now synchronizes the third window with the object HARRY in the second window, and the second window with the object MANAGER in the first window. Figure 2 illustrates the resulting situation.

Having established these displays, the user now browses in the list of managers by scrolling the second window. While browsing, the third window changes automatically to display information on each manager (in particular, the office of the manager).

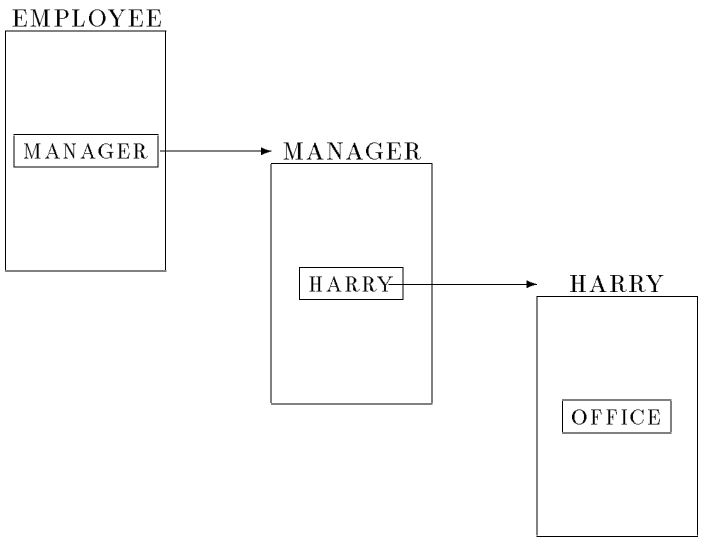

Figure 2: Three synchronized windows

When finished, the user returns to the first window and scrolls the list of subclasses, replacing MANAGER by TECHNICAL. The second window changes automatically to display information on this class, and the third window changes automatically to display information on a particular member of this class. The user now returns to the second window, to browse in the list of members of the technical staff.

Note that if there are no members of the technical staff, the second window that is used to drive the third window will be empty. In this case the third window (and possibly other descendent windows) will display the window NULL. If TECHNICAL is later replaced by TEMPORARY in the first window, and there are temporary employees, the third window will change to display the properties of a particular member of this class.

4 The User Interface

The pilot implementation of ViewFinder implements the external model described in Section 3 in a multi-window graphical environment, and in this section we describe this specific user interface in detail.[7] A multi-window graphical environment provides the most natural and effective interface, but other user interfaces are possible as well; for example, a simple textual interface in which the system outputs all data and menus to a simple terminal, and users input their choices by typing. The present implementation of ViewFinder is based on a "kernel" system that permits easy construction of multiple user interfaces.

[7]This user interface is a complete implementation of the external model; yet one could conceive a more powerful graphical implementation.

4.1 The Conventions of ViewFinder

ViewFinder is implemented as a SunView [36] application program. Since conceptual simplicity is a fundamental requirement of ViewFinder, it employs a relatively small subset of the rich set of communication primitives supported by SunView. For the most part, these primitives are common to all windowing systems. The basic primitives used in ViewFinder are listed below (the terminology is slightly different from that of SunView).

- **Cursor**: A screen indicator, whose position is controlled by a movement of a mouse. The action of depressing a mouse button will be referred to as *pointing* (at the item or area where the cursor is located).
- **Button**: a small labeled box. The action as pointing at a button will be referred to as *depressing* the button.
- **Display window**: A framed rectangular area of the screen, with a horizontal header at its top, and a vertical scroll-bar at its left. The header shows the label of the display window; the scroll-bar enables users to control the contents of the display window with the aid of the mouse. For brevity, we shall often refer to display windows simply as windows.[8]
- **Menu**: A framed list of choice items. Menus pop up when the user points at predetermined areas of the screen (for example, the header of a window) and disappear when the mouse button is released. When the user points at a menu item, it is emphasized, and if the button is released over an emphasized item, this item is *selected.* A menu item may show an arrow at its right hand side: if the user points at the arrow, then another menu pops up (a "pull-right menu"); the user can thus "walk" through menus, until a final selection is made.
- **Communication box**: a framed rectangular area of the screen, used for exchanging information between the user and the system. Examples are:
  - **Selection box**: a box with an array of buttons. It pops up in situations when the user must select an option. It disappears after a selection is made.
  - **Message box**: a box with a ViewFinder message and a button labeled "close". It pops up to display the result of a computation or an error message. It disappears when the button is depressed.
  - **Confirmation box**: a box with two buttons labeled "confirm" and "cancel". It pops up when the user initiates certain actions whose effect is relatively substantial. The buttons are used to confirm or cancel the initiated action. After either button is depressed the box disappears.

[8]Recall that *window* is also a user-model term, referring to a particular version of a frame in the view of an object. This should cause no confusion, as the context would allow to infer the proper interpretation of the term. Indeed, the interpretations can usually be identified, as frame windows will always be presented in display windows.

- **Input box**: a box with a prompting message. It pops up when keyboard input must be obtained from the user. It disappears when the user terminates the input by depressing the "return" key.
- **Structured box**: a box that incorporates several of the above four kinds of communication boxes.

