OPOSSUM: Desk-Top Schema Management through Customizable Visualization*

Eben M. Haber Yannis E. Ioannidis[†] Miron Livny

Department of Computer Sciences, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706
{haber,yannis,miron}@cs.wisc.edu

Abstract

Several recent trends have changed the usage and users of schemas beyond those of a database administrator's tool for describing database contents. Distribution of computing power to the desk-top and increasing needs for data management have broadened the base of schema users to include people who are not database experts. The advent of graphical user interfaces has brought schemas into use as templates for a variety of database operations such as query specification and browsing. Such changes demand advanced schema management techniques, primarily schema visualization, in order to support productive interaction between increasingly novice users and increasingly complex schemas. In this paper, we present OPOSSUM, a flexible, customizable, and extensible schema management system. Working within the established paradigm of schema editing through direct manipulation, OPOSSUM employs several novel techniques to offer the following capabilities: enhancement of schema visualizations with user-specific information; exploration of schemas through choice of visual representations; and creation of new visual representation styles when existing ones prove unsatisfactory. We discuss the architecture of the system and the methodology that guided its development, and illustrate its most important features through examples of how it has been used. OPOSSUM is operational and is in use by three groups of experimental scientists on the University of Wisconsin campus as a tool for experiment and database design.

1 Introduction

Traditionally, database schemas have been developed, examined, and modified by specialized database administrators using stylized data definition languages. The primary role of these schemas has been to define the information content of the database. Today, several trends are expanding this limited usage and user base. Distribution of computing power to the desk-top and increasing needs for data management have broadened the base of users who must interact with database schemas to include people who are not database experts. These non-expert[1] users need to design and access their databases, yet they have no desire to learn or use arcane database languages. Graphical User Interfaces (GUIs) provide more user-friendly access to schemas, and tend to broaden the role of schemas, using them as templates for other database operations such as querying and browsing. The growing number of non-expert users and increasing schema roles demand improved techniques and tools for schema management.

Schema visualization[2] is the key issue that arises in any attempt to provide improved tools for schema management. Improved schema visualization is particularly helpful for non-experts, but it aids all users interacting with the database or sharing information about the data among themselves. It is also a tool to aid in the management of large and complex schemas; diagrammatic presentations are generally easier to understand than text, and different visual styles can be used to highlight or filter out different schema information. Although existing database systems and prototypes support GUIs that present schemas in non-textual ways, these GUIs do not address all of the challenges posed by complex schemas and non-expert users. Most of them provide little flexibility, supporting only a single hard-wired visual style.

To better support schema visualization, we have developed a Desk-Top Schema Manager (DTSM) called **OPOSSUM** (**O**btaining **P**resentations **O**f **S**emantic **S**chemas **U**sing **M**etaphors). OPOSSUM follows the established GUI paradigm in supporting creation and modification of schemas through direct manipulation of schema visualizations. It is unique, however, in offering all of the following capabilities:

1. *Externally Defined Data Models and Visual Styles:* Arbitrary visual representation styles for schemas in any data model can be defined by the user in a declarative and extensible manner. Thus, the process of schema visualization is not realized through hard-wired code, but through external definitions provided by the user.

2. *Personal and Aesthetic Information:* Schema visualizations can be enhanced with user-specific annotations and

*Work supported in part by the National Science Foundation under Grant IRI-9224741.

[†]Additionally supported in part by the National Science Foundation under Grant IRI-9157368 (PYI Award) and by grants from DEC, IBM, HP, AT&T, and Informix.

Proceedings of the 21th VLDB Conference
Zurich, Switzerland, 1995

[1]We use the term "non-expert" to refer to users who are not database experts, regardless of their expertise in other areas.

[2]*Schema visualization* describes a representation of the information of a schema that a person can look at, or the process of creating such a representation from a schema. *Visual style* refers to the general form of a schema visualization.

aesthetic preferences. In other words, it is possible to visually capture information that is important to the user, yet not needed by the underlying database.

3. *Multiple Visual Styles:* Multiple visualizations can simultaneously exist for the same schema, so that a user can switch back and forth between them during schema exploration whenever a different visual style is desired.

4. *Mixed Visual Styles:* Mixed visualizations of a schema can be created, where different parts of it are presented using different visual styles, each style emphasizing something different.

In the remainder of this paper, we describe how OPOSSUM achieves these capabilities, discussing the architecture of OPOSSUM, the methodology that guided its development, and its most important features.

OPOSSUM has been developed as part of the user interface to ZOO, an ongoing effort to develop a Desk-Top Experiment Management System[3]. The goal of ZOO is to enable scientists to manage the entire life-cycle of their experimental studies via a single uniform desk-top interface. This is achieved by placing the conceptual/logical schema of the experiment data at the center of ZOO. Whether designing a study, invoking an experiment, querying the data, or analyzing query results, a graphical presentation of the schema is used to perform the activity; in essence, the schema captures the experimental study itself. Schemas in ZOO are based on the MOOSE object-oriented data model [37], which is similar to many other object-oriented or semantic models. (MOOSE schemas are used in many of the examples in this paper.) The needs of several experimental scientists collaborating in the development of ZOO have led us to realize the importance of improved and flexible schema management and have been our main guide in the design and implementation of OPOSSUM. OPOSSUM is currently used by some of these scientists for experiment and database design and their overall feedback has been very positive.

The rest of this paper is organized as follows. Section 2 provides a more thorough definition of a Desk-Top Schema Manager and its functionality. Section 3 outlines our formal approach to visualization. Section 4 describes OPOSSUM, the DTSM that we have implemented. Section 5 presents examples of how OPOSSUM supports visualization of different information. Section 6 describes related work, and Section 7 summarizes and offers conclusions.

2. Desk-Top Schema Managers

A Desk-Top Schema Manager may be generally defined as a tool to allow all kinds of users to view, comprehend, and manipulate schemas in all the roles schemas play. In order to understand the functionality of a DTSM in more detail, it is first necessary to consider how users will work with it.

[3]Naturally, the various modules of ZOO are named with acronyms like MOOSE, FOX, EMU, LOBSTER, and of course, OPOSSUM!

2.1 Modes of User Interaction

A user of a Desk-Top Schema Manager may interact visually with the system in one of the following three modes: *creation/modification* of visual styles, *creation/modification* of schema visualizations (with analogous effect on the underlying schemas), and *exploration* of schemas. These do not necessarily occur independently, but their separation helps in identifying their properties.

