Designing Database Interfaces with DBface

ROGER KING
University of Colorado
and
MICHAEL NOVAK
INRIA-Rocquencourt

DBface is a toolkit for designing interfaces to object-oriented databases. It provides users with a set of tools for building custom interfaces with minimal programming. This is accomplished by combining techniques from User Interface Management Systems (UIMS) with a built-in knowledge about the specific kinds of techniques used by object-oriented databases. DBface allows users to create graphical constructs and interactive techniques by taking advantage of an object-oriented database environment and tools. Not only can database tools be used for creating an interface, but information about the interface being built is stored within a database schema and is syntactically consistent with all other schema information. Thus, an interface can deal with data and schema information, including information about another interface. This allows for easy reusability of graphical constructs such as data representations.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; E.2 [**Data**]: Data Storage Representations; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; H.2.3 [**Database Management**]: Languages; H.2.4 [**Database Management**]: Systems—*query processing*; H.2.8 [**Database Management**]: Database Applications; I.3.4 [**Computer Graphics**]: Graphics Utilities

General Terms: Design, Experimentation, Human Factors

Additional Key Words and Phrases: Graphical interfaces, object-oriented databases, user interface management systems

1. INTRODUCTION

DBface is a UIMS-type tool intended specifically for building graphics-based interfaces to object-oriented databases. It is a window-based, interactive graphical system that allows interfaces to be built with minimal program-

We would like to thank ONR for their support under contracts N00014-86-k-0054 and N0014-88-k-0559 and USWEST for their support under contract OCG0683B.
Authors' addresses: R. King, Dept. of Computer Science, University of Colorado, Boulder, CO 80309, email: roger@cs.colorado.edu; M. Novak, INRIA Rocquencourt, B. P. 105, 78153 Le Chesnay Cedex, France, email: novak@madonna.inria.fr.

ming effort. Although DBface borrows a lot from UIMS technology, it is not designed to be a UIMS with database knowledge built in, rather, it is designed to be a database tool with UIMS knowledge built in. Because DBface concentrates only on the creation of object-oriented database interfaces, certain knowledge about object-oriented databases can be taken advantage of. This includes knowledge about schemas, type-subtype hierarchies, methods, and database tools such as data definition languages (DDL). Therefore, a certain amount of knowledge about the types of interfaces being designed and the objects they manipulate is built into DBface. For example, DBface can use graphical representations of database objects that already exist in the database as defaults for a new interface that is being built. If a new representation is needed it can be built with the database tools available to DBface. New representations are stored alongside the old ones in the database. Users can work with representations in the same manner as with any other database objects. DBface takes advantage of the structure of the database and of its tools in order to make designing an interface easier and faster. Also, by making the underlying structure of the interface dependent on the database, the interface is forced to stay consistent with the database. Thus, the line between the application (the database) and its interface becomes *seamless*.

In the past, database interfaces were generally designed to provide all-purpose, fairly complete, access to a database. Until a few years ago, most of these database interfaces where nongraphical. Aside from programming language interfaces, they include relational algebras such as ISBL [35], relational calculi like QUEL [13], and query languages such as SEQUEL [2], which falls somewhere in between the two. These interfaces vary in many ways, but they did provide a large degree of completeness. Unfortunately, they were never designed to be used by novices. Then came interfaces such as QBE [38], which provides a form style interface for specifying queries. In the last few years there has also been much interest in graphical database interfaces. Among these are interfaces such as ISIS [6, 8], Schemadesign [31], Ski [21], and SNAP [3], which allow schema manipulation in an interactive graphical environment, and Databrowse [31], a graphical data manipulation tool designed for viewing and editing logical entities, rather than just relational records. They also include office forms systems such as FORMANAGER [37], Freeform [22], and SPECDOQ [24] that provide a nonexpert interface for the storage and retrieval of office data. Unlike the nongraphical systems, the graphical ones tend to be easier for nonexperts to use; however, this comes at the cost of completeness.

DBface approaches the database interface issue from a different direction. Rather than trying to create one interface that everyone can use, we feel that being able to rapidly create new or altered interfaces could prove more useful. This allows interfaces to be specifically tailored for a particular application or to be built for a certain set of users, rather than one interface being designed as a do-everything compromise. Thus, DBface has the following goals: incorporation of a UIMS-type tool into a database management system (DBMS), a unified data model for the interface and the database, encapsulation of interface data and metadata into the database, an extensible framework of reusable graphical representations of database constructs, building of a database application and its corresponding interface as an integrated unit, faster and easier design of database interfaces with minimal programming, and examination of *families* of interfaces designed for a set of related applications.

The incorporation of DBface directly into the DBMS makes several things possible. It allows the interface being built to "share" the database data model and to use the tools provided by the database (the encapsulation of interface data into the database also helps). Also, all interface data and metadata, including the interface structural description, data flow descriptions, display routines, etc., can be stored in the database. This allows for pieces of an interface to be used for building other interfaces, since everything is accessible through the database. It also allows the database to maintain interface consistency. Another issue with database interfaces is that many of them include the design of a new database application. By combining the development of the application and the interface and by using the techniques described above, DBface allows rapid development of database interfaces with minimal programming.

If interfaces can be rapidly developed, then a set or family of interfaces for a particular group of tasks becomes a reasonable idea. Instead of creating an unwieldy all-purpose interface, the designer can tailor a family of interfaces. For example, there might be a data entry interface, a data retrieval interface, and a browser for a particular application. A preliminary version of the DBface concept was described in [23]; here, we detail the step-by-step design of an interface and examine the issues involved in creating reusable interface components.

1.1 Background

Although the previous section mentions some of the unique features of DBface, it may still be unclear why it would not be easier to merely use an existing UIMS to build a variety of object-oriented database interfaces. To show how DBface differs from the UIMSs available and why these differences are important, some background on UIMSs is necessary.

To give a consistent model for looking at interface design and comparing the approach of various UIMSs, we will use the Seeheim model [9, 10] of user interfaces (Figure 1). The Seeheim model is a logical model that breaks the user interface into three logical components: the presentation component, the application interface model, and the dialogue control. The presentation component is responsible for producing device output and gathering user input. The application interface model is responsible for representing application data and making it available to the interface, as well as providing the application with access to the interface. The dialogue component forms the bridge between the presentation component and the application interface model. It makes sure that the application carries out user requests and that the presentation component produces the output requested by the applica-

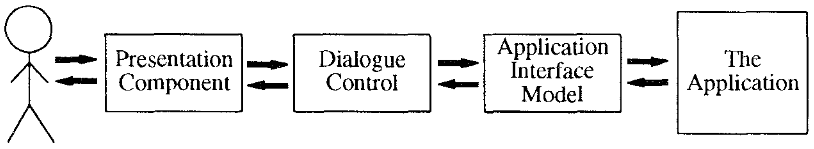


Fig. 1. The Seeheim model.

tion. Since this model is fairly general, a wide variety of UIMSs may be viewed in terms of it.

Although any UIMS must necessarily deal with all three components of the Seeheim model, many systems have focused their research most strongly on one component. Much of the UIMS research has been on the dialogue control [11]. Although there have been systems with dialogue models based on transition networks, grammars, and events, these systems share a common perspective. UIMSs such as ADM [32], Grins [27], Menulay [4], MIKE [28], and Trillium [15] focus their work on the dialogue between the user and the application. The research emphasis in these systems has been on how to provide the user with the appropriate tools for specifying this dialogue in a reasonable and convenient manner. Several other systems have centered their work on the application interface model. Filters [7] and Coral [34] each provide a method of specifying relationships between application and interface objects. GWUIMS [33] and Higgens [17] both allow for sharing of data between the application and the interface. Similarly, the presentation component has been the focus of some systems which emphasize allowing users to create new interaction techniques. Peridot [26] users create these techniques by using examples to show how they should appear, while users of [5] apply direct manipulation principles [19, 25] to specify them.

Work has also been done that centers around several components or on the interface as a whole. GROW [1] emphasizes building modifiable and reusable interfaces. Its features include communication between the application and the interface via messages, a kernel of graphical objects arranged in a taxonomic hierarchy, and user-specified interobject relationships. ITS [36] provides a tighter coupling between the application and the interface to allow for rapid development of highly interactive applications. This is done using data sharing and a layered application architecture.

Due to the differences between general application interfaces and database intefaces, we feel that currently available UIMSs do not address our needs. The most relevant differences are (a) even though some systems communicate with application data, general-purpose interfaces need no knowledge of database schemas, while this knowledge is necessary for a database interface, and (b) creating a database interface often involves creating a new application; creating a general-purpose interface usually does not.

Since DBface knows about database schemas, users may access database objects, methods, etc., and incorporate them directly into the interface. Also, interface objects are stored in the database and have the same structure as database objects. Thus, only one data model exists. Even those UIMSs which share some data between the interface and the application still require users to view interface and application objects separately and provide information about how the two are related in order to integrate them. Having a single data model makes this unnecessary. By providing access to database tools such as query languages and methods, DBface also gives users more ways of rapidly constructing an interface. Instead of writing code to generate an interface technique, a user may invoke a database method or use a query language to define the technique. Interface objects may also be reused, since they are stored in the database and are not hard-wired to any particular interface.

Many database interfaces require some new applications to be built in order to support them. For example, an office forms interface involves much more than an interface to the database. New functionalities must be built. These may include new mathematical functions such as computing the city and state sales tax on a sales field, and utilities such as an interoffice memo system. Both of these new functionalities involve more than simple interaction with the DBMS. In order to support this type of interface, we wish to allow the user to interactively design an interface and its corresponding application simultaneously [17]. Unlike in a UIMS, our approach treats the interface being designed as an integrated database environment, rather than a dialogue between a distinct user interface and an application. Instead of defining merely an interface, we define the visual, functional, and interactive aspects of the environment. Thus, certain otherwise hard-to-obtain functionalities such as binding representations to objects can be achieved. This also allows for "realistic" default interfaces and faster specification of representations. In fact, such an integration of database and UIMS technology has been suggested before [12, 29].

