The Design of KIVIEW: An Object-Oriented Browser

Amihai Motro
Computer Science Department, University of Southern California, University Park, Los Angeles, CA 90089-0782

Alessandro D'Atri
Dipartimento di Ingegneria Elettrica, Universita dell'Aquila, Monteluco Roio, 67100 L'Aquila, and Dipartimento di Informatica e Sistemistica Universita di Roma, Via Buonarroti 12, 00185 Rome

Laura Tarantino
Dipartimento di Informatica e Sistemistica Universita di Roma, Via Buonarroti 12, 00185 Rome

Abstract

KIVIEW is an object-oriented browsing interface to databases. The underlying *internal* model is a semantic network, defined via a collection of triplet facts which are governed by a small set of integrity requirements. KIVIEW's only *external* structure is a display, called *view*, that presents all the information that is pertinent to a given object. KIVIEW features two kinds of activity, called *navigation* and *manipulation*. Navigation is an exploration activity wherein the user repeatedly displays new views of objects that are referenced in current views. To speed up repetitive navigational tasks, 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, the contents of the other views change automatically. Manipulation is a processing activity wherein the user operates on existing views to create new virtual views. Manipulation is interleaved with navigation to assemble information located while browsing into *results* of browsing sessions. Using convenient graphic tools, these two activities are integrated into a single tool, which is flexible and effective, yet simple to use.

This work was supported in part by the European Communities under ESPRIT project 1117; the work of Motro was also supported by NSF Grant No. IRI-8609912 and by an Amoco Foundation Engineering Faculty Grant; the work of Tarantino was also supported by ENIDATA.

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. While browsing, users gain insight into the contents and organization of the searched environment. Eventually, the search either terminates successfully or is abandoned.

Often, the conceptual model is a *network* of some kind, and browsing is done by *navigation*: the user begins at an arbitrary point on 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 example of this approach is the interface designed and implemented by Cattell [2]. The interface is to an entity-relationship database [3], 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*. The principles of this interface were carried over to Cypress, a database management system developed by Cattell at Xerox [1]. Cypress starts with a data model based largely on constructs derived from well-known data models, complementing it with an extensive array of features.

Browsing is offered as the principal retrieval method for loosely-structured databases [8]. Such databases are heaps of facts that do not adhere to any conceptual design. As facts are named binary relationships between data values, the data may be regarded as a network of values. Two styles of browsing, called *navigation* and *probing*, are defined. Both are derived from a query language based on predicate logic.

BAROQUE [7] is a browsing interface to relational databases. BAROQUE establishes a view of the relational database that resembles a semantic network, and provides several intuitive commands for scanning it. The network integrates both schema and data, and supports access by value.

Browsers have been developed for other relational systems, for example, SDMS [5], INGRES [11] and DBASE-III [4]. These are actually 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 what other relations this value may appear (quite difficult), then formulate a formal query, and resume browsing in the new relation.

While, in general, most of these tools define search processes that are indeed simple to perform, they usually suffer from two major drawbacks. First, because of the requirement for simplicity, browsing is often performed with low-level commands. Consequently, browsing sessions tend to be quite inefficient, and may become tedious after a while. More importantly, to our knowledge, none of these tools 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 this paper we describe a browsing interface, called KIVIEW, that addresses these problems. The underlying internal model 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 (user) model.

KIVIEW browsing sessions interleave two activities, called *navigation* and *manipulation*. Navigation is an exploration activity wherein the user repeatedly displays new views of objects that are referenced in current views. 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.

Manipulation is a processing activity wherein the user operates on existing views to create new virtual views. Manipulation is interleaved with navigation to assemble information located while browsing into *results* of browsing sessions. In analogy with relational databases, where formal queries operate on database relations to create relations which are answers to these queries, manipulation commands operate on database views to create views which are results of browsing sessions. When first created, virtual views are as general as possible; their "type" then changes dynamically with each manipulation to conform to their new "contents". This approach agrees with the uncertainty that characterizes browsing.

Using convenient graphic tools, these two activities are integrated into a single tool, which is flexible and effective, yet simple to use. KIVIEW was designed to interface with the KIWI system, a software package that aims at providing friendly and knowledgeable access to multiple databases, using techniques from knowledge engineering, logic programming and database management [10].