When the system is requested to open a view of an object (a token or a class), it selects one of the frames of this view, and displays the initial interval of elements in a window of standard size. The first position of the displayed interval is the *distinguished position.* Every window is assigned a unique identification number. The window has a header labeled $object\_name.frame\_name$ and $window\_id(parent\_window\_id)$.[9] A scroll bar along the left border allows users control over the portion of the frame that is displayed in the window.

The frame which is opened by default is determined by establishing an intuitive order among the various frames, and opening the first non-empty frame. For class objects the order is: *subclasses*, *members*, *properties*, *superclasses*; for token objects: *properties*, *classes.* Recall that Rule 1 of the internal model guaranteed that each class has at least one superclass, and Rule 2 guaranteed that each token belongs to at least one non-type class. Thus, the default frame is guaranteed to be non-empty. The system then fills the window with the initial interval of the frame. The elements are arranged vertically, with object names as buttons and relationship names as text.

By depressing these buttons, the user requests the system to display the corresponding objects in new windows. The window in which a button is depressed becomes the *parent* of the new window.[10] By pointing at the background and at the header areas of each window, the user may bring up menus for performing additional operations. These menus are described in the next two subsections.

4.2 Global Commands

When ViewFinder is invoked, a single window appears, covering the entire screen. This window is displayed throughout the entire session, and it encompasses all activity. A row of buttons at the top of this window offers six *global commands.* The result of depressing these buttons is described below.

4.2.1 database

When **database** is depressed, a selection box appears, with a column of buttons labeled with the available databases. By depressing a button, the user opens that database (the

[9]The definition of a parent window will be provided shortly.

[10]An alternative method, in which the user types in directly the name of the new object, is described in the next subsection. In this case the new window does not have a parent.

database currently open, if any, is closed). The name of this database is also the name of the root class of its generalization hierarchy, and its *subclasses* frame is opened in a window. In this specific case only, the list does not show the subclasses (which are type classes, such as STRING or NUMBER), but *their* subclasses. This default frame resembles a "directory" of this database.

Figure 3 shows the ViewFinder screen at the beginning of a session. After issuing the **database** command from the global menu, the user selected the PERSONNEL database, and the *subclasses* frame of the root class PERSONNEL is displayed in a window.

4.2.2 view, restore

When **view** is depressed, an input box appears, and the user is prompted for the name of an object, which is then displayed in a window. Such "open by name" is an alternative to the usual "open by pointing", and is useful when the user wishes to view an object which is not referenced anywhere on the screen.

The command **restore** causes a selection box to appear with a column of buttons labeled with the names of the windows most recently closed. By depressing a button, the user redisplays that window, exactly as it was when closed. This command is used together with the **close** command, to be described later.[11]

4.2.3 refresh, help, quit

The command **refresh** synchronizes all windows with their parent windows. This command is used together with the **sync** command, to be described later.

The command **help** causes a selection box to appear, with a column of buttons labeled with the basic operations. Depressing any of these buttons causes a message box to appear, with specific information on this operation.

The command **quit** terminates the ViewFinder session. Confirmation is required.

4.3 Browsing Commands

When the user points to the background of a window which displays a frame in the view of an object, the principal menu of ViewFinder appears. The nine commands in this menu implement the functionalities described in Section 3.

[11]By closing and restoring windows, users can work around the limit imposed by SunView on the number of open windows.

4.3.1 open, close, replace

The command **open** opens another window for displaying another frame (possibly the same frame) of this object. The new window will have the same parent as the current window. A pull-right menu lists the possible choices: **superclasses**, **subclasses**, **members** and **properties** for a class object, and **classes** and **properties** for a token object.

The command **close** closes the current window. Recall that this window can be restored later by using the global command **restore**.

The command **replace** replaces the frame currently displayed in this window with another frame of this object. A pull-right menu lists the possible options: **superclasses**, **subclasses**, **members** and **properties** for a class object, and **classes** and **properties** for a token object.

Assuming the situation shown in Figure 3, consider the following actions. First, the object PERSON is selected from the PERSONNEL window, thus opening a second window with the *subclasses* frame of PERSON. Then, the object EMPLOYEE is selected from the PERSON window, thus opening a third window with the *subclasses* frame of EMPLOYEE. Next, the **open** command is issued to open a fourth window with the *members* frame of EMPLOYEE.[12] Finally, the **replace** command is issued, and the user is now considering the options for the frame that will replace *members* (Figure 4).

4.3.2 search

The command **search** scrolls the contents of the window to a specified item. A pull-right menu appears with four options: **forward** and **backward**, **next** and **previous**. If either **forward** or **backward** is selected, an input box appears, prompting the user for a string of characters. The window is then scrolled either forward or backward, until the first element (object or property) that matches this string occupies the distinguished position. If no occurrences of this string are found, the window is not scrolled, and a message is displayed in a box. If either **next** or **previous** is selected, the search is repeated for the previous search string, in either forward or backward direction. Search strings may include the wildcard character "*", which matches any substring.