In *visual style creation and modification,* a user describes new ways to visualize schemas, or changes existing visual styles. These are then used during the other two modes of interaction for visual representation of schemas.

In *schema creation and modification,* a user constructs a visualization that captures the structure of the underlying data schema, as well as any personal information and aesthetic preferences. This is done through visual manipulations that, based on their effects, may be classified as follows: *editing,* which affects information in the underlying data schema, *enhancing,* which affects personal annotations but not the data schema, and *embellishing,* which affects only the aesthetics of the visualization.

In *schema exploration,* a user simply examines the schema. The most powerful tool to aid this task is dynamic, fine-grained flexibility of visual representation, allowing a schema to be visualized using arbitrary styles. Other useful tools include hard copy output (paper provides unsurpassed portability and resolution, and the ability to piece together visualizations larger than any computer screen), as well as traditional operations such as zooming and panning.

2.2 Functionality

Given the schema management needs discussed in Section 1 and the modes of interaction described above, a DTSM should offer the following capabilities with respect to functionality and flexibility, which are not found in current database GUIs:

• Supporting evolution in the fundamental visual styles. Over time, users may find existing visual styles unsatisfactory in highlighting the important aspects of their schemas. They should be allowed to change the styles to reflect this. Thus, the visual styles should not be hard-wired into the system.

• Allowing users to add personal information to schema visualizations beyond that which the database needs. Each (group of) user(s) may have a different view of the high-level organization of the schema and may need to express different concepts on top of the data model. To capture these varied concepts, a DTSM must appear to operate with potentially a different data model for each user.

• Accommodating the varying senses of aesthetics between different users. Users should be permitted to change the appearance of schema visualizations, and the system should maintain it to the greatest extent possible.

• Providing choice in visual representation. There are many ways to visualize schema information, and users may differ over which they find most intuitive. In addition, visual styles that work well for small schemas may not scale up, and different styles may highlight different aspects of a schema. A DTSM should provide a choice of different representations for the same schema; this choice should be available in any granularity, so that it can be made independently for individual pieces of the schema, and also dynamically, so that a user can easily flip between different representations during schema exploration.

2.3 Example

The last three desirable capabilities mentioned above are illustrated in the following example. Consider an object-oriented database storing information on various Companies, which consist of Departments, Employees, and Buildings, where all four of these entities are captured as classes. Figure 1 shows a visualization of this database schema in the form of a graph, where classes and relationships appear as nodes and edges, respectively.[4]

Assume that a user wishes to represent the personal information that Employees, Addresses, and Floors are more important than the other classes. A DTSM should allow users to express such information: in Figure 1, the important classes are given a different background color.

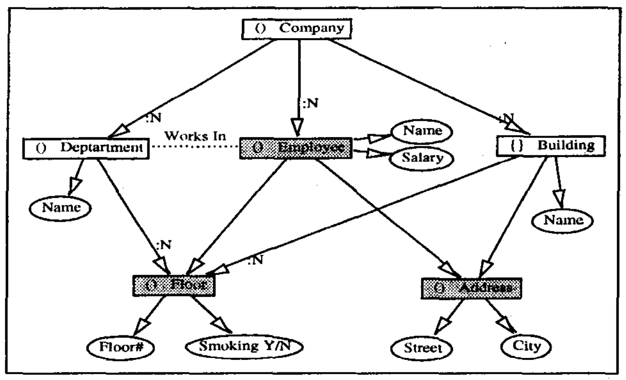

Figure 1: A simple schema using a graph style.

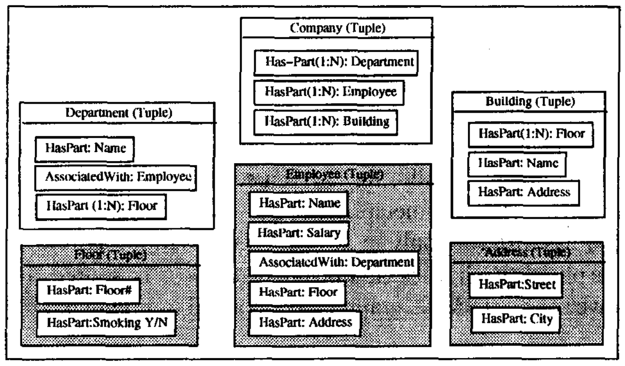

Figure 2: The same schema using a containment style.

Figure 2 shows a different visualization of the above schema. This time a containment representation is used, where each class appears as a box containing a set of smaller boxes, one for each relationship it has to another class. Clearly, both visualizations are equivalent in capturing the database schema and the personal information. It is also clear that there are trade-offs in the quality of the two visual representations. Figure 1 is better at showing global structure and transitive relationships. On the other hand, it is worse at showing immediate relationships because there may be very long edges and many edge crossings. Figure 2 brings out local structure better in such cases.

Note that the user may also affect the aesthetics of a visualization: in Figure 1, the shared classes, Floor and Address, are moved below the others to make it easier to identify them and to reduce spatial density of edge crossings. The specific placement reflects no schema or personal information, and is therefore not captured in Figure 2 where such layout is unnecessary.

3 Visualization Methodology

A database system is able to manage many different data arrangements using a declarative description of the data, i.e., the schema, and generic components for storing and manipulating the data. To manage schemas as described in Section 2, a DTSM must take a similar approach: use declarative description of schemas and their visualizations, and generic methods to manipulate them. These cannot be developed in an ad hoc manner; the design and implementation of a customizable and extensible schema manager must be based on a formal understanding of the interplay between schemas and their visual presentations. In this section, we briefly discuss the formalism of the visualization process underlying the design and development of OPOSSUM, which is presented in detail elsewhere [17].

3.1 Overview

As is well known, every database schema is an instance of some data model, and a data model essentially defines a set of valid schemas for that model. In exactly the same way, we introduce the notions of a *visual model* and a *visual schema.* A visual schema is just a visualization rendered on a screen, drawn on paper, or otherwise expressed. It is always an instance of a visual model. For example, a specific graph drawn on the screen with its nodes and edges in specific locations is a visual schema, and the set of all possible such graph drawings is defined by a visual model that specifies their general look.