Although DBface only runs on an object-oriented database called Cactis [16, 18], the ideas apply to any object-oriented database. Our approach will not support interfaces outside the object-oriented database realm. This is not as limiting as it seems to be, since many applications may be placed into an object-oriented database. For example, much of the circuit board design software currently available uses simple files to store circuit information. Such systems could easily fit into the object-oriented database paradigm. One could argue that aside from making it possible to use DBface with the application, storing the data in a database would make the application itself more manageable. Also, DBface could be used to build the circuit design application as an integrated database application and interface.

1.2 The DBapp Concept

Since building a database interface often involves developing both an interface and some new application on top of the DBMS, having knowledge of the database schema and tools becomes quite important. In order to explain this, it is necessary to examine just how the structure and construction of database interfaces differs from general interfaces. In Figure 2 we see three kinds of interfaces. Part A shows a general interface, while parts B and C show database interfaces. In part B, an interface is built on top of the application,

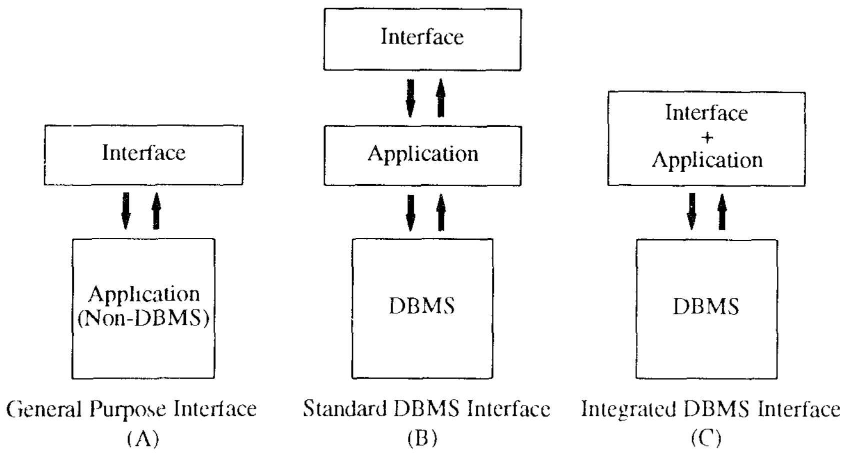
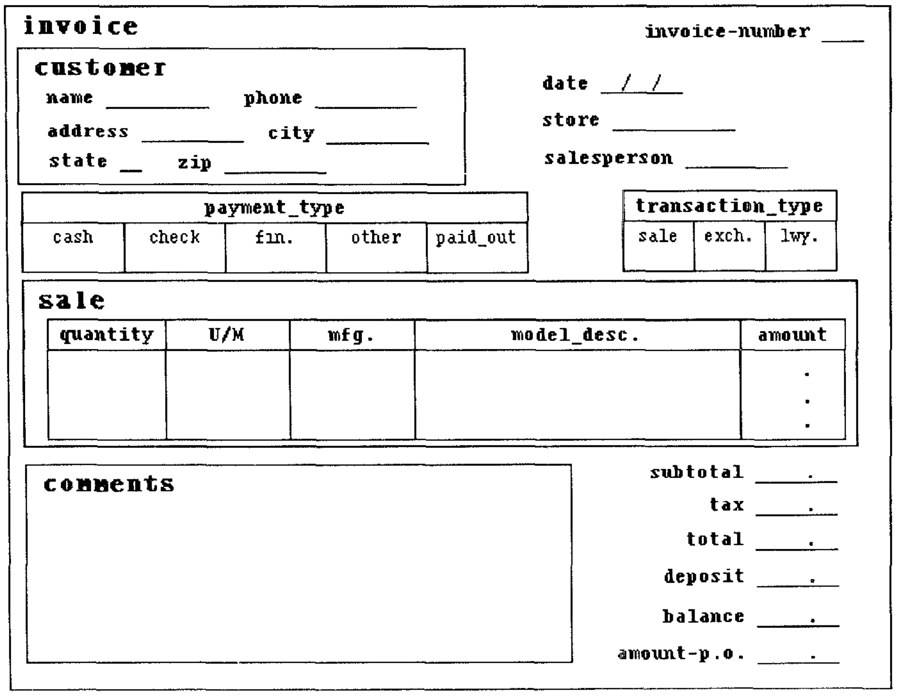

Interface
Application (Non-DBMS)
General Purpose Interface (A)
Interface
Application
DBMS
Standard DBMS Interface (B)
Interface + Application
DBMS
Integrated DBMS Interface (C)

Fig. 2. An interface comparison.

which in turn is built on top of the database. These kinds of interfaces occur often enough when dealing with DBMSs that the line between what is a database application and what is a database interface can become quite blurry. In fact, a large percentage of what we view as database applications are primarily interfaces with some applications built in. This list includes many schema browsers, schema editors, form-based data entry and retrieval systems, etc. How we view these database applications may make no difference structurally; however, we feel that part C is the most conceptually correct way to view them and thus contributes to the development of more efficient creation methodologies. From now on, we will refer to these integrated DBMS applications as *DBapps*.

When creating a traditional (nondatabase) application, the application and interface are generally built separately. The application handles functionality, and the interface is concerned with I/O. Unlike conventional interfaces, a DBapp includes both application and interface routines. For example, when designing a traditional electronic invoicing system (one that contains its own data storage and therefore is not built on top of a database), the application would generally be designed first. This would involve creating storage structures, functions for storing and retrieving data, functions for calculating totals, subtotals, taxes, etc. Next, a user interface would be built (either manually or using a UIMS).

A DBapp is generally designed in a different manner. Consider what designing this invoicing system on top of a database involves. Unlike the example above, most of the storage structures and storage and retrieval functions already exist as part of the DBMS. Therefore, much of the application already exists. What still needs to be built are the mathematical functions and the interface. Because Cactis methods (like methods in most object-oriented databases) are computationally complete, functions such as averages, totals, standard deviations, transitive closures may also be incorpo-



invoice invoice-number ____

customer

name ________ phone ________

address ________ city ________

state __ zip ________

date _/_/_

store ________

salesperson ________

payment_type				
cash	check	fin.	other	paid_out

transaction_type		
sale	exch.	lwy.

sale

quantity	U/M	mfg.	model_desc.	amount
				.
				.
				.

comments

subtotal ____.__

tax ____.__

total ____.__

deposit ____.__

balance ____.__

amount-p.o. ____.__

```
subtotal = 0; object = CURR_OBJ; att1 = amount; att2 = quantity;
FOR all_obj WHERE (all_obj RELATED_TO object)
        IF ( (TYPE_OF all_obj) equals (sale) )
                subtotal = subtotal + (all_obj.amount * all_obj.quantity);
```

Fig. 3. Invoice and associated query.

rated into the database. Note that this would be very difficult to do in a relational language such as SQL. These mathematical functions are most significant in the context of the interface (they are attached to a particular interface functionality); therefore, it makes sense to build them concurrently with the interface. This does not mean that they have to be built simultaneously (this is up to the designer), but merely that each interface object may have some attached functionality. For example, a user might want to enter sales and have the system calculate subtotals, tax, etc. The DBapp is designed so that entering a sale results in these totals being calculated. This involves creating operations for doing such things as calculating totals across a set of objects, etc. Figure 3 shows a sample invoice and the body of a routine (in a pseudoquery language) that uses the sale amounts and quantities to calculate a subtotal for a particular invoice (the invoice is the CURR_OBJ). This routine directly accesses the database in order to retrieve the attributes it needs. By making this routine part of the DBapp, the designer has the flexibility to go back and forth between building the interface part and the application part.

Designing the application and the interface concurrently is beneficial for several other reasons. First, this is a somewhat intuitive way to view the development of a DBapp. Generally, one thinks of the functionality of an interface construct while designing that construct. This does not mean that both components need be created simultaneously, but rather that they are related pieces of a more general construct. Second, this allows for a natural linking of interface constructs and their functionality. For example, both the graphical output associated with displaying an average salary and the application that actually calculates this salary can be implemented as database methods attached to some common database object type. This also solves the problem of how to represent application data within an interface, since a DBapp can directly share data with the database. Third, a DBMS maintains these constructs and the connections between them. Last, and perhaps most important, a common style of interaction for creating both visual and functional constructs is provided. This is more likely to result in a seamless DBapp. Section 4 illustrates this by showing how an interface for viewing computer networks is built with DBface.

2. ARCHITECTURE

DBface is designed to integrate smoothly with an object-oriented database. It is built on top of a DBMS named Cactis [16, 18]. Since the database is an integral part of DBface and since using DBface involves using Cactis schemas constructs and the various schema manipulation tools (DDL, C language interface, etc.) available within Cactis, we will give a brief description of Cactis before proceeding to describe the architecture of DBface.

2.1 Cactis

Cactis is designed to support applications that require complex functionally defined data. Techniques based on attribute graphs are used to optimize the maintenance of this data. Cactis views an application environment as a collection of *constructed objects*. Objects may have *attributes* and *relationships*, both of which are typed. A constructed object's type is determined by its attributes and its *connectors*. An attribute is an atomic property of a constructed object. These atomic properties may be of any C data type, except pointer. In Figure 4 (part of a Cactis DDL file), the object type **country** has the attributes **inst, name, border, new_site_in, map_drep**, and **sites_in**. **Inst**, **new_site_in**, **map_drep**, and **sites_in** are integers, while the types of **name** and **border** are constructed types defined earlier in the DDL file. The values associated with **new_site_in, map_drep**, and **sites_in** are calculated by their associated methods.