Assume that, while viewing the *members* frame of EMPLOYEE, the user issues the **search** command and its **forward** option, and types the string JOHN into the input box. The *members* frame is scrolled forward until the first occurrence of this string is encountered. The user issues **search** again, and is now considering the different search options (Figure 5).

[12]Note that only immediate members of EMPLOYEE are listed; i.e., those who are not members of any of its four subclasses.

4.3.3 order

The command **order** modifies the ordering of the contents of the window. A frame-dependent pull-right menu offers the available options. For frames of tokens the menu lists the options **new** and **refine**; each of these options has its own pull-right menu with the options **by-property** and **by-name**; and each of these options has its own pull-right menu with the options **ascending** and **descending**. Thus, the user selects one of eight combinations.

The option **new** is for establishing a new order for the frame, while the option **refine** is for establishing a secondary order to resolve the order between elements that have the same position according to the prevailing order. The option **by-name** orders the members according to an order that is implied by the type. The option **by-property** orders the members by the values they possess for some property. In the latter case, a selection box appears listing the closure of the properties of this class, and the user is asked to select a property. If the user selects a multi-valued property, a second selection box appears listing various aggregate functions (see below) that are applicable to the type of the property, and the user is asked to select a function. The tokens will be ordered by the aggregate of the values they possess for this property. The options **ascending** and **descending** determine whether the items are listed from lowest to highest, or conversely. If the frame is ordered by a property which is not mandatory, the tokens that do not possess this property are marked and positioned at the end, and a suitable message appears.

For frames of classes or properties, the pull-right menu lists only the options **ascending** or **descending**. Classes are always ordered lexicographically; properties are ordered lexicographically by the relationship; properties that have the same relationship are ordered according to the target object (if the targets are classes, the order is lexicographic; if the targets are tokens, the order is determined by the type).

Assume that, while viewing the *members* frame of EMPLOYEE, the user issues the **order** command and the options **new**, **ascending** and **by-property**. When presented with a selection box with the properties of EMPLOYEE, the user selects SECRETARY which is multi-valued, and was then presented with a selection box of possible aggregate functions (Figure 6). Selecting **count** will cause the employees to be reordered by the number of their secretaries.

4.3.4 aggregate

The **aggregate** command computes functions on the set of items in the current frame and displays the results in a message box. For frames of tokens, a pull-right menu lists the options **by-name** and **by-property**. The option **by-name** is for computing aggregates of the tokens themselves, while the option **by-property** is for computing aggregates of some property of the tokens. In the latter case, a selection box appears listing the properties of this class, and the user is asked to select a property (it need not be mandatory or single valued). For either option a selection box appears listing the available aggregate functions: **count**,

maximum and **minimum** for tokens or properties of all types, and, in addition, **sum** and **average** for tokens or properties of the type NUMBER. For frames of classes or properties only a selection box appears, listing the functions **count**, **maximum** and **minimum**.

Results of aggregate functions are displayed in a message box. When the function **maximum** or **minimum** by some property is applied to a frame of tokens, the message box includes also the tokens that possess the minimal or maximal value.

Assume that, while viewing the *members* frame of EMPLOYEE, the user wishes to find the total number of employees. The user issues the command **aggregate** with its option **by-name**, and is then presented with a selection box of possible aggregate functions. Selecting **count**, the user is presented with a message box showing the total number of employees (Figure 7).

4.3.5 select

The command **select** restricts the items of a frame to a selected subset. A frame-dependent pull-right menu offers up to eight options. For frames of tokens it lists the options **by-enumeration**, **by-existence**, **by-condition**, **select**, **show-all**, **restore**, **undo** and **quit**. For all other frames, it offers the same options, except **by-existence** and **by-condition**.

The options **by-enumeration**, **by-existence** and **by-condition** are for *marking* items to be selected. When **select** is selected all items are initially marked.

When **by-enumeration** is selected, a selection box appears with these buttons: **to-top**, **single**, **to-bottom** and **quit**, with **single** being the default selection. In this mode, depressing any marked button in the frame unmarks that token, and depressing any unmarked token marks it. If the mode is changed to **to-top** or **to-bottom**, depressing a marked (unmarked) button in the token frame unmarks (marks) all tokens between the token and the first or the last token. When the user has finished marking, he presses **quit**, to return to the main **select** menu.

Assume that, while viewing the *members* frame of EMPLOYEE, the user wishes to restrict the list to certain employees. The user issues the command **select** with its option **by-enumeration**. Figure 8 shows the marking process: elements with "*" in their button have already been marked.

When **by-existence** is selected, a selection box appears listing the non-mandatory properties, and a **quit** button. When a property button is depressed, all the tokens that possess a value for this property are marked. **quit** returns the user to the main **select** menu.

When **by-condition** is selected, a structured box appears with four boxes. One selection box lists all the properties. A second selection box lists numerical comparators ($=, \neq, <, >, \leq$ and $\geq$), logical connectors (**and**, **or** and **not**), opening and closing parenthesis, and two buttons labeled **clear-condition** and **quit**. The other two boxes are an input box and a message box. The user constructs the selection condition by depressing property buttons, comparators, connectors and parenthesis. After depressing a comparator, the user must enter a value in the input box. The message box displays the selection condition as it is being constructed. The construction process may be restarted by depressing **clear-condition**. **quit** returns the user to the main **select** menu.