The primary contribution of our framework is that it brings visualizations and visual models to the same level as traditional "data" schemas and data models, demonstrating duality between the two. Hence, the basic techniques that are applied to translate schemas between different data models [24] can also be applied between visual and data models. Such a translation establishes a correspondence between a schema and its visualization (visual schema). In particular, we introduce the notion of a *visual metaphor*, which is defined as a mapping from (the elements of) a visual model to (the elements of) a data model. Through this mapping, one is able to

[4]All figures in this paper showing example schemas have been created as PostScript output from OPOSSUM.

assign meaning to a visualization (visual schema) in terms of the underlying data schema, e.g., a rectangle in an E-R graph indicates an entity set in the underlying E-R schema. In our framework, both the visual and data models and the visual metaphor may be specified declaratively: the models as instances of a meta-model, and the metaphor as a set of pairs of elements from these models. Thus, schema visualization does not have to be realized through hard-wired code, but can be established externally in an extensible way.

With the above framework, it becomes easy to define different ways to visualize schemas of any given data model. All that is required is the specification of different visual models and corresponding visual metaphors to the same data model. Moreover, given multiple metaphors, one may combine them, which permits schemas to be displayed with different parts visualized through different metaphors.

Another benefit of the above formalism stems from the fact that visual models usually offer many more degrees of freedom in visual representation than is necessary to capture the characteristics of a data model. For example, the locations of the nodes of a graph visualization carry no meaning with respect to the graph itself, so no matter how a graph is laid out, it still has the same meaning. Thus, even when using a single visual metaphor, one data schema may be represented by many visual schemas. This plurality of representation can be used to capture personal and aesthetic information, discussed earlier, both of which do not reflect information in the data schema. Personal information is encapsulated in a superset of the data model called the *personal model.* It is visualized through an enhanced visual metaphor from the visual model to this personal model. Aesthetic information is captured in the parts of the visual model not mapped by the metaphor. Figure 3 shows the relationships between data, personal, and visual models, along with the original and enhanced visual metaphors.

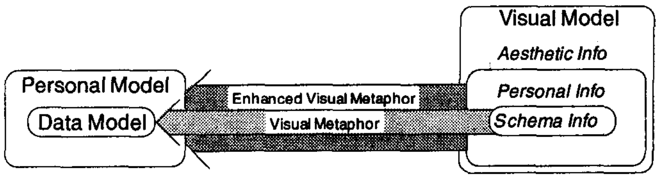

Figure 3: The Various Models and Metaphors.

3.2 A Brief Example

A complete discussion and examples of model and metaphor definitions appears elsewhere [17]. The crux of the formalism is that models are described in terms of *primitive types*[5], *attributes* of those primitives, and possible *values* of the attributes. Models also include *constraints* that must be satisfied by every schema in the model. Metaphors are defined as mappings between the parts of the models. A simple example of such a mapping is demonstrated in Figure 4. The visual model includes a primitive type that appears as a rectangle with two labels inside. The data model includes the primitive type "Class", which has attributes Name and Kind, and the Kind attribute has a set of allowable values. The metaphor establishes correspondences between the two primitive types, their attributes, and possible values, giving meaning to visual schemas.

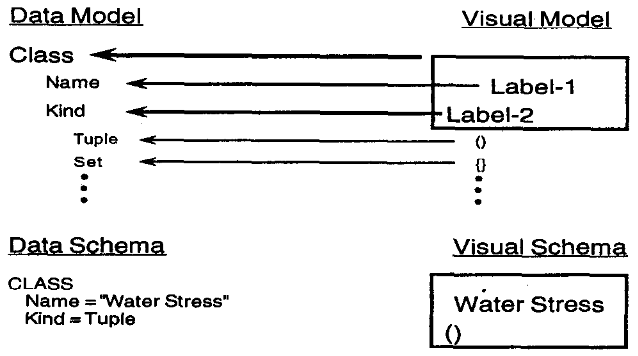

Figure 4: Metaphor example, with corresponding schema instantiations.

3.3 Impact of Visualization Methodology on DTSM Flexibility

The visualization methodology described above establishes the foundation upon which a DTSM like OPOSSUM may be built to provide all of the necessary capabilities listed in Section 2.2. Declarative definition of models and metaphors allows users to introduce them in an evolutionary way based on their changing needs (first capability). Customized personal models allow users to enhance schema visualizations with user-specific annotations (second capability), while any unused elements of a visual model allow the user to aesthetically improve schema visualizations (third capability). Finally, support for multiple and mixed visual models allows users to view the same information in multiple ways (fourth capability).

All users of a DTSM will share the same data model, but each one may have a distinct personal model. In addition, there may be several visual models with as many or more visual metaphors mapping between them and the various personal models. Finally, for a given schema, personal model, visual model, and metaphor, there may be multiple visual schemas differing only in aesthetic information. This extensive plurality of representation opportunities, the source of the power of a DTSM, is graphically depicted in Figure 5.

4 Design and Implementation of OPOSSUM

4.1 System Architecture and Implementation

Figure 6 shows the overall architecture of OPOSSUM, which consists of three main components: the *schema manager*, the *meta-creator*, and the *storage manager*. The schema manager supports schema creation/modification and exploration (Section 2.1) and can be used by non-experts. It consists of

[5]To avoid any confusion, we should note that we use 'primitive type' as a shorter version of 'type of primitive', indicating a fundamental modeling element/construct, and not as a qualified noun phrase, indicating a type that is basic/simple.

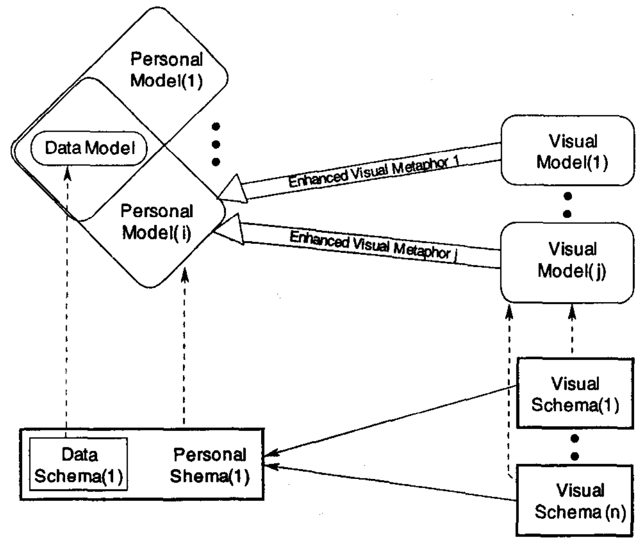

Figure 5: Plurality of representation opportunities.