A relationship is a directional mapping from one constructed object to one or more constructed objects. Restrictions such as nonnull or unique may also be put on a relationship. Relationships are instantiated via connectors. For, example, the connectors **sites** and **cntry** could be used to create a relationship between an instance of **country** and an instance of **site**. A relationship does not exist until two instances have actually been connected. Relation-

```
instance type country
    relationships
        plug sites              : country_site;
    attributes
        inst                    : int32;
        name                    : nametype;
        border                  : bordtype;
        new_site_in             : int32 := add_site(inst);
        map_drep                : int32 := draw_map(inst,name,border);
        sites_in                : int32 := iterate tmp : int32
                                          init 0
                                          for each w0 in sites do
                                               tmp := tmp + sites.w0
                                          end;
    end;

instance type site
    relationships
        socket cntry            : country_site;
        plug to_site            : site_site;
        socket from_site        : site_site;
        plug computers          : site_comp;
    attributes
        inst                    : int32;
        site_name               : nametype;
        country_of              : int32 important := cntry.w0;
        loc                     : pairtype;
        computers_drep          : int32 := draw_all_comps(inst,site_name);
        map_site_drep           : int32 := draw_site(site_name,loc);
        connect_drep            : int32 := connect_site(inst,loc);
        computers_at            : int32 := iterate tmp : int32
                                          init 0
                                          for each w0 in computers do
                                               tmp := tmp + computers.w0
                                          end;
    end;
```

Fig. 4. Section of Cactis DDL file describing CSNET network.

ships may also be used to pass attributes from one object to another when calculating derived attributes. For example, site names could be passed to an instance of type **country** to produce a list of all the sites in a given country. This makes it unnecessary to store the information in two places.

2.2 DBface

In order to effectively manage visual representations and coordinate interface tasks, DBface is composed of the *representational* and the *operational* components (Figure 5). The representational component is responsible for managing and storing data representations (*management module*) and for display-

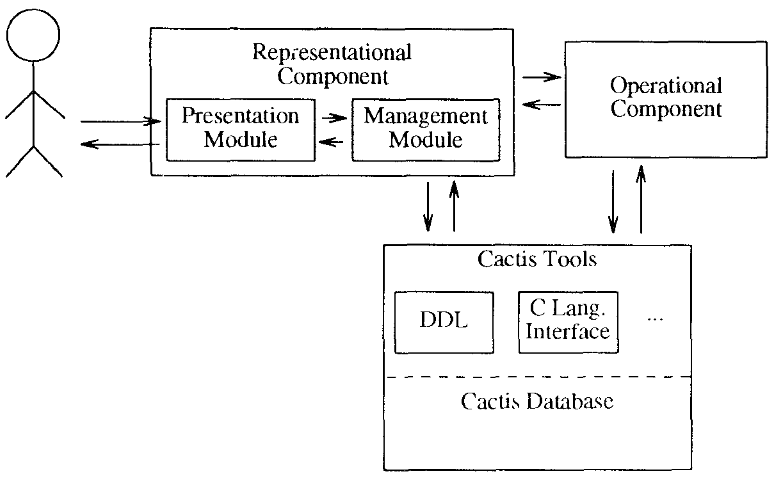


Fig. 5. DBface architecture.

ing them and handling user I/O (*presentation module*). The operational component is responsible for the functionality of the interface. This includes keeping track of state changes, defining interaction techniques, etc. Both components use the Cactis data model and have access to the database management tools provided by Cactis for data and schema manipulation. Both components also communicate directly with Cactis, but each has its own distinct tasks. Since DBface uses Cactis for storage of interface data and metadata, much of the following discussion is Cactis specific. However, the particular database being used is less important then the fact that DBface is closely tied to an object-oriented database.

The representational component is responsible for the visual aspects of an interface. Its primary function, therefore, is to deal with graphical representations of database objects. These objects, which may be either schema or data objects, may be constructed objects, relationships, or attributes. The management module builds and maintains the methods used to produce these object representations and the presentation module displays them. Thus, if we build an interface where a data object of type **country** has a representation that consists of a map of that country, the management module finds the right method and makes sure that we have the correct parameters for it, then the presentation module uses the method to actually draw the map. All the information needed to do this would typically be encapsulated within the type **country**. The secondary duties of the presentation module are the input and output associated with an interface. Input will generally consist of reading in some user data or command, while output will consist of either displaying new objects on the screen or invoking a different representation of objects already present. An object may have many different visual representations in the same interface, each of which might be used at various times within the same interface. For example, an instance of **coun-**

try could contain a map and a written description of the country. A query that requests the country's location may produce a map of the country, as well as maps of bordering countries, while a query requesting the history of the country may produce the written description.

The operational component is responsible for query resolution. It processes user queries and sends the results to the representational component so that the correct screen updates are performed. The operational component's function is very similar to that of the dialogue control in the Seeheim model. Since it uses the Cactis data model and has access to the Cactis database management tools and data, no application interface model is necessary. Instead, the operational component talks directly to the application. This has several advantages. First, it allows DBface to store the operational description of an interface within Cactis. Thus, the operational description also is a database object and therefore, Cactis can manage both the data and the operational description in a very similar manner. Not only does this make storage of interface descriptions convenient, but it also allows operations to be functionally dependent on anything present in the database. This allows an interface to change its behavior as the database is modified.

Second, since these tools give the interface direct access to the application data, semantic feedback can be gathered merely by examining the relevant data. That is, the interface can easily check what is happening within the application. Similarly, constraints can be checked and adhered to. The most important effect of allowing the operational component access to these tools occurs when designing interfaces. Instead of only being able to bind a user request to some application subroutine (which must be coded specifically for that application), the interface designer may construct a query using one of these tools. Thus new interface functionality can be added quite rapidly in most cases and much of the functionality of Cactis can be easily plugged into an interface.

DBface is implemented on Sun workstations. It is written in C and runs in a UNIX environment. Window management is done using the Sunviews window package. Sunviews also handles the graphics and user input. Since Sunviews manages the windows, they may be moved, hidden, resized, collapsed, etc., just as any other Sun window. Methods for creating representations are stored in Cactis along with the database objects they represent. No distinction is made between them and regular Cactis objects. Operation descriptions are also stored in Cactis, but they are distinct from regular Cactis objects.

3. FUNCTIONALITY

Generally, UIMS users have approached interface design by specifying screen layout, then binding each possible user action to a specific application subroutine. Often, interface routines for the application program to call are also provided. DBface takes a different approach to designing interfaces. The user may define what we view as two closely linked aspects of the interface, appearance and functionality, concurrently. When defining appearance we are really defining two kinds of visuals: interface constructs and database objects. By interface constructs we mean items such as menus, scrollbars, etc., as well as concerns like screen brightness, icon sizes, etc. Defining the appearance of database objects involves specifying representations for object types. A representation may be identical for each instance of an object type or it may be data dependent. In fact, it could be dependent on external data. For example, we could use the system clock to determine the brightness of a picture that represents the object instance **sun**.

By defining the appearance of database objects separately, we need not worry about them when defining functionality. Rather, we just specify which already defined type representation is to be used. Since the various representations for an object are encapsulated within that object, the DBMS maintains them and DBface determines which one to actually invoke. Thus, the type of the query result determines the screen appearance. Any type that has no user-defined representation will use a built-in default representation. For example, if a query results in an object of class **person**, the interface automatically uses the representation of person that has been defined as the default (the system default is used if none has been defined). The default for all objects could be as simple as printing the object type (schema) or object id (data) in a box. The user may specify a different representation when desired. If one wishes to leave the screen alone and produce the result elsewhere, this may also be specified. To show how appearance and functionality are interactively defined using DBface, we will show examples of each.

3.1 Representation Definition

Representations in DBface are defined with the representation definition window. It provides tools for building new representations and for plugging existing representations into an interface. To illustrate how representations are defined, we will look at an interface we are building for viewing a subset of CSNET [30] sites. The DBface main window allows a user to select representation definition or operation definition windows, as well as auxiliary functions needed for building interfaces. Figure 6 shows a representation definition window. The buttons on the left specify what kind of object is being worked with and the line on top is a status line. Two views are possible: *data* and *schema*. The schema view lets us work with object types, while the data view lets us work with object instances. Two levels are also available in the schema view: *schema* and *hierarchy*. The schema level shows object types and their attributes and connectors, while the hierarchy level shows a forest of classes and subclasses. The current level and view are both **schema**. Note that there has been no specific type selected, thus the current representation is applicable to all types in the database.

The representation we are viewing is the default schema representation and was selected using a popup menu (not shown). The text enclosed in rectangles represents constructed object types and the unenclosed text represents the attributes of the constructed objects. The arrow-shaped boxes represent connector plugs and the inverted arrow boxes represent connector sockets. Any constructed object with a plug may establish a relationship

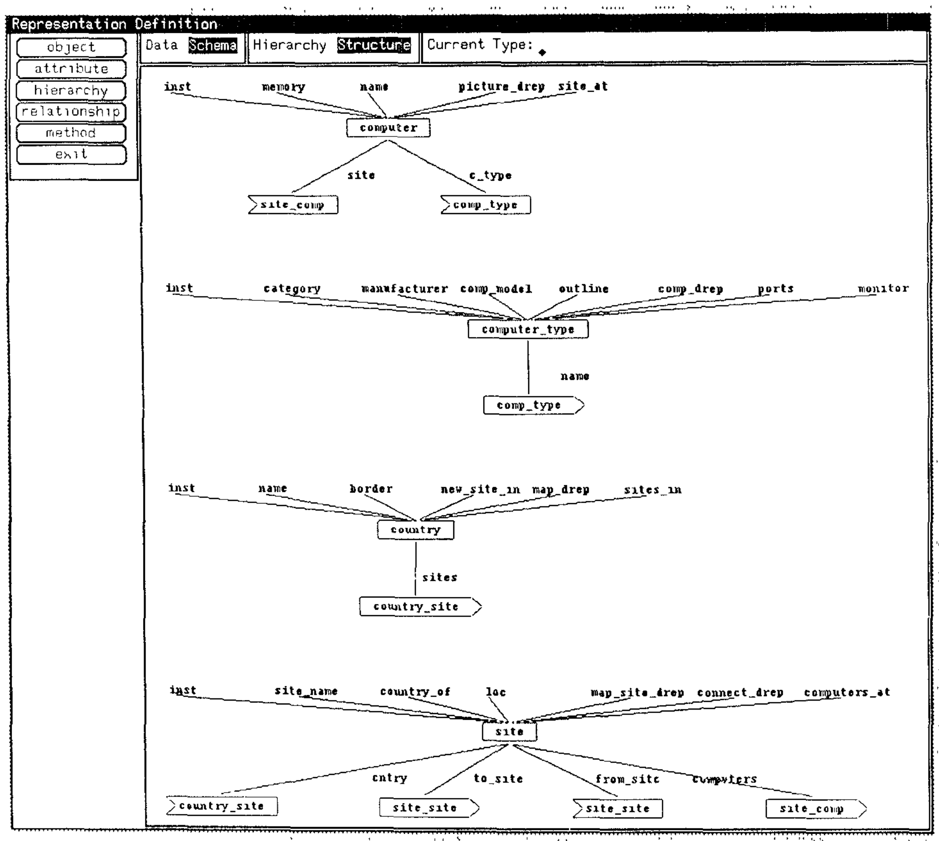
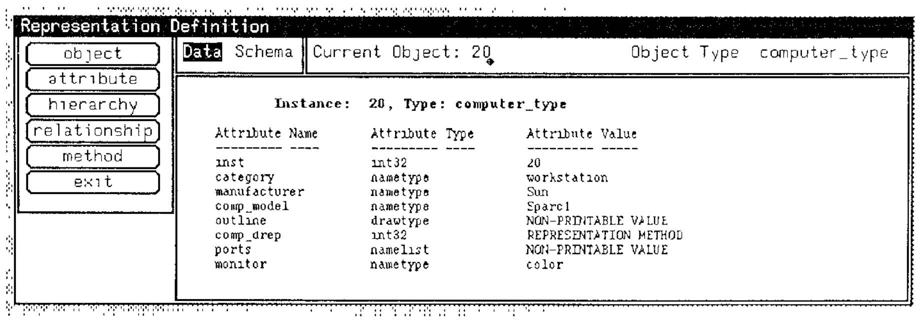