Assume that, while viewing the *members* frame of EMPLOYEE, the user wishes to restrict the list to employees with salary over 45,000. Figure 9 shows the process of constructing the selection formula.

These three marking processes can be combined, providing a flexible tool for selecting the items to be retained and the items to be removed. The other options of this menu perform the actual selection. The command **select** removes all the unmarked items from the window. The command **show-all** reverses the effect of **select**: all unmarked items are returned to the window. Marking and selection commands may be interleaved to provide a process of incremental filtering. The command **undo** allows users to cancel the effect of the most recent selection: if performed immediately after **select**, it restores the situation after the next-to-last selection; otherwise, it restores the situation after the last selection. The command **restore** shows all the items of the frame and marks them, allowing users to start a new selection process. The **quit** command terminates the selection session: the present window is retained and markings disappear. Subsequent selection sessions will resume the situation just before the **quit** command was entered. A window that has been subjected to selection will have the word SELECT appended to its header label; e.g., EMPLOYEE.MEMBERS.SELECT.

4.3.6 closure

The command **closure** augments the window with distant classes, members or properties. A pull-right menu offers two options: **all** and **immediate**. Selecting **all** augments the frame with all distant classes, members or properties, as appropriate. Selecting **immediate** cancels the closure operation.[13] A window that has been subjected to closure will have the word ALL appended to its header label; e.g., EMPLOYEE.PROPERTIES.ALL.

Assume that, while viewing the *properties* frame of EMPLOYEE, the user issues a **closure** command and selects the option **all**. The frame is then augmented with all intensional properties of EMPLOYEE, for example, (AGE,YEARS) or (CHILD,DEPENDENT) (Figure 10).

4.3.7 sync

The command **sync** synchronizes the window with its parent window. The command requires that the window is a child of a still-open parent window (i.e., recall that a child window is created by pointing to the object in the distinguished position of another window, the parent window). A pull-right menu offers three options: **auto-refresh**, **demand-refresh**

[13]When **closure** is applied to a frame that has been subjected to **select**, the frame is augmented with the closure elements of the entire frame, not only those of the selected frame.

and **break**. The first two options regulate the refresh policy of this window. With automatic refresh, the window will be refreshed whenever the contents of the parent window is changed. With demand refresh, the window will only be refreshed when the user depresses the global **refresh** button. This way, the user can scroll the parent window, without affecting the child window, until he has located the desired element. **break** breaks the synchronization between this window and its parent window.

As discussed in Section 3.3, when a child window is refreshed during synchronization, it should preserve the attributes of the former window; i.e., show the same frame, display a "similar" interval within that frame, and maintain its closuring, ordering and selection choices. Presently, only limited inheritance is maintained.

An example of synchronization is shown in Figures 11 and 12. First, the user opens *members* frame of DEPARTMENT and scrolls it until MANUFACTURING occupies the top (distinguished) position. Then, the user opens the *properties* frame of MANUFACTURING and scrolls it until the MANAGER property occupies the top position. Finally, the user opens two windows on HARRY showing its properties and classes. Figure 11 shows the present situation. The user now issues the **sync** command with its **demand-refresh** option in the last three windows. The third and fourth windows are now synchronized with the second window, which is synchronized with the first window. When the list of departments in the root window is scrolled, the properties of the "next" department, and the properties and classes of its manager will be displayed in the three descendent windows. Figure 12 shows the situation after the root window has been scrolled up one line.

4.4 Layout Commands

When the user points to a window header, the standard SunView menu pops up with six commands to control the layout of the current window: **close**, **move**, **resize**, **expose**, **hide** and **quit**.

Note that **close** and **quit** both close the window, but the former encapsulates it in an icon that may later be reopened. Recall that windows can also be closed with the **close** command from the browsing menu, and can later be restored with the global command **restore**. Note also that the command **resize** may affect the number of items displayed in the window.

Figure 3: Starting a ViewFinder session

Figure 4: Opening new windows and replacing windows with other windows

Figure 5: Searching a value

Figure 6: Ordering by the total number of the occurrence of a property

Figure 7: Computing an aggregate

Figure 8: Selecting by enumeration

Figure 9: Selecting by condition

Figure 10: Augmenting a frame with distant properties

Figure 11: Setting up a synchronized tree of windows

Figure 12: The same tree of windows after the root had been scrolled

5 Interfacing ViewFinder with Different Data Models

Because its internal model has low-level structures, ViewFinder can be interfaced easily to different data models, including various object-oriented models, various entity-relationship models, and various extensions of the relational model that feature a generalization hierarchy and nested attributes. In each model, the ViewFinder network (i.e., the triplet facts) must be extracted from the given database. The present implementation works only with databases that actually conform to the internal model of ViewFinder (i.e., the present system must be given a file of triplet facts).