The remainder of this paper is organized as follows. Sections 2 and 3 define the internal and external models. Sections 4 and 5 are devoted to navigation and manipulation. Section 6 outlines the design of an interface that implements these operations, and Section 7 concludes with a brief summary and discussion of additional research directions.

2 The Internal Model

In this section we define the underlying *internal model* of KIVIEW. This model is not apparent to the users of KIVIEW, but is necessary for defining the structures and operations of the *external model*. We note that this internal model is not supported directly by the KIWI system. Rather, it is an intermediate model between KIVIEW's external model and KIWI's model.

2.1 Objects and Relationships

As mentioned earlier, 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 [8,6]. 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 left and right objects of a fact will be called, respectively, the *source* and the *target* of the fact.

We distinguish between two kinds of objects and four kinds of facts, and we define eight different requirements that must be satisfied by any collection of facts. They are described below.

2.2 Class and Token Objects

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 the following definitions the symbols a, b, c denote classes, x, y, z denote tokens, m, n denote objects (either tokens or classes), and r, s denote relationships.

2.3 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 *specification* 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 the following three conditions:

- Transitivity of generalizations:
  If $< a, \prec, b >$ and $< b, \prec, c >$ are generalization facts, then $< a, \prec, c >$ is also a generalization fact.
- Irreflexivity of generalizations:
  If $< a, \prec, b >$ is a generalization fact then $a \neq b$.
- 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 a 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. 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. Together, transitivity and irreflexivity guarantee *acyclicity*, so that the generalization relationship imposes a *hierarchy* on the classes of the database.

We assume the existence of a class which is a generalization of every other database class. The name of this object is the name assigned to the database. Here, we shall sometimes refer to this object as *database*. Finally, we require that every token is a member of at least one class:

- Required participation of classes:
  There exists a class *database*, and for every other class a there exists a generalization fact $< a, \prec, database >$.
- Required participation of tokens:
  For every token x there exists a class a and a membership fact $< x, \in, a >$.

2.4 Specific and Generic Facts

A *specific* fact associates two tokens. Examples of specific facts are <JOHN,AGE,32> and <BETTY,PLAYS,FLUTE>.

A *generic* fact associates a source class with a target class or with a target token. Some generic facts are *mandatory*: there *must* be a specific fact that associates *each* of the members of the source class with a member of the target class or with the target token.

Examples of generic facts are <PERSON,AGE,YEARS>, <KNIGHT, OBEYS,ARTHUR> and <PERSON,PLAYS,INSTRUMENT>. The first two facts are mandatory. The first and last facts have target classes, while the middle fact has a target token. The facts <JOHN,AGE,32>, <RICHARD,OBEYS,ARTHUR> and <BETTY,PLAYS,FLUTE> are examples of specific facts that are associated, respectively, with these generic facts.

We require that specific and generic facts satisfy the following four conditions:

- Instantiation of mandatory facts:
  If $< a, r, m >$ is a mandatory fact, then either

  1. m is a class: for every membership fact $< x, \in, a >$ there exist a membership fact $< y, \in, m >$ and a specific fact $< x, r, y >$.

2. m is a token: for every membership fact $< x, \in, a >$ there exists a specific fact $< x, r, m >$.

- Support of specific facts by generic facts:
  If $< x, r, y >$ is a specific fact, then either
  1. There exist membership facts $< x, \in, a >$ and $< y, \in, b >$ and a generic fact $< a, r, b >$.
  2. There exist a membership fact $< x, \in, a >$ and a generic fact $< a, r, y >$.
- Propagation of generic facts over generalizations:
  If $< a, r, m >$ is a generic fact and $< a, \prec, c >$ is a generalization fact, then $< c, r, m >$ is a generic fact.
- Propagation of mandatory facts over generalizations:
  If $< a, r, m >$ is a mandatory fact and $< b, \prec, a >$ is a generalization fact, then $< b, r, m >$ is also a mandatory fact.

The first requirement defines mandatory facts. The second requirement prohibits database facts that are not supported by generic facts and may be regarded as enforcing "strong typing" on facts. The last two requirements provide additional semantics of the generalization relationship. Generic facts are also generic for all superclasses (but may be unapplicable to subclasses). Mandatory facts are also mandatory for all subclasses (but may not be mandatory for superclasses). For example, consider the generalization hierarchy: TEMPORARY≺EMPLOYEE≺PERSON. If all employees work for departments, then all temporaries work for departments, but only some persons work for departments, while some do not. And if some employees have offices, then some persons have offices, but, possibly, temporaries never have offices.