Fig. 6. Schema default representation.

with a type having a matching socket. For example, an object of type **computer** may have a relationship called **c_type** with an object of type **computer_type**. The corresponding relationship (for the object of type **computer_type**) would be **name**.

The data objects that this schema represents also have default representations. Figure 7 shows a default representation for a data object of type **computer_type**. The default data representation is merely a textual display of the object type and its attributes. Note that some attribute values are nonprintable. These are merely complex data types. They could be printed if we wished to break them down into their simple types. For lack of space, we do not do that here. Also note that **comp_drep** is an attribute that corresponds to a representation method (attributes ending with **_drep** are associated with data representations while those ending with **_srep** are associated with schema representations).

Figure 8 shows a nondefault data representation that draws a particular computer type, along with its manufacturer and model name. This representation was selected from the same popup menu as the default and was



```
Instance:  20, Type: computer_type

Attribute Name      Attribute Type      Attribute Value
--------- ----      --------- ----      --------- -----
inst                int32               20
category            nametype            workstation
manufacturer        nametype            Sun
comp_model          nametype            Sparc1
outline             drawtype            NON-PRINTABLE VALUE
comp_drep           int32               REPRESENTATION METHOD
ports               namelist            NON-PRINTABLE VALUE
monitor             nametype            color
```

Fig 7. Data default representation

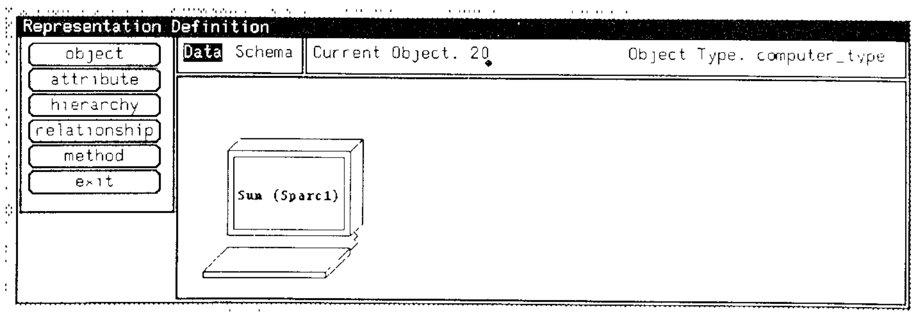


Fig 8. Computer_type data representation.

displayed using the method associated with the attribute **comp_drep** (the suffix **_drep** is used by DBface to ascertain that **comp_drep** is a representation method and should be listed in the menu along with the default representation). The data that this method needs in order to draw this representation is stored in other attributes of the object type **computer_type**. One of these attributes is a list of the data points needed to draw the representation. Others include the computer manufacturer, model name, etc. Note that once this method has been written, it is available for any interface to use since it is part of the object type **computer_type**. This encapsulation allows DBface to easily keep track of the various representations available for any data type. Since DBface keeps track of the representations available for various object types, interfaces can easily access a variety of representations. Not only does this make it easy to build an interface, it also allows for easy reuse of the data representations when building more than one interface. In fact, the various representations could be made into a library by providing the appropriate access routines.

3.2 Representation Storage

DBface representations are stored as reusable methods dependent on some set of object attributes. Although an attribute could be a simple bitmap to be dumped by the method, this is not the way that most data representations

```
instance type computer_type
    relationships
        plug name       : comp_type;
    attributes
        inst            : int32;
        category        : nametype;
        manufacturer    : nametype;
        comp_model      : nametype;
        outline         : drawtype;
        comp_drep       : int32 := draw_computer
                              (manufacturer,comp_model,outline);
        ports           : namelist;
        monitor         : nametype;
    end;
```

Fig. 9. Selection of Cactis DDL file describing computer types.

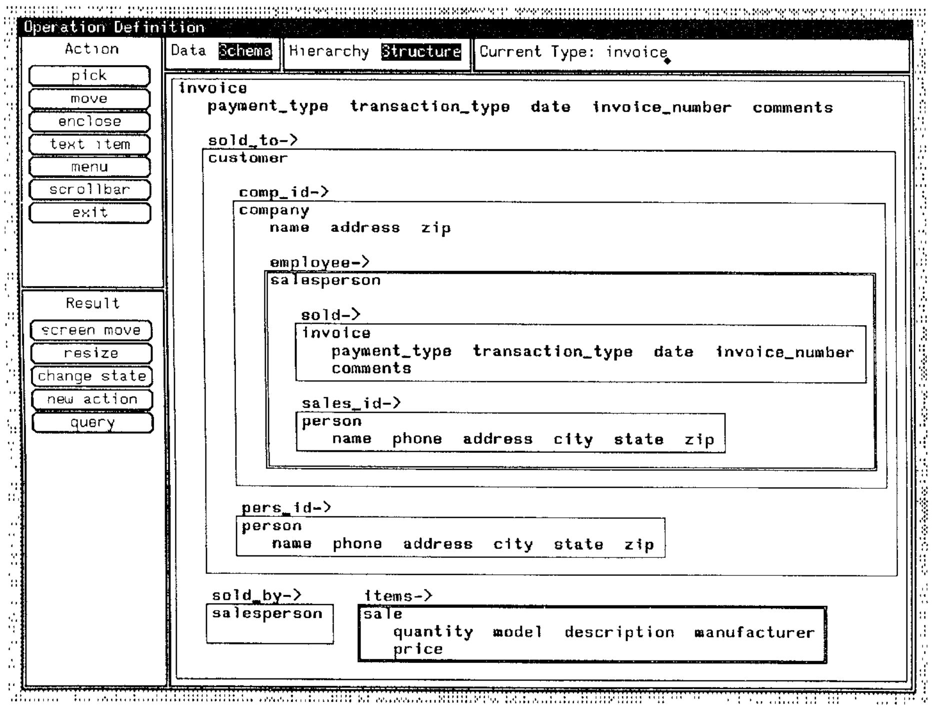
are done. Generally, the data provided allows the representation method to create the representation in a more intelligent manner than a simple bitmap. The choice, however, is up to the designer. For example, the way that the computer type shown in Figure 8 is stored and drawn can be seen by examining the object type **computer_type** in the section of a data definition file shown in Figure 9. The method for drawing a computer type is connected to the attribute **comp_drep** and dependent on the attributes **comp_name, comp_model**, and **outline**. **Outline** is the attribute that contains the actual coordinates for drawing a picture of the given computer.

Another important aspect of the storage method used is the transparent integration of new and old data representations. Data representations are maintained by Cactis and are not hard-wired to one particular interface. Also, because all the data representations created previously look no different than newly created ones, they may be used for building new interfaces and as building blocks for creating new representations. This is a key factor in providing representation reusability.

3.3 Operation Definition

Operations defined in DBface include menu actions, queries, etc., as well as state changes used for control flow. These operations are created using the operation definition window. We will demonstrate some of the features of the operation definition window by showing how some features of a more traditional database interface would be built. As our example, we will describe how part of an existing interface, an office forms system called Freeform [22], would have been built with DBface.

Figure 10 shows an operation definition window. Although there is no reason we cannot have both the representation definition and operation definition windows up simultaneously, we will only use one at a time in order to make our diagrams less crowded. Unlike the schema in Figure 6, this representation uses inclusion to represent relationships. Thus, an object related to a second object will be included inside the second object. For



```
invoice
   payment_type  transaction_type  date  invoice_number  comments
  sold_to->
  customer
     comp_id->
     company
        name  address  zip
        employee->
        salesperson
           sold->
           invoice
              payment_type  transaction_type  date  invoice_number
              comments
           sales_id->
           person
              name  phone  address  city  state  zip
     pers_id->
     person
        name  phone  address  city  state  zip
  sold_by->        items->
  salesperson      sale
                      quantity  model  description  manufacturer
                      price
```

Fig. 10. A schema form representation.

example, an object of type **invoice** is related to an object of type **customer** through the relationship **sold_to**. Note that the direction of a relationship is fairly arbitrary. Attributes are also included within the constructed object they belong to. The label on top of each box is the name of the relationship between the object represented by the box and the object it is included in. The type of an object is in the upper left corner of its surrounding box. The bold box around **salesperson** denotes that the relationship **employee** (between **company** and **salesperson**) is one-to-many. Since Cactis uses connectors to create relationships, the relationships shown in this diagram are really only potential ones. To have real relationships, one must talk about object instances rather than object types.

Figure 11 shows a further refined form representation. Although this representation was developed from the previous one, it deals with data, rather than schema, information. Thus, the relationships in this representation have been instantiated. Aside from following real connections, this representation also provides blanks for filling in or displaying attribute values. Further refinement will include making representations different for various types of attributes, and representing multivalued relationships such as **items** with some appropriate structure like a table. Once we finish with our changes and store them, the newly created schema representation will become available from the popup menu.