Note that it should not be necessary to *store* the semantic networks extracted from databases. Thus, interfacing does not require converting entire databases to ViewFinder's internal format. As users browse in a database, the section of the network in the "neighborhood" being browsed is extracted from the actual database with a relatively small set of queries. A similar approach was taken by the relational browser BAROQUE [27], where "real-time" response was achieved with the aid of various indices. This process provides many opportunities for optimization; for example, how to anticipate future browsing directions, and pre-construct sections of the network, while the user is busy observing the present data; or how to determine which sections of the network recently constructed should be "cached", in case they will be needed in the near future. This topic is currently being studied.

This section focuses on the interface between ViewFinder and external databases. For reasons of space, the discussion must be limited to a single data model, and we have chosen the object-oriented data model. To keep the discussion independent of specific object-oriented database systems, it seems appropriate that we consider the ODMG-93 standard [8], which is being developed by the members of the Object Database Management Group.[14] Again, for reasons of space, we shall focus only on the main concepts of the ODMG Object Model.

5.1 The ODMG Model

An ODMG database is a collection of persistent denotable *objects*, categorized by *types*. Denotable objects are either mutable (*objects*) or immutable (*literals*). All denotable objects have identity: the identity of a literal is typically the bit pattern that encodes its value; the identity of an object is termed *object identifier* (OID). Objects may have names; a name must refer to a single object, but an object may have several names.

Objects of a given type must have common *characteristics*: a common behavior and a common range of states. *Behavior* is defined by a set of *operations* that may be executed on objects of the given type. The *state* of objects is defined by the *values* they carry for a set of *properties*. Properties are further classified into *attributes* and *binary relationships*, where the former take literals as values, and the latter define *traversal paths* between types.

[14]The ODMG voting member companies and most of the reviewer companies are committed to support this standard, which is therefore expected to become the de facto standard for the object-oriented industry.

A type has one *interface* and one or more *implementations*. The interface defines the external interface supported by instances of the type (i.e., properties and operations). An implementation defines the actual *data structures* for representing the instances of the type, and the *methods* that operate on these data structures.

A type is itself an object, and, as such, may have properties, including *supertypes*, *extent* (i.e., the instances of this type), and *keys*. A subtype (1) inherits all the attributes, relationships and operations of its supertypes, (2) may add additional properties and operations, and (3) may refine inherited properties and operations. Multiple inheritance is allowed. An instance of a subtype is also an instance of each of its supertypes.

These concepts are illustrated by this following example that shows an interface definition for a type EMPLOYEE. The interface is defined in ODL, the object definition language of the ODMG Object Model. The example includes a structured literal (the attribute ADDRESS) and a collection-valued property (the relationship HAS_SECRETARIES).

```
interface EMPLOYEE : PERSON        //EMPLOYEE has supertype PERSON

// type properties:
(   extent EMPLOYEES
    keys COMPANY_ID
)

// instance properties:
{   attribute String COMPANY_ID;
    attribute Integer SALARY;
    attribute Structure ADDRESS { Integer NUMBER, String STREET, String CITY}
                        ADDRESS;
    relationship Set <EMPLOYEE> HAS_SECRETARIES;
}
```

5.2 The ViewFinder Representation of an ODMG Database

ViewFinder represents an ODMG database with a set of facts that can be browsed to explore the generalization hierarchy and to view the properties of type and instance ODMG objects. There is no support for object behavior, since the function of a browser is to search the contents of databases.[15] The ViewFinder representation of various ODMG structures will be denoted with the mapping μ.

Object Names

A ViewFinder object must have exactly one name (indeed, objects *are* names), whereas an

[15]Nevertheless, it is possible to extend the external model with a *method frame* to display information about the methods associated with objects.

ODMG object may be referred to by several names. A named ODMG object is represented in ViewFinder by one of its ODMG names; an unnamed object is assigned a name.

Generalization Facts

There is a simple one-to-one correspondence between the types of an ODMG database and the classes of its ViewFinder representation. The ODMG distinction between literal types and nonliteral types induces a partition of the ViewFinder classes into *literal classes* and *nonliteral classes*.

For every literal type t (either built-in or defined by an interval, an enumeration or a structure), $\mu(t)$, its ViewFinder counterpart, is a subclass of one of the ViewFinder type classes; e.g., INTEGER, STRING, DATE, BOOLEAN, and REAL. Specifically,

$\mu(\text{INTEGER}) \prec \text{INTEGER}$,
$\mu(\text{FLOAT}) \prec \text{REAL}$,
$\mu(\text{BOOLEAN}) \prec \text{BOOLEAN}$,
$\mu(\text{DATE}) \prec \text{DATE}$,
$\mu(t) \prec \text{STRING}$, for $t \in \{\text{CHARACTER}, \text{CHARACTER\_STRING}, \text{BIT\_STRING}, \text{TIME}, \text{TIMESTAMP}\}$,
$\mu(t) \prec \mu(T)$, if t is an interval of values of type T,
$\mu(t) \prec \text{STRING}$, if t is an enumeration or a structure.[16]

For every nonliteral type t, $\mu(t)$ is a (nonliteral) class with the same name as t. The supertypes of t induce generalization facts with $\mu(t)$ as source. If the only supertype of t is the root of the ODMG type hierarchy, then ViewFinder includes the generalization fact $(\mu(t), \prec, \text{STRING})$; otherwise, if t has the supertypes $T_1, T_2, \ldots, T_n$, then ViewFinder includes the generalization facts $(\mu(t), \prec, \mu(T_i)),\ i = 1, \ldots, n$.