2.5 Immediate and Distant Objects and Properties

Four of the requirements defined above (the transitivity of generalizations, the inheritance of membership over generalizations, the propagation of generic facts over generalizations, and the propagation of mandatory facts over generalizations) can be described 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 $< c, \prec, a >$. A token x is an *immediate instance* of a class a, if $< x, \in, a >$ there is no class b such that $< b, \prec, a >$ and $< x, \in, b >$.

Given a fact $< m, r, n >$, which is either specific or generic, the pair $< r, n >$ will be referred to as a *property* of m. A property $< r, n >$ is an *immediate generic property* of a class a, if $< a, r, n >$ is a generic fact and there is no class b such that $< b, \prec, a >$ and $< b, r, n >$ is also a generic fact. A property $< r, n >$ is an *immediate mandatory property* of a class a, if $< a, r, n >$ is a mandatory fact and there is no class b such that $< a, \prec, b >$ and $< b, r, n >$ is also a mandatory fact.

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

2.6 Model Summary

A *database* is a collection of generalization facts, membership facts, generic facts (some of which are mandatory) and specific facts, that satisfy these eight requirements:

1. Transitivity of generalizations.
2. Irreflexivity of generalizations.
3. Inheritance of membership over generalizations.
4. Instantiation of mandatory facts.
5. Support of specific facts by generic facts.
6. Propagation of generic facts over generalizations.
7. Propagation of mandatory facts over generalizations.
8. Required participation of tokens and classes.

3 The External Model

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

3.1 Display Structures: Views, Frames and Windows

For every database object, we define a structure called *view* that includes all the facts in which this object participates. The facts are sorted into several disjoint subsets 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. *superclassses*: the classes that are immediate generalizations of a.
3. *subclasses*: the classes that are immediate specifications of a.
4. *properties*: the immediate generic properties of a (properties that are mandatory are tagged).

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 subclass; and the superclasses are those shown in the *superclasses* frame, as well as their superclasses. Similarly, the generic properties of the class in view are those shown in the *properties* frame, as well as the generic properties of any subclass (the mandatory properties are those tagged in the *properties* frame, as well as mandatory properties of any superclass).

For example, a class EMPLOYEE may have the following view (mandatory properties are suffixed with "!"):

1. *members*: JOHN, TOM, MARY, BETTY, FRANK.
2. *superclasses*: PERSON.
3. *subclasses*: MANAGER, TECHNICAL, TEMPORARY.
4. *properties*: <WORKS-FOR,DEPARTMENT>!, <POSITION,TITLE>!, <OFFICE,ROOM>, <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 specific properties of x.

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

1. *classes*: EMPLOYEE, FATHER.
2. *properties*: <WORKS-FOR,MANUFACTURING>, <POSITION,SUPERVISOR>, <OFFICE,MB475>, <AGE,32>, <FATHER-OF,BILL>, <FATHER-OF,JULIE>.

The contents of each frame are displayed in one or more independent *windows*. Each window displays an *interval* of objects or properties, according to a particular *ordering*. Views, frames and windows will be referred to collectively as *displays*.

3.2 Display Operations: Open, Close, Order, Scroll and Activate

To access the information in the view of an object it must be opened. This is done with the operation *open*. For example, *open*(EMPLOYEE) opens the view of the object EMPLOYEE and *open*(JOHN) opens the view of the object JOHN.

Each view is composed of frames: four in case of a class object, or two in case of a token object. To access the information in a frame it too must be opened with the operation *open*. For example, if EMPLOYEE is an open view, then *open*(EMPLOYEE.*members*) opens its *members* frame.

Each frame incorporates several windows in which objects or properties are displayed. Windows are created with the operation *open*. For example, if *members* is an open frame in the view EMPLOYEE, then *open*(EMPLOYEE.*members*.1) opens the window numbered 1 in the *members* frame.

The operation *close* is the inverse of *open*.

To control the order in which the objects or properties appear in a window, the operation *order* is used. For example, if ALPHA is an ordering, then *order*(PERSON.*members*.1, ALPHA) imposes this ordering on the objects of the *members* frame of the class PERSON, as they are displayed in window 1.