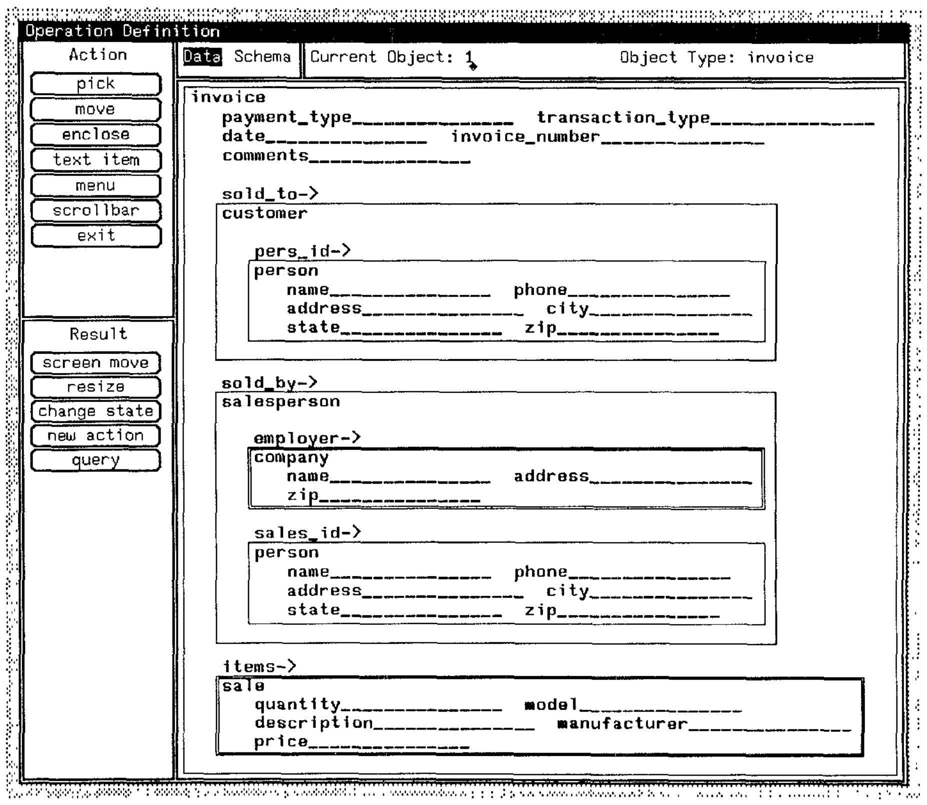
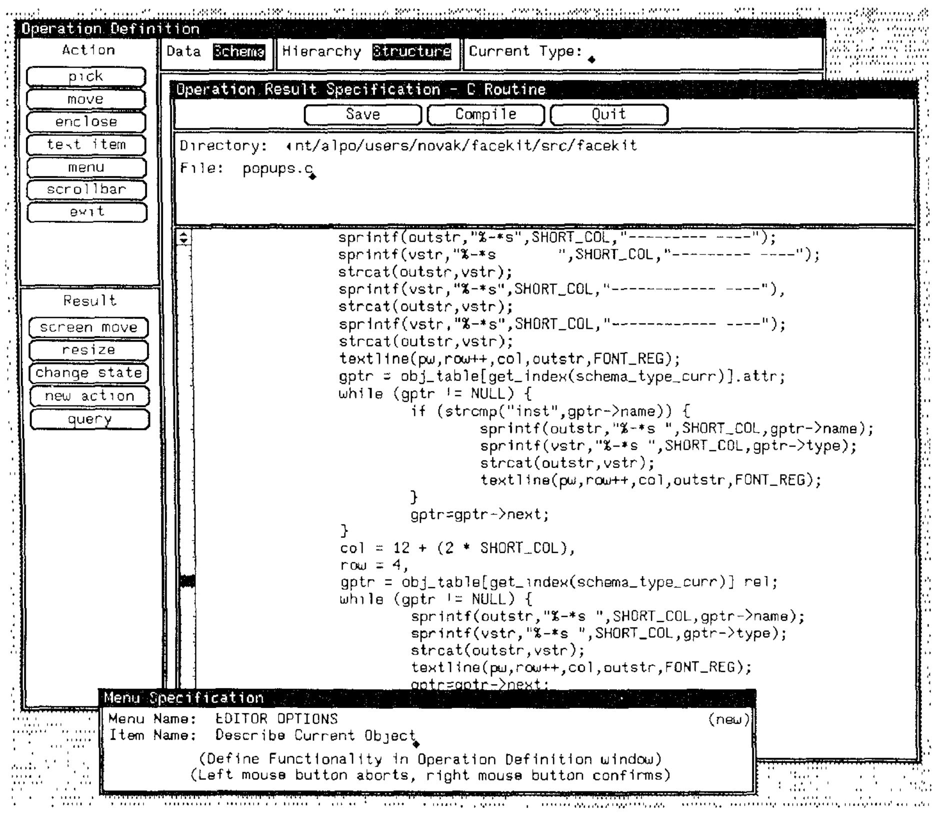


Fig. 11. A data form representation.

Working with the operation window, a user can create interface techniques. For example, by selecting an action (or sequence of actions) from the Action buttons and a result (or sequence of results) from the Result buttons, the user can bind the desired functionality to a set of actions. Suppose the sequence **pick, new action,** and **menu** is selected. This specifies the following sequence. The user picks an object. The result of this is a new action. This new action will be the appearance of a menu. Thus, a pick in Freeform will create a menu. Since a menu is to be created, a Menu Specification window pops up, for the user to specify which menu to use. Figure 12 shows such a window, with a new menu being defined. We could have also chosen an already defined menu, but instead, we will demonstrate how one goes about defining a menu. This is done by defining each menu item and its corresponding functionality. The menu item being defined in Figure 12 is **Describe Current Object**. The result of choosing this menu item will be query, which will be defined in the Operation Result Specification window.

We will define this query using the C interface to Cactis. The query starts by doing some type checking to make sure a legal schema type is being described. If it is not, an error message is displayed in the output window and



```
Directory:  ‹nt/alpo/users/novak/facekit/src/facekit
File:  popups.c
```

```
        sprintf(outstr,"%-*s",SHORT_COL,"--------- ----");
        sprintf(vstr,"%-*s       ",SHORT_COL,"--------- ----");
        strcat(outstr,vstr);
        sprintf(vstr,"%-*s",SHORT_COL,"----------- ----"),
        strcat(outstr,vstr);
        sprintf(vstr,"%-*s",SHORT_COL,"----------- ----");
        strcat(outstr,vstr);
        textline(pw,row++,col,outstr,FONT_REG);
        gptr = obj_table[get_index(schema_type_curr)].attr;
        while (gptr != NULL) {
                if (strcmp("inst",gptr->name)) {
                        sprintf(outstr,"%-*s ",SHORT_COL,gptr->name);
                        sprintf(vstr,"%-*s ",SHORT_COL,gptr->type);
                        strcat(outstr,vstr);
                        textline(pw,row++,col,outstr,FONT_REG);
                }
                gptr=gptr->next;
        }
        col = 12 + (2 * SHORT_COL),
        row = 4,
        gptr = obj_table[get_index(schema_type_curr)] rel;
        while (gptr != NULL) {
                sprintf(outstr,"%-*s ",SHORT_COL,gptr->name);
                sprintf(vstr,"%-*s ",SHORT_COL,gptr->type);
                strcat(outstr,vstr);
                textline(pw,row++,col,outstr,FONT_REG);
                gptr=gptr->next;
```

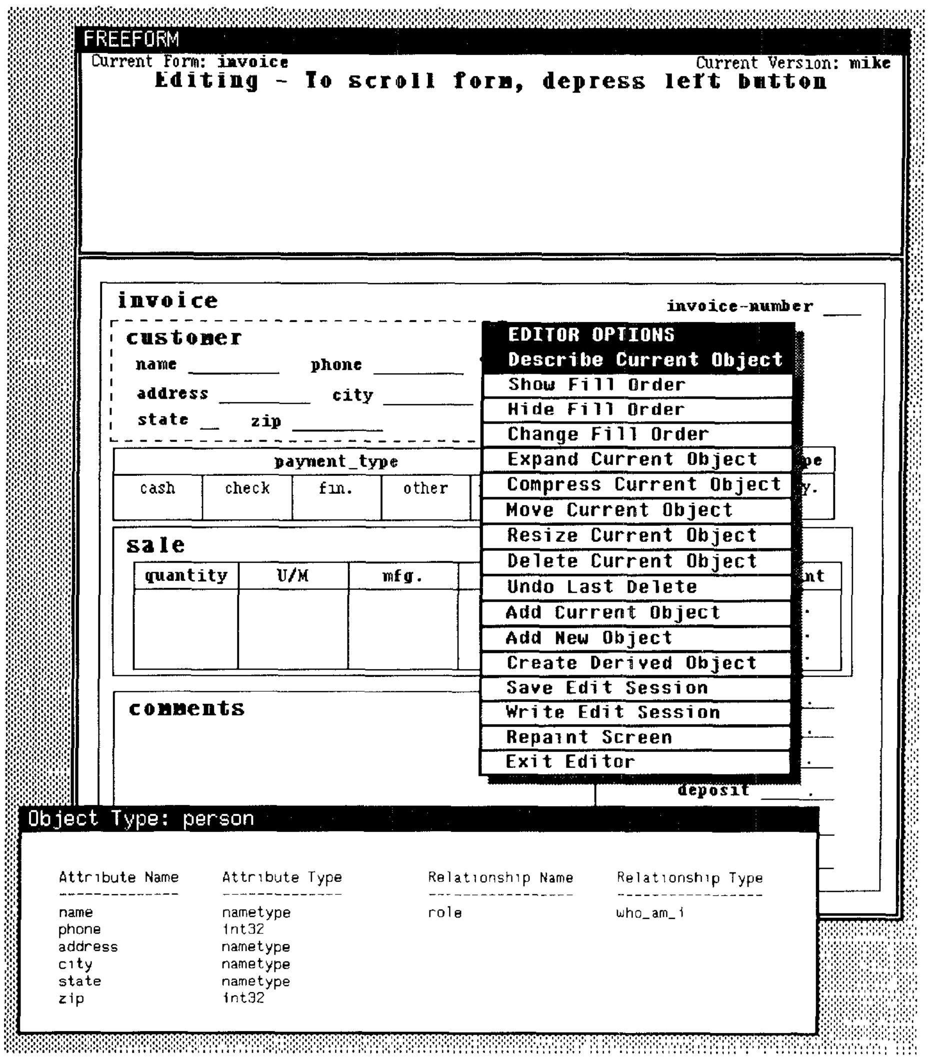
```
Menu Name:  EDITOR OPTIONS                                  (new)
Item Name:  Describe Current Object
        (Define Functionality in Operation Definition window)
        (Left mouse button aborts, right mouse button confirms)
```

Fig. 12. Adding functionality to a menu item

the query is complete. Otherwise, all attributes and relations valid for the schema type being described are found by walking the database schema. This information is formatted and displayed in an output window. Although many UIMSs allow C routines to be bound to interface constructs [14], the advantage here is that DBface implements them as Cactis methods, thus giving all queries direct access to any database data or data dictionary information needed. By having the DBMS manage functions, DBface gains more computational power than by merely using C routines.