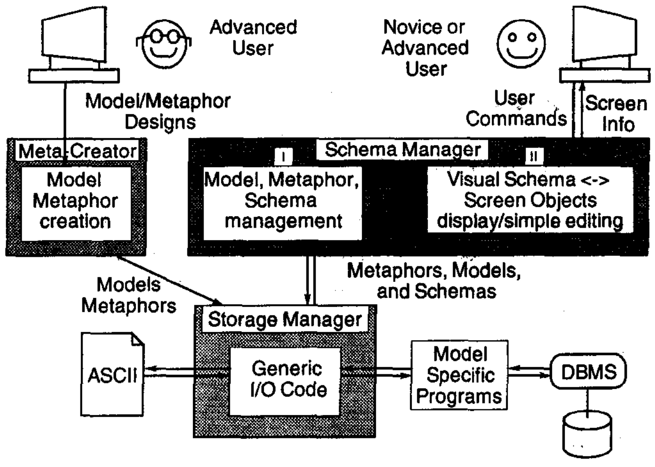

Figure 6: The Architecture of the OPOSSUM DTSM.

two modules. Module I accepts as input descriptions of data models, visual models, and metaphors. Just as a database system can manage data given a schema, Module I can manage schemas given this model and metaphor information. At runtime, models, metaphors and schemas are represented as instances of C++ classes, and all operations on them are managed by methods. Module II uses InterViews [23] to convert visual schemas to screen objects. Given any specific visual model and metaphor, this code provides direct manipulation tools for creating, modifying, exploring, and printing schema visualizations.

The second component, the meta-creator, supports creation/modification of models and metaphors and is intended to be used by relatively knowledgeable people. Ideally, the schema manager should be used for meta-creation through meta-models and a visual meta-metaphor that capture the structure of models and metaphors.

Finally, the third component provides storage and input/output to the first two components for models, metaphors, and schemas. Through generic code, it offers storage to ASCII files on disk. In addition, it can communicate schema information with external programs for further processing. When a model is defined for OPOSSUM, the definition includes a list of external programs that may be invoked by the user to process schemas of that model. Examples of such processing include sending the schema to the database, applying an automatic layout program to a visual schema, or producing statistics about a schema. These external programs are necessary because schemas in different models must be treated differently, e.g., a relational schema will be sent to a different database than an object-oriented schema. We should also mention that the Storage Manager separates the parts of a schema that are meaningful to the data model from those that are not, permitting many visual schemas in many visual models based on the same underlying data schema.

4.2 A 'Demo' of OPOSSUM

In this subsection, we describe how OPOSSUM operates and how users interact with it. Assume, for example, that a user wants to use OPOSSUM to manage MOOSE database schemas and prefers to view them as graphs. The user (or possibly a more knowledgeable designer) must first use the meta-creator to define the MOOSE data model, a directed graph visual model (like that seen in Figure 1), a metaphor between the two, and any external functions needed to communicate with the database. Once these models and metaphor are in place, the user can invoke the schema manager. Using the model descriptions, the system customizes itself to work with schemas of those models. An example of this customization is shown in the screen dump of Figure 7.

Near the top of the window, the "File" and "Edit" menus provide several generic capabilities for file operations (e.g., printing a schema or saving it to a file), as well as visual manipulation (e.g., cut, paste, zooming, panning, etc.). The "Misc" menu contains an option for each of the model-defined external schema processing programs mentioned in section 4.1 (e.g., it might include a program for sending the schema to a database). On the left, there is a column of buttons, each button defining a mode for direct manipulation of screen objects. There are generic buttons (e.g., to select, move, or stretch) and buttons specific to the visual model. In particular, there is a button for each primitive type of the visual model which, when selected, permits the generation of corresponding primitive instances. When these types are mapped by the metaphor to primitive types in the Data Model, the buttons are labeled with the Data Model type names. In Figure 7, there are two model-specific buttons, one for the Class primitive type and one for the Relationship primitive type. By selecting the Class button, every subsequent mouse click on the drawing area generates a graph Node, which is mapped by the metaphor to a Class in the underlying data schema. By selecting the Relationship button, a subsequent mouse click starts the process

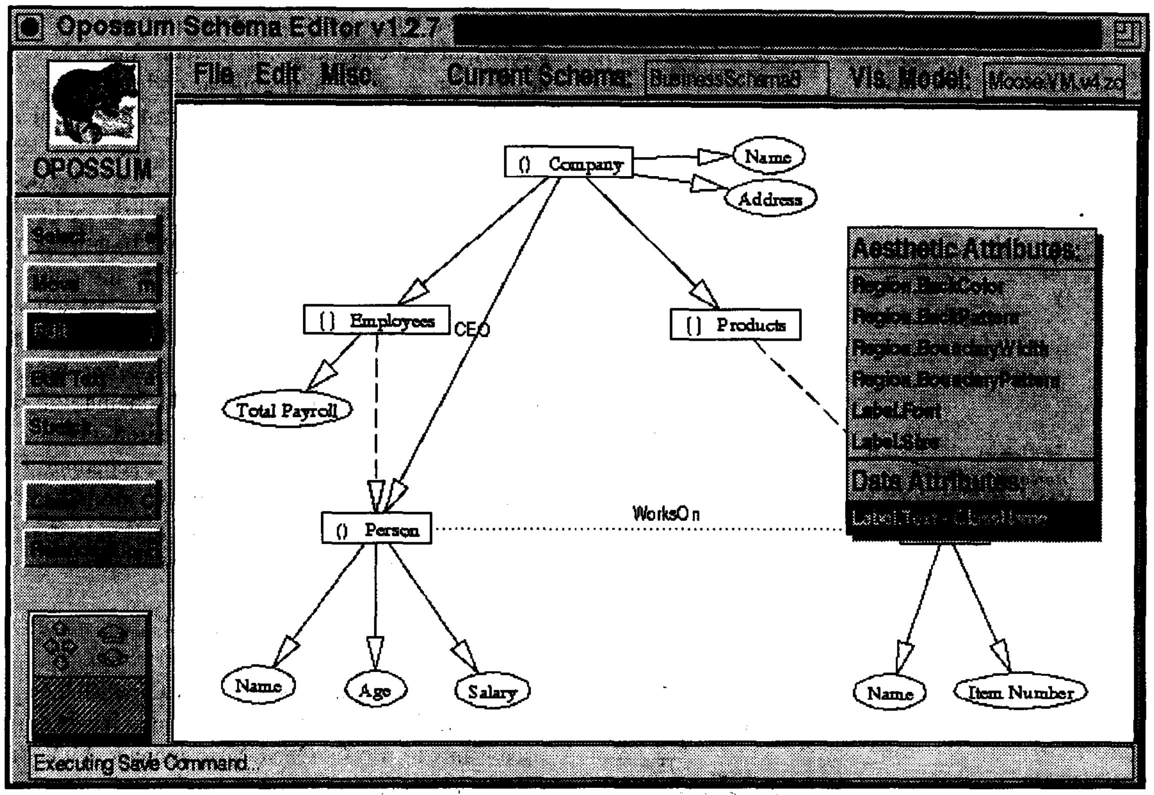

Figure 7: The OPOSSUM Interface, using the Edit Tool.