Since only a limited number of objects or properties of a frame are displayed in a window at any time, the operation *scroll* is used to "slide" the window over the frame, and thus control its visible part.

At each time at most one view is active. Similarly, in the active view at most one frame is active, and in the active frame at most one window is active. Activation is done with the operation *activate*. For example, *activate*(EMPLOYEE) activates the view EMPLOYEE, *activate*(EMPLOYEE.*members*) activates the *members* frame of the view EMPLOYEE, and *activate*(EMPLOYEE.*members*.1) activates window 1 in the *members* frame of the view EMPLOYEE. When a display (i.e., view, frame or window) is activated, the previously active display of that kind is automatically deactivated. When a display is activated, all its containing displays are automatically activated. Thus, after *activate*(EMPLOYEE.*members*.1) the view EMPLOYEE becomes the active view, its *members* frame becomes the active frame, and its window 1 window becomes the active window. When a new display is opened it is activated automatically. It is only possible to operate on active displays. For example, only an active window may be scrolled.

4 Navigation

Navigation is an exploration activity, wherein the user repeatedly displays new views of database objects. First, we describe the elementary navigation operations, then the synchronization feature.

4.1 Elementary Navigation

When KIVIEW is invoked, the view of the object *database* is opened by default. The *subclasses* frame of this view, which is also opened by default, resembles a "directory" of this database. For example, in database PERSONNEL, this frame may show the subclasses EMPLOYEE, DEPARTMENT, etc.

From this point on, the user applies the five display operations described in the previous section to open, activate and close displays (views, frames or windows), to order windows, and to scroll their contents[1].

In effect, this process corresponds to *navigation* in the underlying semantic network. When a node is visited, its immediate neighborhood is displayed in a view. Opening a new view of an object referenced in the active view corresponds to crossing an edge to visit an adjacent node.

4.2 Synchronization

In elementary navigation, the different views, once opened, are independent: the information displayed in any view may be modified without any effect on the information displayed in the other views. *Synchronization* allows users to set up several views, connected in a tree-like structure, so that when the information displayed in the root view is modified (e.g., by scrolling its active window) the contents of the other views change automatically.

[1]Other operations, to control the layout of displays, will be discussed in Section 6.

Let V be an open view and let p be an object that appears in another open view. The operation $sync(p, V)$ establishes a synchronization link between p and V (more precisely, between the *structures* that currently display p and V). When p is overwritten in its window by another object p', the view of p' will be displayed in V.

Such synchronization links may be repeated to form a *tree* of views. When any view in this tree is activated, and the information in one of its windows is scrolled, changes will be made to all the *descendent* views that are synchronized with this window.

Synchronization is terminated with the operation $break(V)$, where V is an open view. This operation breaks the synchronization between V and its ancestors. Synchronization is broken automatically, when the window in which synchronization was established (or any of its containing displays) is closed, or when the view at the other end of the link is closed.

The precise effect of synchronization depends on the window in which p is displayed: we distinguish between windows in frames of *objects* (i.e., *members* or *classes*) and windows in frames of *properties*. If the window displays objects, the synchronization link is purely *positional*: if p occupied the i'th position in the window when synchronization was established, then V will always display the view of the object that occupies the i'th position. If the window displays properties, the synchronization link is also *semantic*: if the property that occupied the i'th position when synchronization was established was $< r, p >$, then V will display the view of the object of the property that occupies the i'th position only if the relationship of this property is r. Otherwise V will display a special view called NULL.

Note that the are two methods by which the contents of a window may change. The window may be *scrolled* by the user, or it may be *refreshed* with the frame of another object due to a synchronization link at a higher level. In the latter case we must determine the contents of the refreshed window. If the window displays objects, the refreshed window will display the initial interval of the frame (according to the prevailing ordering). The object occupying the i'th position will then be displayed in V. If the new frame is empty, then V will display the view NULL. If the window displays properties, the refreshed window will be scrolled until the first property with relationship r occupies the i'th position. Its object will then be displayed in V. If the new frame does not have a property with relationship r, then V will display the view NULL.

We demonstrate the use of synchronization with a description of a browsing session that involves three views.