For example, the ODMG type EMPLOYEE induces these generalization facts:

$$(\mu(\text{EMPLOYEE}), \prec, \mu(\text{PERSON})),$$
$$(\mu(\text{ADDRESS}), \prec, \text{STRING}).$$

Intensional Facts

The only literal classes that can appear as source of intensional facts are classes representing structures. The structure $t = \langle slot_1, slot_2, \ldots, slot_n \rangle$, where $slot_i$ is defined as $\langle slot\_name_i : [Collection\_specifier]\ \ T_i \rangle$,[17] is represented in ViewFinder with the intensional and mandatory facts $(\mu(t), slot\_name_i, \mu(T_i)),\ i = 1, \ldots, n$ (if the slot contains a collection specifier, the intensional fact is multi-valued).

For example, the structure type ADDRESS induces these intensional facts:

[17]The collection specifier is optional and is one of {Set, Bag, List, Array}, and T_i is a literal type.

$(\mu(\text{ADDRESS}), \text{NUMBER}, \mu(\text{INTEGER}))$, mandatory and single-valued,
$(\mu(\text{ADDRESS}), \text{STREET}, \mu(\text{STRING}))$, mandatory and single-valued,
$(\mu(\text{ADDRESS}), \text{CITY}, \mu(\text{STRING}))$, mandatory and single-valued.

The interface body of a nonliteral type t determines the set of intensional facts having $\mu(t)$ as source. There is a one-to-one correspondence between the characteristics (attributes and relationships) of an ODMG nonliteral type t and the ViewFinder properties of $\mu(t)$. A characteristic signature in the interface body of t, defined as

$$characteristic\_specifier\ \ [Collection\_specifier]\ \ target\_type\ \ name\ ^{18}$$

is represented in ViewFinder with the intensional fact $(\mu(t), name, \mu(target\_type))$. The property of $\mu(t)$ is mandatory if the corresponding characteristic is a key (or belongs to a compound key) and multi-valued if the corresponding characteristic signature contains a collection specifier.

The distinction between attributes and relationships is somewhat blurred, and is retained only by the nature of the target class $\mu(target\_type)$, which is a literal class for properties representing attributes and a nonliteral class for properties representing relationships.

The characteristics of EMPLOYEE induce these intensional facts:

$(\mu(\text{EMPLOYEE}), \text{COMPANY\_ID}, \mu(\text{STRING}))$, mandatory and single-valued,
$(\mu(\text{EMPLOYEE}), \text{SALARY}, \mu(\text{INTEGER}))$, single-valued,
$(\mu(\text{EMPLOYEE}), \text{ADDRESS}, \mu(\text{ADDRESS}))$, single-valued,
$(\mu(\text{EMPLOYEE}), \text{HAS\_SECRETARIES}, \mu(\text{EMPLOYEE}))$, multi-valued.

Membership Facts

The ViewFinder representation $\mu(t)$ of a literal type t is implicitly instantiated and its member frame cannot be browsed, with the exception of enumeration types whose instances are explicitly declared. There is a one-to-one correspondence between the values of an enumeration t and the members of $\mu(t)$: the enumeration $t = \{v_1, v_2, \ldots, v_n\}$ is represented in ViewFinder with the membership facts $(\mu(v_i), \in, \mu(t))$, $i = 1, \ldots, n$.

For a nonliteral type t the membership facts with $\mu(t)$ as target are determined by the instance objects belonging to the extent of t. There is a one-to-one correspondence between the instances of t and the members of $\mu(t)$: $extent(t) = \{o_1, o_2, \ldots, o_n\}$ is represented in ViewFinder with the membership facts $(\mu(o_i), \in, \mu(t))$, $i = 1, \ldots, n$.

Extensional Facts

The state of any ODMG instance object consists of instances of the characteristics (attributes and relationships) defined by its type. The object characteristics define abstract states: they appear within the interface definition of an object type rather than in the implementation.

[18]The characteristic_specifier is **attribute** or **relationship.**

An object state can be treated as a list of pairs $(characteristic\_name, value)$, where the value may be an individual object or a collection of objects, depending on whether the characteristic is single-valued or collection-valued. The correspondence between the state of o and the extensional facts having $\mu(o)$ as source is defined in the following way:

- Each single-valued characteristic instance of an object o is represented in ViewFinder with an extensional fact $(\mu(o), characteristic\_name, \mu(value))$.
- Each collection-valued characteristic instance of o is represented in ViewFinder with the extensional facts $(\mu(o), characteristic\_name, \mu(v_i))$ for $v_i \in value$.