for the generation of an Edge, in which the system asks the user to specify the two Nodes connected by the Edge.

The difference in system behavior for Node or Edge creation is not achieved by any specialized code written explicitly for the graph/MOOSE models. It is generic code that takes into account information that exists in the models. For example, the model specifies that the Edge primitive type has two attributes of type Node whose values may not be null. As a result, the system knows that it must ask the user to indicate two Nodes whenever an Edge is created. Thus, OPOSSUM uses the declarative definition of models and metaphors to capture not only appearance but behavior of visualizations as well.

In addition to creating instances of the various primitive types of the visual model, a user may also want to change the values of the attributes of some of these instances. Spatial (location) and textual attributes can be modified directly through the 'Edit-Text', 'Move', and 'Stretch' buttons. All other attributes are modified using the 'Edit' button. When 'Edit' is selected, a mouse click on a visual primitive instance results in the appearance of a pop-up menu with all the primitive's attributes that are relevant to the area of the click. By choosing one of the menu entries, a dialog box appears through which one may specify a value for the corresponding attribute. An example of the process is shown in Figure 7, where the system is in 'Edit' mode, the menu with attributes of a Node primitive has appeared (the Node being barely visible below the menu), and the text of the central label of the Node has been chosen for modification. Note that the attributes are distinguished into those that are mapped to the underlying data model (e.g., Label.text mapped to ClassName) and those that simply affect aesthetics. If a personal model had been specified as well, then the attributes in the pop-up menu would have been partitioned into three categories.

4.3 Implementation Status

The current implementation of OPOSSUM consists of about 31K lines of C++. It provides the entire spectrum of functionality described above, with the exception of a few issues on which work is still in progress (but close to completion). Specifically, the storage manager cannot yet store personal and aesthetic information in an underlying database; this is currently stored in ASCII files only. In addition, the meta-creator does not use the capabilities of the schema manager, instead requiring models and metaphors to be specified as expressions in a formal grammar. Work is under way to allow meta-editing of models; a complete meta-model has been defined and is undergoing testing.

Due to the diverse hardware environments of our collaborating scientists in ZOO, we have strived to maintain portability. Thus, OPOSSUM currently runs on several Unix workstation platforms, including DecStations, HP Snakes, Sun SparcStations, and SGI Indigos. As for underlying database systems, we have developed code to communicate schemas to both Informix and the MOOSE-based database system used in ZOO.

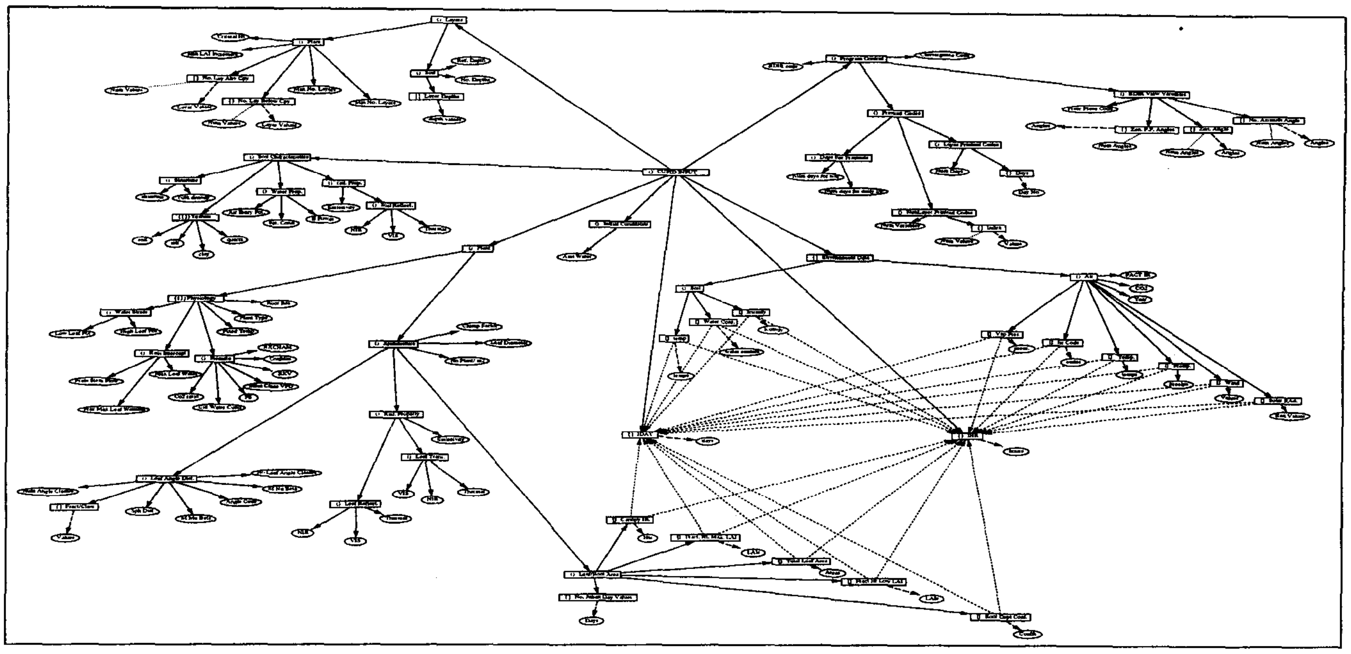

Figure 8: The input of the Cupid simulation model as a MOOSE schema, created using OPOSSUM.