The result of selecting the **Describe Current Object** menu item from within Freeform is shown in Figure 13. Because every object has either an explicit or implicit visual border, the current object (the one being picked) is the innermost object whose border the cursor is within when the menu is selected. In this case it is the form field labeled customer (object type **person**) that is surrounded by the dashed line. Note that the query also finds the connector **role**, even though it is not part of the form being displayed. This is because it is part of the object type **person**.

DBface also provides a mechanism for control flow through the use of state diagrams. The operation definition window provides tools for creating states



Current Form: invoice Current Version: mike

Editing - To scroll form, depress left button

Attribute Name	Attribute Type	Relationship Name	Relationship Type
name	nametype	role	who_am_i
phone	int32		
address	nametype		
city	nametype		
state	nametype		
zip	int32		

Fig. 13. Selecting a Freeform menu item.

and binding state changes to user actions. Although state diagrams are not new [20], and DBface does not enforce their use, we feel they are an effective methodology and strongly encourage them. Like other interface constructs, state diagrams are stored as sets of related database objects. An example of their use will be shown in the next section.

3.4 Concurrency and Recovery

Previous sections have discussed the benefits of using the DBMS for creating interfaces and for storing application data. There are also several advantages provided by the database when actually using an interface built with DBface.

Since the interface is being run as a series of database transactions, data consistency is automatically provided by the DBMS transaction mechanism. This makes it possible to provide concurrency and recovery for any interface built with DBface.

Often, we may let several users execute an interface concurrently. With conventional UIMSs, this capability must be built into the application (or perhaps into the interface). Without the proper safeguards in place, the data may be rendered inconsistent. In interfaces built with DBface, the DBMS takes care of problems created by concurrent access. Let us examine an interface that uses the invoice in Figure 11 for data entry. If two users are both adding a sale at the same time, there is potential for data inconsistency. The DBMS will sequence the two transactions and maintain consistency. This is invisible to the user.

Another potential conflict can be seen if we look at the form being edited in Figure 13. If two users are editing the same form at the same time and both try to add a new field in the same location, we wish to only allow one of them to succeed. Here, the DBMS automatically locks out one user while the other adds the field. Even though one user's attempt to add a field will now fail, the DBMS will automatically recover. The transaction is rolled back to a point where data consistency is achieved. Although recovery may not be invisible (there will probably be some error message), the data will not be compromised. Recovery from system crashes, power failures, etc. is handled similarly. Although any UIMS could call a DBMS routine and provide transaction management for a particular operation, the tighter coupling between the interface and the database in DBface allows for more flexibility. One or more operations may be made into a transaction or the whole interface may be a single transaction. This allows interfaces to have correct, concurrent behavior at any level of granularity desired.

4. A NETWORK EXAMPLE

A software engineer might want an easy way to get information about computer networks. With network configurations changing fairly often, storing the information in a database and having a graphical interface to access this information makes good sense. Because such an interface does not look much like a "regular" database interface, it is useful for illustrating the variety of interfaces that can be built with DBface.

Although an existing interface may be used as a starting point, here we will create a brand new interface called **network**. This is done from the DBface main menu (not shown) and causes several internal events to happen. First, instances of object type **interface** and **state** are automatically added to the database. These objects types, as well as some others, are used to keep track of the structure and functionality of interfaces built with DBface. The structure of these types is automatically provided by the DBMS. In order to establish a start state for this interface, these two instances are connected with a relationship and the new instance of type **state** is made the start state. Now a new interface with a start state exists.

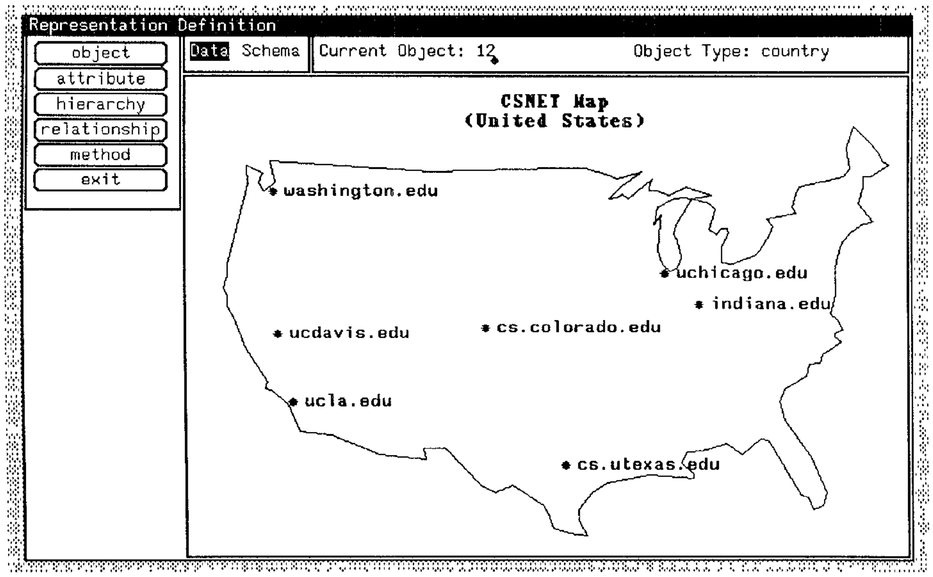


Fig. 14. A CSNET map representation.

Second, a makefile and a main program are automatically generated for this interface. The main program initializes the new interface by setting up the necessary Sunviews protocols and creating an initial I/O window. The makefile compiles in the main program, the necessary DBface code, and Cactis. If we compiled and executed our new interface at this time, it would merely bring up an initial window for the interface. Now we need to start adding representations and functionality to our interface.

4.1 Creating Some Representations

To create the interface, we need to have representations of all the database objects that the interface will display. These include the objects of type **country, site, computer**, etc. These representations also need to be "layered" so that we may show combinations of objects in some meaningful way.

In Figure 14, we show a pictorial representation of a small subset of the CSNET sites in the United States. This representation was built by invoking several representation methods using a layered approach. The map is drawn using two different representation methods (shown in the partial DDL file in Figure 4). The attribute **map_drep** of type **country** is used to invoke the method **draw_map**, which draws the map of a country (in this case the U.S.A.). **Draw_map** follows the relationship sites to find all the sites in this country. For each site visited, the method **draw_site** is invoked through the attribute **map_site_drep**. This draws the individual sites. No arguments need to be passed to the methods since they are encapsulated within each object instance. Thus, **draw_map** only has to follow the relationship **sites** to

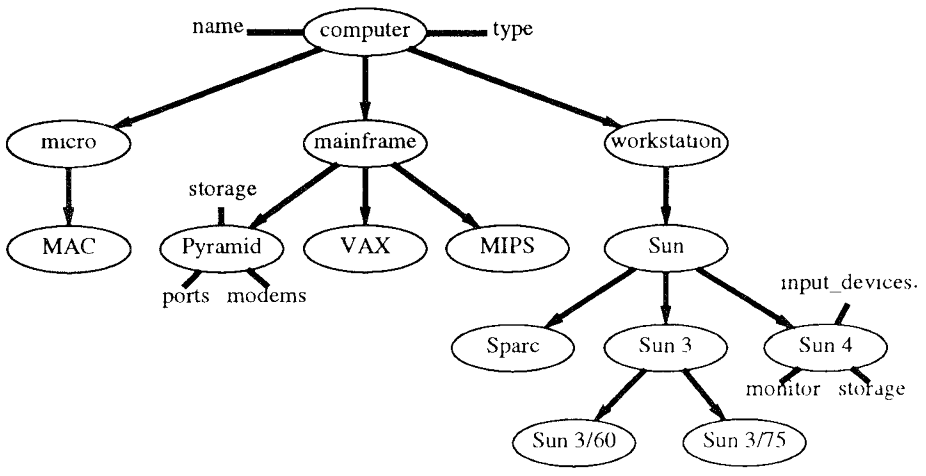


Fig. 15 Class hierarchy.

each **site** instance and then reference the attribute **map_site_drep** for the site to be drawn.

Each **site** contains a number of computers. Figure 15 shows a class hierarchy for the schema type **computer**. In this hierarchy, **computer** is the base type, **mainframe, workstation**, and **micro** are subtypes, and so on. Classes are represented by enclosed text and attributes are represented by unenclosed text. Attributes are connected with lines, subclasses are connected with arrows. Although this hierarchy is not complete (for readability), we can see that computers such as a **Pyramid** and a **Sun** 4 are different and contain different attributes. The database contains methods (accessible to DBface) which know about these differences and will later be used to draw the computers that exist at a particular site.

4.2 Adding Control Flow

Aside from having the necessary representations available, the network interface must also be able to handle state transitions. This makes it possible for the operational component to process queries at the proper times and let the representational component know what representations are valid at any given time. Similarly, the system must also keep track of when input is necessary.