6 Conclusion

ViewFinder is a graphical tool for browsing and querying databases. Its design emphasizes simplicity of operation, advanced functionalities, and generality (with respect to the databases with which it can be used). ViewFinder was fully implemented as a prototype. It is written in the C language, and it runs on a Sun Workstation in the Unix operating system environment. The external model uses the SunView window management system. Presently, the database is in the format of the internal model.

Work on ViewFinder is still continuing. At the user's end, we are interested in improving the current implementation of the external model, to achieve a more attractive graphical user interface. At the database end, we are interested in interfacing ViewFinder to external database systems and we are investigating implementation techniques that assure good performance. In this section we describe two additional thrusts of research: extending ViewFinder with features for assembling browsing results, and extending ViewFinder to allow users to modify the database.

6.1 Displaying Views of Virtual Objects

Presently, ViewFinder allows users to explore a database by repeatedly displaying views of existing database objects, usually by pointing at their references in views presently displayed. Work is underway on extensions to ViewFinder that will allow users to construct views that do not correspond to existing database objects. Specifically, users will be provided with a new set of commands for creating *virtual* objects from actual database objects, and displaying their views.

To our knowledge, existing database browsing tools do not provide structures and operations for constructing *results* of browsing sessions. Consequently, users who encounter many items of interest in a search process, may simply have to copy them onto a note pad. In analogy with relational databases, where formal queries operate on actual database relations to create relations which are answers to these queries, the new commands will operate on database views (objects) to create views (objects) that are results of browsing sessions.

Virtual objects are always classes. When created (in the beginning of a browsing session) virtual classes are most general and have no members. During the session they are manipulated to include objects of interest, and their position on the generalization hierarchy changes correspondingly to reflect their new "contents". At the end of the session they contain the "output" of the session. The dynamic changes in the "class" of a virtual class agrees with the uncertainty that characterizes browsing.

As an informal example, consider a database that includes the following generalization hierarchy: the classes MALE, FEMALE and EMPLOYEE are specializations of PERSON, and the classes MANAGER and TECHNICAL are specializations of EMPLOYEE. Assume that we are browsing in this database to determine who should get a salary raise. Initially, we create a new virtual class called RAISE. Next, we insert into RAISE the classes MANAGER and TECHNICAL in their entirety. Then, we retain in RAISE only the members of the class FEMALE, and we restrict RAISE to include only employees in the manufacturing department. Finally, we insert TOM into RAISE and we delete BETTY from RAISE. Altogether, the members of the virtual class RAISE are the female employees in the manufacturing department that are either managers or members of the technical staff, but excluding Betty and including Tom.

It is possible to represent the definition of a virtual class in the form of a new kind of facts, to be called *definition facts*, which would be displayed in a new (fifth) frame of the virtual class.[19] Browsing can then be extended to allow users to browse also in definitions. Since definitions are often used to formalize higher level concepts, such definitions, like inference rules, may be regarded as a form of *knowledge*. Altogether, ViewFinder would be integrating access to *data* (i.e., token objects), *schema* (i.e., class objects and their hierarchies), and *knowledge* (i.e., definitions of virtual classes).

6.2 Extending ViewFinder to Handle Database Modifications

While presently ViewFinder is a tool for database retrieval (e.g., browsing, querying), it may be extended to allow also database modification. Modifications may include the *creation* of new objects (either classes or tokens), the *deletion* of existing objects, and the *alteration* of existing objects.

Modifications will be defined at the level of the external model. That is, users will alter the information displayed in views, or will use menu commands to create and destroy objects. These changes will be checked for consistency with the rules governing the internal model, and will then be translated into modifications of the underlying semantic network (i.e., insertion, deletion or alteration of triplet facts).

[19]Actual classes would have empty definition frames.

When ViewFinder is interfaced with other data models, it would be necessary to propagate the modifications to the actual database. Issues of update propagation are currently being studied.

Acknowledgement The implementation of ViewFinder was done by Fabrizio Prosperi Porta (preliminary version) and Walter Tross (current version). The authors wish to thank them for their valuable suggestions and comments.

References

[1] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[2] R. Agrawal, N. H. Gehani, and J. Srinivasan. OdeView: The graphical interface to Ode. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Atlantic City, New Jersey, May 23–25), pages 34–43, 1990.

[3] Ashton-Tate, Culver City, California. *DBASE-III Reference Manual*, 1984.

[4] J.L. Bell. Reuse and browsing: Survey of program developers. In D. Tsichritzis, editor, *Object Frameworks*, pages 197–220. Centre Universitaire d'Informatique, Université de Genève, 1992.

[5] D. Bryce and R. Hull. SNAP: A graphics-based schema manager. In *Proceedings of the IEEE Computer Society Second International Conference on Data Engineering* (Los Angeles, California, February 5–7), pages 151–164, 1986.

[6] R. G. G. Cattell. An entity-based database interface. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Santa Monica, California, May 14–16), pages 144–150, 1980.

[7] R. G. G. Cattell. Design and implementation of a relationship-entity-datum data model. Technical Report CSL-83-4, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, May 1983.

[8] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93 (Release 1.1)*. Morgan Kaufmann, San Francisco, California, 1994.

[9] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, January 1976.