5 Case Studies: OPOSSUM in Action

At present, we have used OPOSSUM with the Relational, MOOSE, and E-R data models, in each case using a variety of visual models and metaphors. The visual model of graphs has proved to be meaningful for all three data models, especially enriched with primitives for capturing abstraction (described in Section 5.2). We have also used OPOSSUM beyond schema visualization, as the core of query and object visualization tools for ZOO. This section describes how OPOSSUM can be used for different kinds of schemas and other data. Special attention is given to how OPOSSUM can be used to support the complex functionality of visual abstraction.

OPOSSUM is currently used by scientists in Genetics, Biochemistry, and Soil Sciences to design schemas for their databases or experiments. The feedback in all cases has been very positive, especially with respect to OPOSSUM's ease of use: people are able to learn the system and then organize and layout large schemas with hundreds of classes within a few hours. This informal feedback from our users has proved very valuable; as our user base expands we intend to do more organized usability assessments to better tailor the operation of OPOSSUM to its intended users.

5.1 Cupid, MOOSE, and a Graph Visual Model

A group of soil scientists have used OPOSSUM to layout a MOOSE schema describing the parameters of Cupid, a FORTRAN simulation model of plant growth [19, 20]. The input part of the Cupid schema alone has 159 classes (see Figure 8). The visual model is that of a directed graph, where Nodes and Edges represent classes and relationships, respectively, Node shape and a text field represent class kind, and the Edge's line pattern indicates the type of the relationship. The tool has brought substantial improvements in the scientists' work. Before OPOSSUM, the soil scientists' only reference to Cupid was the input data file to the Fortran program, which had grown increasingly fractured and confusing over the years. They now use the visual schema of Figure 8 as their reference in thinking about the model, planning experiments, and explaining experiments to other scientists.

5.2 Grouping: a Means of Visual Abstraction

Large schemas present a major problem for schema managers; it is often difficult to convey both large-scale structure and fine details. In this section, we focus on abstraction, a well-known approach for managing large graphs. We demonstrate how OPOSSUM can be customized to support it, illustrating in the process several features of the system. The ease with which this is done provides a case study of the flexibility and extensibility of our tool.

5.2.1 Problem

During schema exploration, a user often wants to see both the high-level structure of schemas and their fine details. Due to size and resolution limitations of common displays, however, any attempt to view a very large schema in its entirety will result in text and possibly shapes that are too small to decipher, obtaining neither the overall structure nor the details. In particular, when large schemas are displayed as graphs, the amount of Nodes, Edges, and Edge crossings may limit many of the advantages of such a metaphor (e.g., Figure 8).

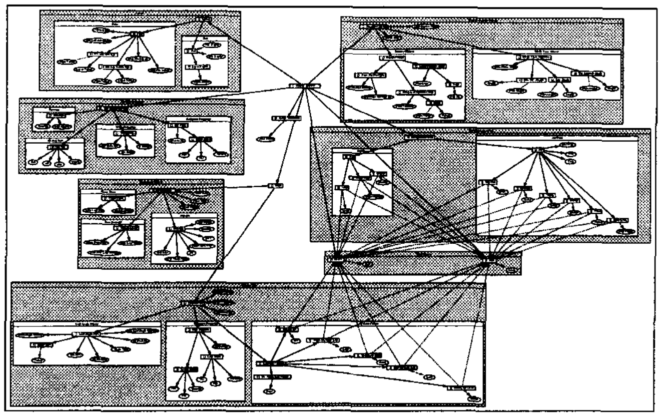

Figure 9: The CUPID input schema, Grouped.

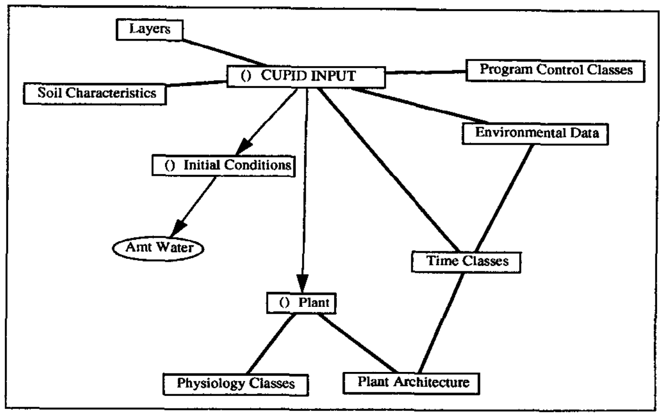

Figure 10: Cupid, abstracted.

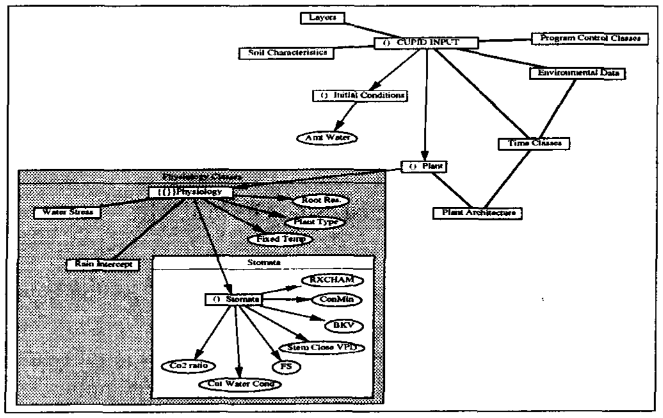

Figure 11: Cupid, partially abstracted.

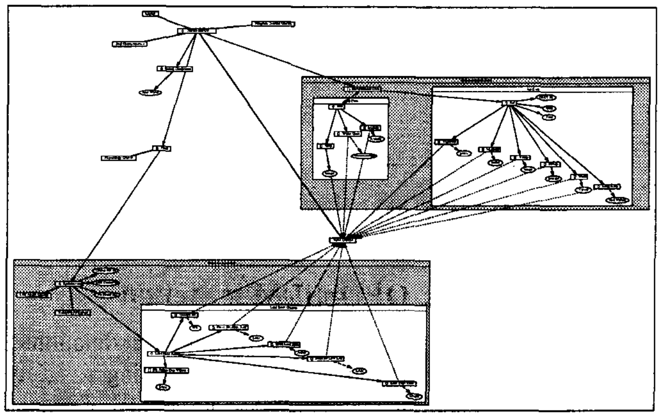

Figure 12: Cupid, partially abstracted another way.