Figure 16 shows a DBface Operation Definition window being used to define the control flow of the network interface. The state diagram for the network interface is being displayed and state 3 (highlighted) is the current state. The arcs (defined below the diagram) all correspond to menu choices. Note that there is no way to leave state 3 at this time. The popup menu shown to the right of the state diagram is being used to select a new state to go to. This popup was bought up after a user specified that a menu pick would result in a state change. This was done by selecting the **menu** button

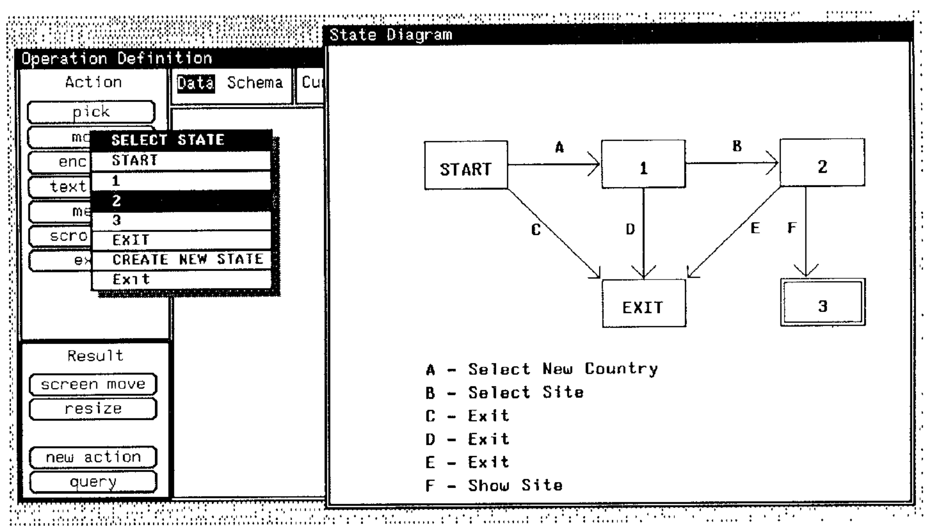


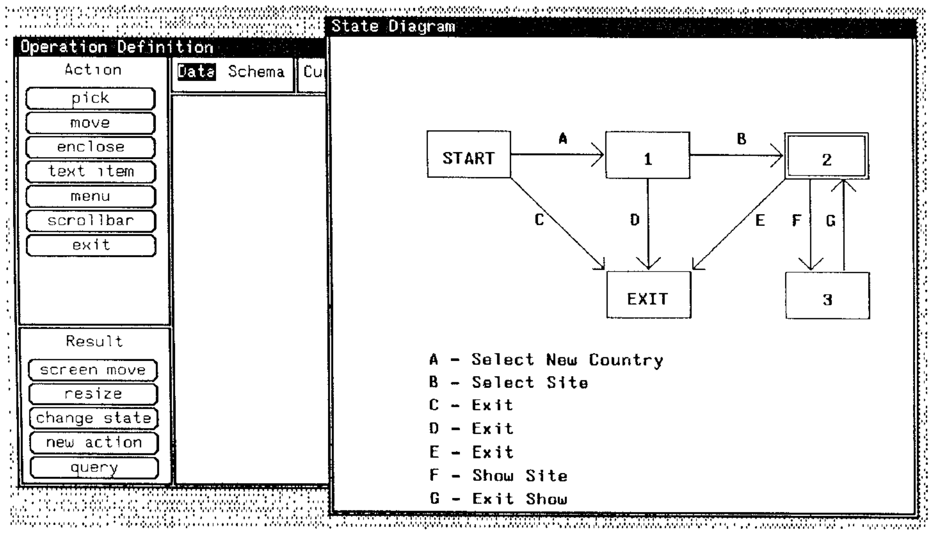
Fig. 16. The network interface state diagram.

from the set of **Action** buttons and the **change state** button from the **Result** buttons. The **change state** button is not visible during the life of the popup menu, but it can be seen in Figure 17. Once a state is selected, the user will be prompted for more information about the menu choice that causes the state change. However, the system already knows that it is the menu available in state 3 that the choice will come from.

Figure 17 shows the state diagram with the new arc from state 3 to state 2 added. The menu choice that will enable this state change will be **Exit Show**. There is now a way to reach every state and a way to get from any state to the exit state. Since state diagrams are maintained by the DBMS, they are also reusable. **Network** was built in a couple of days using DBface. Since most of the representations needed for both input and output were already part of the database, they were merely "plugged in" to **network**. As a result, the total amount of code written by hand for this interface was less than 500 lines of C. About half of this code consisted of new data representations and the rest consisted of some Sunviews font, window, and event-handling routines.

4.3 The Network Interface

Upon entry, the interface lets the user select a country, then a site. Once we do this, we can also request that **network** show us the site connections. This results in the map seen in Figure 18. The country and site name are displayed as part of the window header and all the site connections the current site (cs.colorado.edu) has are graphically displayed. The connections are drawn by accessing a representation method attached to the object type **site**. This method finds all the sites connected to the site of interest and then draws the connecting lines. Other database methods attached to the object



A - Select New Country
B - Select Site
C - Exit
D - Exit
E - Exit
F - Show Site
G - Exit Show

Fig. 17. Adding a state to the network interface state diagram.

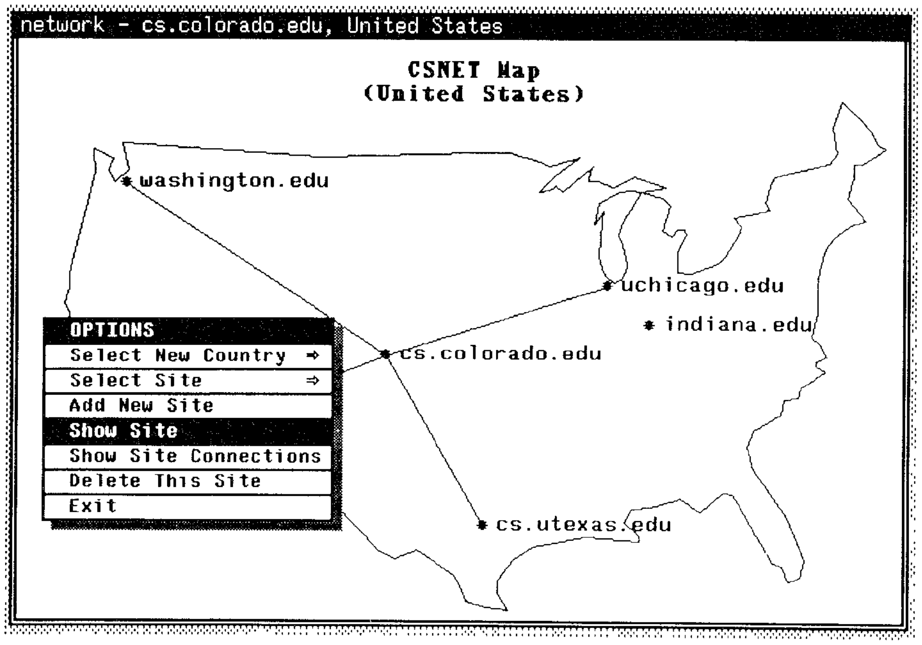


Fig 18. Showing site connections.

type **site** could also be "plugged in." For example, a method that draws a pictorial representation of a site could be selected as alternative feedback.

Now we can see what resides at the current site. Selecting **Show Site** in Figure 18 shows the computers at **cs.colorado.edu**. Note that a small sub-

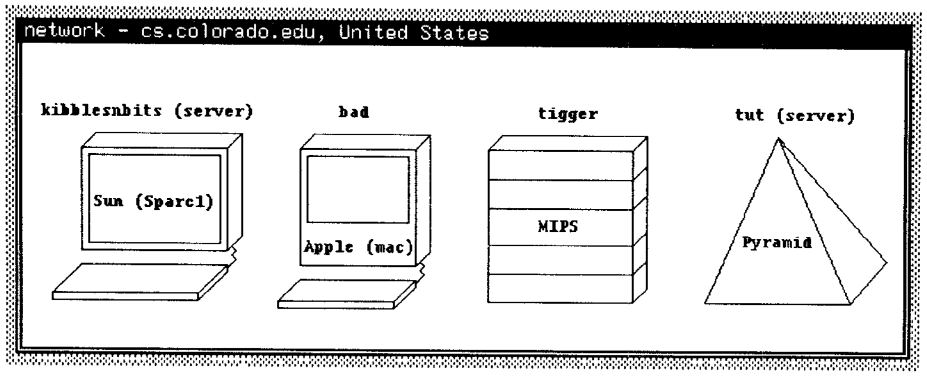


Fig. 19. Showing the computers at a site.

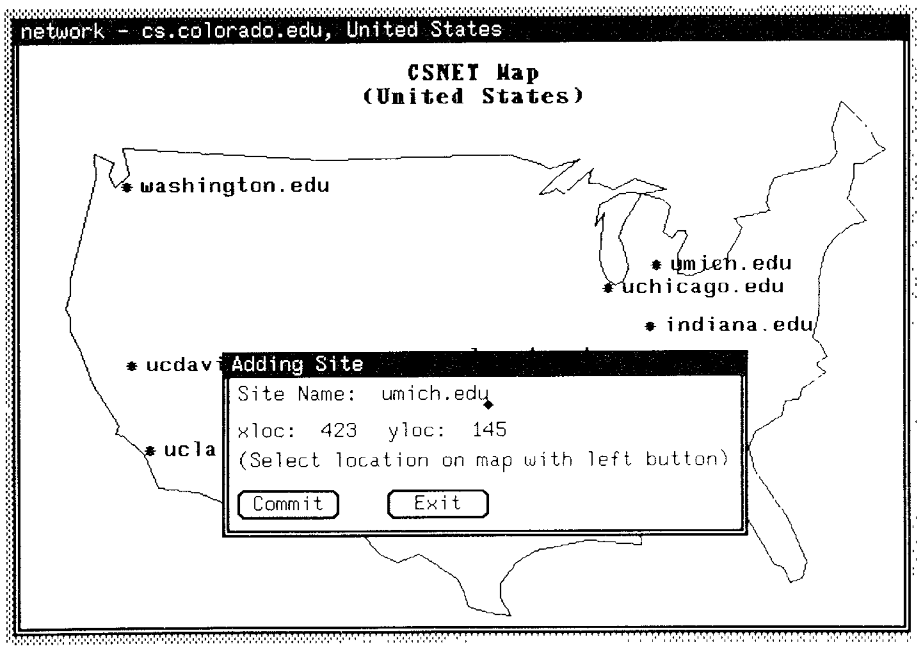
set of computers is being used to represent this site. The method **draw_all_comps** (Figure 4) is used to draw the representation seen in Figure 19. It is built up from the representations for object types **computer** and **computer_type**. Each computer at the selected site is shown along with its type and name. Which computers at a site are servers is also shown. In this example, the servers are **kibblesnbits** and **tut**. The network interface does not even need to know the specifics about each computer at that site. For each computer, the database knows its type and the proper method to invoke for that type.