[10] A. D'Atri and L. Tarantino. From browsing to querying. *Data Engineering*, 12(2):46–53, June 1989.

[11] A. D'Atri and L. Tarantino. A browsing theory and its application to database navigation. In J. Paradaens and L. Tenembaum, editors, *Advances in Database Systems, Implementations and Applications*, CISM Courses and Lectures No. 347, pages 161–180. Springer-Verlag, 1994.

[12] A. Dix and A. Patrick. Query by browsing. In Pete Sawyer, editor, *Interfaces to Databases Systems, Lancaster 1994*, Workshops in Computing, pages 236–248. Springer-Verlag, 1994.

[13] D. Fogg. Lessons from a 'living in a database' graphical query interface. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Boston, Massachusetts, June 18–21), pages 100–106, 1984.

[14] A. Goldberg and D. Robson. A metaphor for user interface design. In *Proceedings of the 13th Hawaii International Conference on System Science* (Honolulu, Hawaii, January 3–4), pages 148–157, 1980.

[15] K. J. Goldman, S. A. Goldman, P. C. Kanellakis, and S. B. Zdonik. Isis: Interface for a semantic information system. In *Proceedings of the ACM-SIGMOD Conference on the Management of Data*, pages 328–342, 1985.

[16] I. Goldstein and D. Bobrow. Browsing in a programming environment. In *Proceedings of the 14th Hawaii International Conference on System Science* (Honolulu, Hawaii, January 8–9), 1981.

[17] M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[18] R. Helm and Y. Maarek. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In *Proceedings of OOPSLA '91, Phoenix*, pages 47–61, 1991.

[19] C. Herot. Spatial management of data. *ACM Transactions on Database Systems*, 5(4):493–513, December 1980.

[20] R. Johnson, M. Goldner, M. Lee, K. McKay R. Schectman, and J. Woodruff. USD — a database management system for scientific research. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (San Diego, California, June 2–5), 1992.

[21] R. King and D. McLeod. An approach to database design and evolution. In M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modeling*, pages 313–327, 1984.

[22] R. King and S. Melville. SKI: a semantic knowledgeable interface. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, August 27–31), pages 30–33, 1984.

[23] T. Learmont and R. G. G. Cattell. An object-oriented interface to a relational database. Technical report, Information Management Group, Sun Microsystems, 1987.

[24] G. A. Miller. Dictionaries of the mind. In *Proceedings of 23rd Annual Meeting of the Association for Computational Linguistics* (Chicago, Illinois, July 8–12), pages 305–314, 1985.

[25] A. Motro. Browsing in a loosely structured database. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Boston, Massachusetts, June 18–21), pages 197–207, 1984.

[26] A. Motro. Assuring retrievability from unstructured databases through contexts. In *Proceedings of the IEEE Computer Society Second International Conference on Data Engineering* (Los Angeles, California, February 5–7), pages 426–433, 1986.

[27] A. Motro. BAROQUE: A browser for relational databases. *ACM Transactions on Office Information Systems*, 4(2):164–181, April 1986.

[28] A. Motro, A. D'Atri, and L. Tarantino. The design of KIVIEW: An object-oriented browser. In *Proceedings of the Second International Conference on Expert Database Systems* (Tysons Corner, Virginia, April 25–27), pages 17–31, 1988.

[29] E. Neuhold and M. Stonebraker (editors). Future directions in dbms research. Technical Report TR-88-001, International Computer Science Institute, May 1988.

[30] J. Nielsen. The art of navigating through hypertext. *Communication of the ACM*, 33(3):296–310, 1990.

[31] X. Pintado and D. Tsichritzis. An affinity browser. In D. Tsichritzis, editor, *Active Object Environments*, pages 51–60. Centre Universitaire d'Informatique, Université de Genève, 1988.

[32] T. R. Rogers and R. G. G. Cattell. Object-oriented database user interfaces. Technical report, Information Management Group, Sun Microsystems, 1987.

[33] A. Silberschatz, M. Stonebraker, and J. Ullman (Editors). Database systems: Achievements and opportunities. *Communications of the ACM*, 34(10):110–120, October 1991.

[34] M. Stonebraker and J. Kalash. TIMBER: A sophisticated database browser. In *Proceedings of the Eighth International Conference on Very Large Data Bases* (Mexico City, Mexico, September 8–10), pages 1–10, 1982.

[35] P.D. Stotts and R. Furuta. Petri-net-based hypertext: Document structure with browsing semantics. *ACM Transactions on Information Systems*, 7(1):3–29, 1989.

[36] Sun Microsystems, Mountain View, California. *SunView Programmer's Guide*, Revision A (Part Number 800-1345-10), 1986.

[37] S. A. Weyer and A. H. Borning. A prototype electronic encyclopedia. *ACM Transactions on Office Information Systems*, 3(1):63–88, January 1985.

[38] H. K. T. Wong and I. Kuo. GUIDE: A graphical user interface for database exploration. In *Proceedings of the Eighth International Conference on Very Large Data Bases* (Mexico City, Mexico, September 8–10), pages 22–32, 1982.