One established approach to this problem is *Grouping*, i.e., to recursively permit subgraphs of a visual schema to form a single abstract unit, a *Group*. A Group may be *collapsed*, i.e., displayed as a single Node with its constituent subgraph invisible, or *expanded*, i.e., displayed with the entire subgraph visible. Grouping has been proposed and implemented in the past [3, 4, 36], although not always on the scale described here. Unlike zooming, which shows the same detail at a smaller scale, Grouping shows less detail at the same scale. By drawing only high-level Groups, the overall structure of a complex schema can be clearly shown. By expanding the subgraphs of selected Groups, the details of these subgraphs can be clearly seen as well. As an example, Figure 9 presents the same schema as in Figure 8, only partitioned into rectangular Groups. Figure 10 shows the schema at its highest level of abstraction, with only eleven Groups and Nodes. Finally, Figures 11 and 12 show the internal structure of one and two top-level Groups, respectively, allowing study of their details.

5.2.2 Grouping as an Example of a Mixed Metaphor

Grouping has already been seen in several systems, e.g., QBD* [3], SUPER [4], and others [36]. Below we describe how it is realized in OPOSSUM, with little effort, as an example of a mixed metaphor.

Working with the typical graph visual model described above, we introduce a new primitive type called a Group. Both the Node and Group primitives are given a ParentGroup attribute, which takes values among the instances of Group in any schema. Also, a Group has an expanded and a collapsed visual representation, the choice between which is controlled by the value of a Representation attribute, which makes the resulting metaphor mixed. In the expanded representation, a Group appears as a large box with a label at the top. In the collapsed representation, a Group appears the same size as a Node.

In addition to the above, the model has several constraints to ensure the visual integrity of a schema with Groups, among which the most important are the following:

1. In the expanded representation of a Group, if the footprint of a Node or Group c is within the footprint of a Group p, then c is a member of p.

2. The footprint of a Group is no smaller than the footprint of all its members.

3. In the expanded representation of a Group, all its members are visible.

4. In the collapsed representation of a Group, all its members are invisible.

5. When a Group's expanded location changes, the locations of its members change by the same amount.

Given the multiple representations of Groups and the potential invisibility of Group members, visibility of Edges becomes an issue as well. To address this, the visual model is enhanced with a GroupEdge primitive type. GroupEdges connect collapsed Groups when those Groups have members that are connected, as for example, in Figure 10. Constraints similar to the above are defined to handle the creation of GroupEdges and the visibility of GroupEdges and regular Edges.

Switching back and forth between two visualizations can cause challenging layout problems. When a Group collapses, e.g., from Figure 11 to 10, the entire graph should move closer together so that it does not become too sparse. Likewise, when a Group expands, the rest of the graph should move away to make space for the Group's expanded representation; otherwise, there may be Node/Group overlaps with undesirable semantics or aesthetics. Managing such layout side effects caused by changes in the Group footprints is a key challenge in implementing Grouping without using special code just for this problem. We address this challenge using a generic mechanism that is built in to OPOSSUM to handle differently-sized alternative representations for the same primitive type. Briefly, the system maintains separate 'base' locations for each possible representation, and whenever layout problems occur (e.g., primitives overlapping or space being freed) each neighboring primitive moves along the line between its current location and the appropriate base location until the problem is solved.

From all the above, it should become clear that OPOSSUM can be customized to provide complex functionality like Grouping with no specialized effort, simply through the appropriate definitions of models and metaphors. Not only is the desired appearance and behavior of Groups obtained, but most tasks involved with using Groups can be accomplished with very few actions on the part of the user. Grouping is a testament to the flexibility and extensibility of our approach to visualization.

5.3 A Relational Schema as a Graph

Another group of scientists, working on NMR research, have used OPOSSUM to develop a very large Relational schema (over 250 tables and more than 2000 attributes) for describing their experiments. They have used another graph-based visual metaphor, with rectangular relation Nodes, oval attribute Nodes, and vertical placement representing relation-attribute relationships. In addition, primary keys are captured by a darker color and foreign keys relationships are represented as Edges. An example of a small piece of this schemas is shown in Figure 13. The NMR researchers have found OPOSSUM very useful in helping them design their large schema since it enables them to see its overall structure and easily navigate around it.

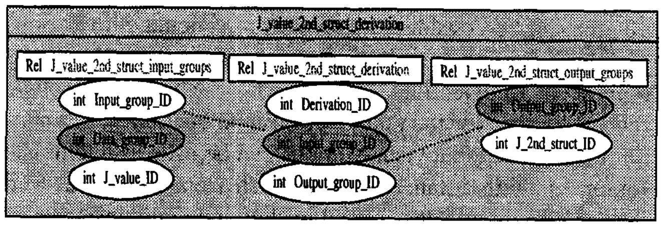

Figure 13: Part of the NMR Relational Schema.

```
Roughness  – P
    Has–Part  From   Structure
Sand  – P
    Has–Part  From   Texture
Sat . Cond.  – P
    Has–Part  From   Water Prop.
Soil  – ()
    Has–Part  To   integer   As   No. Depths
    Has–Part  To   real   As   Ref. Depth
    Has–Part  To   Layer Depths
    Has–Part  From   Layers
Soil  – ()
    Has–Part  To   Icumdy
    Has–Part  To   Water Cont.
    Has–Part  To   temp
    Has–Part  From   Environment Data
Soil Characteristics  – ()
    Has–Part  To   Texture
    Has–Part  To   rad. Prop.
    Has–Part  To   Structure
    Has–Part  To   Water Prop.
    Has–Part  From   CUPID INPUT
Soil Reflect.  – ()
    Has–Part  To   NIR   As   NIR
    Has–Part  To   VIS   As   VIS
    Has–Part  To   THERMAL   As   Thermal
    Has–Part  From   rad. Prop.
Solar RAd.  – []
    Collection–Of To   ObjectName   As   Rad Values
    Collection–Indexed–By To   IHR
    Collection–Indexed–By To   IDAY
    Has–Part  From   Air
```

Figure 14: Part of the Cupid schema, as a textual list.

5.4 Other Visual Models and Other Data

While two of the previous examples used visual models based on directed graphs, Opossum is capable of supporting other types of model. One example is containment for expressing parent-child relationships, something used in Grouping. Another example is textual lists, whose linear nature allows them to be sorted. Figure 14 shows a portion of the Cupid schema visualized using a textual list visual model, sorted by class name. This visual model and metaphor is basically a 2-level outline, representing MOOSE classes as MainPoints, and relationships as SubPoints. MainPoints are pairs of labels with a fixed X location and free Y location. SubPoints include labels beneath and to the right of both the source and the destination of the Relationship. Constraints ensure placement of the SubPoints below the MainPoints that they connect.