First the user opens the view EMPLOYEE and its *subclasses* frame, and scrolls a window in this frame until the object MANAGER appears. Next, the user opens the view MANAGER and its *members* frame, and scrolls a window in this frame until the object BILL appears. Finally, the user opens the view BILL and its *properties* frame, and scrolls a window in this frame until the property <OFFICE,AV678> appears.

The user now synchronizes the third view with the object BILL in the second view, and the second view with the object MANAGER in the first view. Figure 1 illustrates the resulting situation.

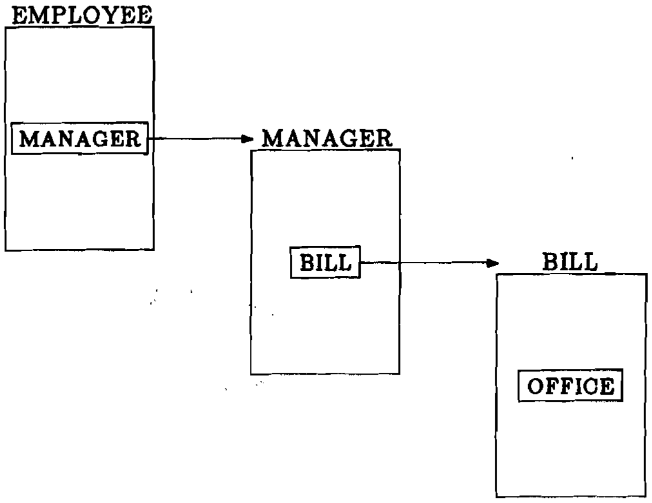

Figure 1: Three synchronized views

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

When finished, the user returns to the first view and scrolls the list of subclasses in the first view, replacing MANAGER by TECHNICAL. The second view 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 view, to browse in the list of technical employees.

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

5 Manipulation

While navigation allows users to examine views of existing database objects, manipulation allows users to display views that do not correspond to existing database objects. Formally, the manipulation operators modify the underlying database by creating new *virtual classes* whose views can then be displayed. However, to users, a manipulation process seems as operating on the *views* themselves. With these operators, users can maintain views that contain the data in which they are interested, while they navigate in the database. In several respects, manipulation may be regarded as a form of querying.

There are six manipulation operators, called *create*, *include*, *retain*, *restrict*, *insert*, and *delete*. In many ways these operators are similar to the schema-modifying operators defined in [9]. Note that all modifications to the underlying database (i.e., additions of new classes and relationships) are temporary, for the duration of the browsing session.

5.1 Create

The operator *create* defines a new virtual class, whose name is provided by the user. The new class is as general as possible: the only object which is related to the new class is the object *database*, which is its immediate generalization. Formally, the operation *create* v, defines a new virtual class v, and inserts the new fact $< v, \prec, database >$. Hence, the view of v is:

1. *members*: none.
2. *superclasses*: *database*.
3. *subclasses*: none.
4. *properties*: none.

create is always the first operation in every manipulation process. As an example, consider a database that incorporates the following generalization hierarchy: the classes MALE, FEMALE and EMPLOYEE are specifications of PERSON, and MANAGER, TECHNICAL and TEMPORARY are specifications of EMPLOYEE. Assume that we are browsing in this database trying to determine who should get a salary raise. As a first step we create a new virtual class called RAISE:

create RAISE

5.2 Include and Retain

The operator *include* makes a virtual class the immediate generalization of another class. Formally, assume that v is a virtual class and that a is another class, and assume that a and v are not related in a generalization relationship. The operation *include* a *in* v inserts into the database the generalization fact $< a, \prec, v >$. In easy to verify that this change preserves the irreflexivity of generalizations (Rule 2), the instantiation of mandatory facts (Rule 4), the support of specific facts by generic facts (Rule 5) and the required participation of tokens and classes (Rule 8). The remaining four rules are concerned with closure properties of generalizations. These rules are preserved by generating or removing the necessary facts. For example, the members of a that are not already members of v are added to v (and to every more general class).

The effect of including a class in a newly created class, is that of duplication; i.e., the existing view is *copied* into the new view. In the previous example, assume that we decide that all managers and technical employees should get a raise. We issue:

include MANAGER *in* RAISE
include TECHNICAL *in* RAISE

The operator *retain* makes a virtual class the immediate specification of another class. Formally, assume that v is a virtual class and that a is another class, and assume that a and v are not related in a generalization relationship. The operation *retain* a *in* v inserts into the database the generalization fact $< v, \prec, a >$. Again, four rules are unaffected, and the four closure properties are preserved by generating or removing the necessary facts. For example, only the members of v that are also members of a are retained in the class v (and in every more specific class).