Since different computers have different attributes, the information given for a Sun will differ from the information given for a Pyramid. The database knows these differences and therefore, much of our work is done for us, especially if there is already a built-in method for displaying attribute information.

Data entry methods can also be maintained by DBface. Figure 20 shows a new site being added. The function that does this, including bringing up the new window and gathering input, is a method attached to the object type **country**. The site name is entered in the small window and the location is entered by clicking the mouse on the map. The location could also be typed in if desired. Then the user selects the commit button to actually create the site. Adding this site causes several things to happen. First, an instance of type **site** is added to the database. Next, this site's **site_name** and **loc** attributes are given values. Then the site is connected to the current instance of **country** and drawn on the map.

5. FUTURE DIRECTIONS

We would like to make a few additions to DBface. These include a debugging tool for keeping interfaces consistent with DDL changes, techniques for the automatic generation of new representation methods from existing ones, and a facility for bootstrapping interfaces built with DBface. The state diagram mechanism is also a likely candidate for added functionality. We have built



Site Name: umich.edu
xloc: 423 yloc: 145
(Select location on map with left button)

Fig. 20 The new site added.

only state diagrams that are not much larger than a Sunviews window and this will not be sufficient in the long run.

We would also like to look at some ways of letting interface designers spend more time being creative and less time developing specific constructs. First, we would like to see if adding some new interactive techniques lets designers spend less time building their own techniques. Would we end up with different types of interfaces? Similarly, would the addition of some specialized database access constructs and routines help minimize the amount of database knowledge a user needs to have, such as knowledge of the DDL? And last, we would like to examine the effects of combining the operational and representational components. Since this would allow all components to be built simultaneously and in the same style, we think it may make an interface designer's job more intuitive and easier.

There are also some broader issues that still need to be examined. First is the issue of portability. Although we feel we have validated the concept of integrating UIMS and DBMS technology, we would like to see if it is possible to create a tool like DBface that can function with a wide set of object-oriented databases. This would also necessitate devising a global DDL that easily translates into a variety of DDLs. And second, we wish to see if the techniques used by DBface can be successfully used by UIMSs. Although techniques that depend on extensive knowledge of the application may not translate well to a general-purpose UIMS, they may prove useful in special-application UIMSs.

DBface has proven to be a useful tool for prototyping interfaces. Although we still need to test the system with nonexpert users, within our research group it has significantly reduced the amount of time needed to prototype an interface. Currently, we have built one complete interface (the network example) and some pieces of another (Freeform); both of these were built much faster than if they had been done by hand. The network interface was built in less than two days. Our guess is that had it been built by hand it would have taken over two weeks.

ACKNOWLEDGMENTS

We thank all the referees, whose comments and suggestions have vastly improved this paper. DBface was previously called FaceKit [23] and has been renamed due to the emergence of a commercial product with that name.

REFERENCES

1. BARTH, P. S. An object-oriented approach to graphical interfaces. *ACM Trans. Graph. 5*, 2 (Apr. 1986), 142–172.
2. BOYCE, R. AND CHAMBERLIN, D. SEQUEL: A structured English query language. In *Proceedings of the ACM-SIGFIDET Workshop on Data Description, Access and Control* (May 1974), 219–261.
3. BRYCE, D. AND HULL, R. SNAP: A graphics-based schema manager. In *IEEE Conference On Data Engineering* (1986), 151–164.
4. BUXTON, W., LAMB, M. R., SHERMAN, D. AND SMITH, K. C. Towards a comprehensive user interface management system. *Comput. Graph. 17*, 3 (July 1983), 35–42.
5. CARDELLI, L. Building user interfaces by direct manipulation. In *ACM UIST Proceedings* (1988), 152–166.
6. DAVISON, J. W. AND ZDONIK, S. B. A visual interface for a database with version management. *ACM Trans. Off. Inf. Syst. 4*, 3 (July 1986), 226–256.
7. EGE, R. K. Defining constraint-based user interfaces. *Data Eng. 11*, 2 (June 1988), 54–63.
8. GOLDMAN, K. J., GOLDMAN, S. A., KANELLAKIS, P. C. AND ZDONIK, S. B. ISIS: Interface for a semantic information system. In *SIGMOID Conference Proceedings* (May 1985), 328–342.
9. GREEN, M. Report on dialogue specification tools. *Comput. Graph. Forum 3* (1984), 305–313.
10. GREEN, M. The University of Alberta user interface management system. *Comput. Graph. 19*, 3 (July 1985), 205–213.
11. GREEN, M. A survey of three dialogue models. *ACM Trans. Graph. 5*, 3 (July 1986), 244–275.
12. GREEN, M. Directions for user interface management systems research. *Comput. Graph. 21*, 2 (Apr. 1987), 113–116.
13. HELD, G., STONEBRAKER, M. AND WONG, E. INGRES: A relational data base management system. In *Proceedings of the AFIPS National Computer Conference 44* (Anaheim, Calif., May 1975), 409–416.
14. HELFMAN, J. I. Panther: A specification system for graphical controls. In *CHI + GI 87 Proceedings* (Toronto, April 1987), 279–284.
15. HENDERSON, D. A. The Trillium user interface design environment. In *CHI 86 Proceedings* (Boston, April 1986), 221–227.
16. HUDSON, S. AND KING, R. CACTIS: A database system specifying functionally-defined databases. In *Proceedings of the International Workshop on Object-Oriented Databases* (Pacific Grove, Calif., Sept. 1986), 26–37.
17. HUDSON, S. AND KING, R. Semantic feedback in the Higgens UIMS. *IEEE Trans. Softw. Eng. 14*, 8 (Aug. 1988), 1188–1206.
18. HUDSON, S. AND KING, R. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Trans. Database Syst. 14*, 3 (Sept. 1989), 291–321.

19. Hutchins, E. L., Hollan, J. D. and Norman, D. A. Direct manipulation interfaces. In *User Centered System Design*, Lawrence, Erlbaum, 1986, 87–124.

20 Jacob, R. J K. A specification language for direct-manipulation user interfaces *ACM Trans. Graph* 5, 4 (Oct. 1986), 283–317.

21. King, R. and Melville, S. Ski: A semantic-knowledgeable interface. In *VLDB Conference Proceedings* (Singapore, Aug. 1984), 30–33.

22. King, R. and Novak, M. Freeform A user-adaptable form management system. In *VLDB Conference Proceedings* (Brighton, England, Sept. 1987), 331–338

23. King, R. and Novak, M. FaceKit: A database interface design toolkit. In *VLDB Conference Proceedings* (Amsterdam, Aug. 1989), 115–123.

24. Kitagawa, H , Gotoh, M , Misaka, S and Azuma, M. Forms document management system SPECDOQ—its architecture and implementation. In *SIGOA Conference Proceedings* (Toronto, June 1984), 132—142.

25 Lee, A., and Lochovsky, F. H Enhancing the usability of an office information system through direct manipulation In *CHI 83 Conference Proceedings* (Boston, Dec. 1983), 130–134.

26. Myers, B. A. Creating dynamic interaction techniques by demonstration. In *Proceedings of the 1987 CHI + GI*, (Toronto, April 1987), 271–278.

27. Olsen, D. R., Dempsey, E P. and Rogge, R. Input/Output linkage in a user interface system. *Comput. Graph. 19*, 3 (July 1985), 191–197.

28. Olsen, D. R. MIKE: The menu interaction control environment. ACM *Trans. Graph. 5*, 4 (Oct. 1986), 318–344.

29. Olsen, D. R. Larger issues in user interface management. *Computer Graphics (ACM SIGGRAPH Workshop on Software Tools for User Interface Management) 21*, 2 (Apr. 1987), 134–137

30 Quarterman, J. S and Hoskins, J. C. Notable computer networks. *Commun. ACM 29*, 10 (Oct 1986), 932–971

31. Rogers, T. R. and Cattell, R. G. G Entity-Relationship database user interfaces In *Readings in Database Systems*, Morgan Kaufmann, 1988, 359–368.

32. Schulert, A. J., Rogers, G. T. and Hamilton, J. A. ADM—A dialog manager. In *Proceedings of the 1985 CHI* (San Francisco, April 1985), 177–183.

33 Sibert, J. L., Hurley, W. D. and Bleser, T W. An object-oriented user interface management system. *Comput. Graph. 20*, 4 (Aug. 1986), 259–268.

34. Szekely, P. A. and Myers, B. A. A user interface toolkit based on graphical objects and constraints. In *OOPSLA Proceedings* (San Diego, Calif., Sept 1988), 36–45.

35. Ullman, J *Principles of Database Systems*. 2nd ed., Computer Science Press, Rockville, Md

36. Wiecha, C., Bennett, W., Boies, S., Gould, J. and Greene, S. ITS: A tool for rapidly developing interactive applications. *ACM Trans. Inf Syst. 8*, 3 (July 1990), 204–236.

37. Yao, S B., Hevner, A R Shi, Z and Luo, S FORMANAGER: An office forms management system. *ACM Trans. Off. Inf. Syst. 2*, 3 (July 1984), 235–262.

38. Zloof, M. M. Query by example. In *Proceedings of the AFIPS National Computer Conference 44* (Anaheim, Calif., May 1975), 431–438.

Received February 1992; revised May 1992; accepted November 1992