In the previous example, assume now that we decide to restrict the salary raise to females only. We issue:

retain FEMALE *in* RAISE

5.3 Restrict

The operator *restrict* substitutes a virtual class with a class that is more specific, by restricting one of its generic properties. There are two basic forms of restriction: by *value* of a property and by *existence* of a property. The former version reduces the virtual class to include all the members that have a *particular* target object for a given relationship; the latter reduces the virtual class to include all the members for which a given relationship is *defined*, regardless of the target value. The generic property used in the restriction need not be mandatory. Formally, assume that v is a virtual class, let $< r, c >$ be a generic property of v (where c is a class), and let x be a member of c. The operation *restrict v by r where c=x* replaces the generic property $< v, r, c >$ with the mandatory property $< v, r, x >$. Again, four rules are unaffected, and the four closure properties are preserved by generating and removing the necessary facts. For example, only the members of v that have the specific property $< r, x >$ are retained. The operation *restrict v by r* is defined similarly, except that all the members of v that have the specific property $< r, x >$ for some x which is a member of b are retained.

In the previous example, assume that we want to restrict further the salary raise to employees in the manufacturing department. We issue:

restrict RAISE *by* WORKS-FOR *where* DEPARTMENT=MANUFACTURING

5.4 Insert and Delete

The operator *insert* inserts a token object into a virtual class. Formally, assume that v is a virtual class and that x is a token object, and assume that x is not a member of v. The operation *insert x into v* inserts into the database the membership fact $< x, \in, v >$. If x does not have a specific property for a mandatory property r of v, then the specific property $< x, r, \text{NOT\_AVAILABLE} >$ is inserted[2].

In the previous example, assume that we decide that TOM (who is not currently a member of RAISE), should be included. We issue:

insert TOM into RAISE

The operator *delete* deletes a token object from a virtual class. Formally, assume that v is a virtual class and that x is a token object,

[2]Instead of inserting this specific fact, we could simply determine that r is no longer a mandatory property. However, if x is then deleted, it would not be possible to restore r to a mandatory property.

and assume that x is a member of v. The operation *delete* x *from* v deletes from the database the membership fact $< x, \in, v >$.

In the previous example, assume that we decide that BETTY (who is currently a member of RAISE), should be excluded. We issue:

delete BETTY from RAISE

Thus, the members of the virtual view RAISE are the female employees that are managers and technical employees in the manufacturing department, excluding Betty but including also Tom.

6 The Interface Design

In this section we outline a user environment that implements the structures and operations described in the previous sections. The design employs graphic tools that provide convenience and efficiency, and it is possible to perform entire browsing sessions with little or no keyboard input.

Most of the input is entered by means of a *mouse*. We shall refer to the action of moving the mouse so that its screen arrow points at the particular item and then clicking the mouse as *selection*. *Icons* are used to represent *menus*. When an icon is selected, a menu of options appears. The user then selects an item from the menu, after which the menu disappears. *Scroll bars* are vertical columns at the side of windows. By selecting the scroll bar at various locations, the user scrolls the contents of the window.

Each open view is displayed as a rectangular area, with a heading that shows the name of the object and an icon. When this icon is selected a menu of frames appears (four in the case of a class view, two in the case of a token view). By selecting an item from this menu, the user opens a frame that would be displayed in the view.

When a frame is opened, it is displayed in a rectangular area within the area of the view, with a heading that shows the name of the frame and an icon. When this icon is selected a menu of windows appears. By selecting an item from this menu, the user opens a window that would be displayed in the frame.

When a window is opened, it is displayed as a rectangular area within the area of the frame, with a heading that shows the number of the window and an icon. When this icon is selected a menu of orderings appears. By selecting an item from this menu, the user orders the contents of this window. The window displays an interval of objects or properties (i.e., relationship-object pairs).

Browsing operations are grouped into three global menus selected through icons.

The *navigation* menu includes six commands:

1. *open*: Open a new display. The user selects the display to be opened: a frame from the frame menu in the active view, a window from the window menu in the active frame, or an object name from the active window. These will open, respectively, a new frame, a new window, and a new view. The user determines the position and shape of the new display. The new display is activated.

2. *close*: Close a display. The user selects the display to be closed. An active display cannot be closed.

3. *activate*: Activate a display. The user selects the display to be activated.

4. *order*: Order the contents of a window. The user selects an ordering from the ordering menu of the window. The window must be active.

5. *sync*: Synchronize the active view with another view. The user selects an object in another view that will drive the active view.

6. *break*: Break synchronization. The synchronization between the active view and its driving view is broken.

The *manipulation* menu includes six commands:

1. *create*: Create a new virtual view. The user is prompted for a name for this view, and must then determine its position and shape. The new view is opened and activated.

2. *include*: Include another view the active view. The user selects the view to be included. The active view must be virtual.

3. *retain*: Reduce the active view by another view. The user selects the view to be retained. The active view must be virtual.

4. *restrict*: Restrict the active view by some property. The user selects the restricting property in the active view, and is then prompted for a value. If a value is entered, then restriction by value is performed; otherwise, restriction by existence is performed. The active class must be virtual.

5. *insert*: Insert a token into the active class. The user selects the token from a window from some other class. The active class must be virtual.

6. *delete*: Delete a token from the active class. The user selects the token from a window in the virtual class. The active class must be virtual.

The *control* menu includes six commands:

1. *reposition*: Move a display on the screen (within the area of the containing display). The user selects the display to be moved, and determines its new position.

2. *reshape*: Change the dimensions of a display (within the area of the containing display). The user selects the display to be reshaped, and determines its new dimensions.

3. *hide*: Hide a display behind the displays that it covers. The user selects the display to be hidden.

4. *expose*: Expose a display that is covered by other displays. The user selects the display to be exposed.

5. *input*: Prompt for keyboard input. The user is prompted for keyboard input (to be used instead of selection).

6. *exit*: Terminate KIVIEW.

Thus, browsing is performed almost entirely by selecting from menus, scrolling windows, and providing placement instructions. The only exceptions are the *create* and *restrict* commands that prompt for additional keyboard input. However, in each case where a selection is expected, the user may also execute the *input* command that will prompt the user for keyboard input. The user can then *type* the name of the selection. This permits displaying views of objects that are not referenced in current views.

7 Conclusion

We described a new browsing interface to databases called KIVIEW. KIVIEW features a single external structure called *view*, and two activities called *navigation* and *manipulation*. A view is a display that presents all the information that is pertinent to a given object. Navigation is an exploration activity wherein the user repeatedly displays new views of objects that are referenced in current views. Manipulation is a processing activity wherein the user operates on existing views to create new virtual views. By interleaving navigation and manipulation, users explore the database while accumulating results.

To improve browsing efficiency KIVIEW features a view synchronization mechanism, that speeds up many repetitive searches. While KIVIEW was designed for the KIWI system, it can interface with any database model from which a semantic network may be extracted, including the relational model (see [7]).

A prototype of KIVIEW is currently being implemented at the University of Rome, using SUN workstations and the INGRES database management system. This initial prototype follows the design outlined in Section 6, which focuses on the features that are unique to KIVIEW. In parallel with this implementation effort, we are considering enriching KIVIEW with additional features. For example, add a new browsing option, whereby the user mentions two objects and requests that the system attempt to *connect* them with a sequence of views, and allow users to display in a view *all* the objects and properties that are related to the given object (i.e., both immediate and distant).

References

[1] 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.

[2] 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, ACM, New York, New York, 1980.

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

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

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

[6] 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, IEEE Computer Society, Washington, DC, 1986.

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

[8] 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, ACM, New York, New York, 1984.

[9] A. Motro. Superviews: virtual integration of multiple databases. *IEEE Transactions on Software Engineering*, SE–13(7):785–798, July 1987.

[10] D. Sacca, D. Vermeir, A. D'Atri, A. Liso, G. S. Pedersen, J. J. Snijders, and N. Spyratos. Description of the overall architecture of the KIWI system. In *ESPRIT '85 Status Report of Continuing Work*, pages 685–700, Elsevier Science Publishers (North-Holland), 1986.

[11] 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, VLDB Endowment (available from Morgan-Kaufmann, Los Altos, California), 1982.