The variety of visual models for schemas also prove useful for queries and data. Directed graphs work well for queries; queries can be considered as pieces of schema with various qualifications and operators attached. To visualize them, one only needs to extend a visual model for schemas to include representations for the qualifications and operators. In a similar manner, schema visualizations can be used as a template for representing instances of data. Such a template works well for browsing, but is less useful in showing many instances at once (unless one has a very good automatic layout program). For many instances, one can use a textual list, or design a visual model especially for the data, e.g., forming an x-y chart by mapping attributes of the data to position in the plane, color and shape of chart ticks, etc. All these examples demonstrate that OPOSSUM has the flexibility to be used as a visualization and editing tool for schemas and many other kinds of information.

6 Related Work

As mentioned earlier, the main contribution of OPOSSUM is its comprehensive support for (a) declarative definitions of data and visual models and metaphors in a way that is explicit, extensible, and external[6] to the database system, (b) multiple and mixed metaphors for visualization, and (c) enhancements of schema visualizations with personal information and aesthetic preferences. To the best of our knowledge, support for (a) alone makes OPOSSUM the only implemented database schema manager whose appearance and behavior can be declaratively customized by the user. OPOSSUM is also unique in supporting (c), personal information.

Some of the ideas in (a) and (b) can be found in other systems. In this section, we present an overview of the spectrum of related (implemented or paper) systems and pinpoint their differences from OPOSSUM. We divide systems into four categories: schema visualization and manipulation tools (schema managers); query or data visualization tools that use schemas as templates; query or data visualization tools unrelated to schemas; and user interface tools, which generate separate stand-alone visualization tools. For the last two categories, the notion of metaphor is generalized to capture a mapping from a visual model to arbitrary internal domains, not simply data models. Table 1 summarizes the overall comparison that follows, indicating for each system surveyed whether or not it supports any of the features in (a) and (b). When several systems are presented as a group, the characterization of the group in parenthesis is used as an identifier. The abbreviations VM and DM are used to indicate 'visual model' and 'data model', respectively.

6.1 Schema Tools

Most database systems have a fixed, hard-coded visual model and metaphor. Although textual representations are most common, there are other frequently used visual metaphors including tables, graphs (such as E-R diagrams), and icons.

[6] We view all three 'ex' properties as equivalent.

There are several systems that provide multiple metaphors for visualizing schemas, including ISIS-V [13], Sidereus [2], SUPER [4], IDDS [25], and WINONA [29]. They provide metaphor choices from among those listed above, as well as more unusual metaphors such as 3-D displays and natural language dialogs with the user. While these systems provide a certain amount of choice, the alternatives are pre-defined and hard-coded. Alternatively, England and Cooper [15] suggest visualizing schemas using a data visualization tool that permits user-specified mappings of information to icons in a 2-D space. This system's flexibility, however, stops at metaphor definition; the data and visual models are hard-coded.

6.2 Schema Template Tools

The idea of using schema visualizations as templates for query specification or data presentation goes all the way back to QBE [39] and GUIDE [38] and has been routinely incorporated in many systems and prototypes.

Although not yet implemented, conceptually the most advanced of these systems (and, in fact, the one that comes closest to OPOSSUM) is DOODLE [12]. It offers a visual declarative language that allows users to define with pictures OODBMS queries or visualizations of database objects. DOODLE defines the visual model and metaphor together, and the former is not separable from the latter. Similarities between OPOSSUM and DOODLE include the declarative specification of models and metaphors, the ability to visually specify models and metaphors, and the ability to have multiple visual languages, i.e., multiple metaphors. The disadvantages of DOODLE over OPOSSUM include its inability to support mixed metaphors, and the fact that DOODLE has not yet been implemented. The advantages of DOODLE include its ability to visually specify visual constraints and data visualizations and the fact that the expressive power of its language is well studied.

On a simpler level, the Relational report generators and forms tools found in most commercial database systems (and some research prototypes [21, 31]) allow users to specify metaphors for schema visualizations to be used for query and data presentation. These metaphors are usually defined either interactively or with a declarative language, and many of them may co-exist (i.e., one may use multiple forms or report definitions for the same data). The data and visual models in such systems are mainly textual, however, and are not extensible.

Finally, there is a multitude of schema template visualization tools in Object-Oriented database systems, which use class methods to procedurally define object visualizations (ADAM [27], DEED [28], ODDS [16], OdeView [1], and others [32]). They all present the user with a single, mixed metaphor, where each object can be seen in one of many ways. A limitation of these systems is that there is no mechanism to deal with co-existing contradictory or ambiguous visualizations.

Other Work	Model Def.		Metaphor Def.		Metaphor Choice	
	Declarative	Extensible	Declarative	Extensible	Multiple	Mixed
Schema Tools						
OPOSSUM	√	√	√	√	√	√
(Most systems)						
ISIS-V [13], Sidereus [2], SUPER [4], IDDS [25], WINONA [29]					√	
[15]			√	√		
Schema Template Tools						
DOODLE [12]	√	√	√	√	√	
(Report Generators) & (Form Systems)			√	√	√	
(OODBMS Visual Tools)		VM only		√		√
Query/Data Tools						
Hy+ [14]						√
Q-PIT [6], PRIMA [33]					√	
[9]					√	√
Tool Generators						
UIDE [34]	√	√	partly	√		
HUMANOID [35]	VM only	VM only				
Binnacle [5] & (Most OODBMS Tool Generators)		VM only		√		√
R1 [18]				√		√
[11]	DM only	DM only				
[8]						

Table 1: Comparison of OPOSSUM with other related systems.

6.3 Query/Data Visualization Tools

There are a few systems that offer choice in metaphors for query and/or data visualization, though their metaphors and models are not extensible. For data visualization, Hy+ [14] deals with graph-type data and provides a hard-wired mixed metaphor to accommodate a grouping abstraction similar to that of Section 5.2. Two other such systems are Q-PIT [6], which proposes allowing user-defined mappings of tuples and attributes to visual primitives and attributes in a 3-D space (although there is no concrete suggestion on how this mapping would be accomplished), and PRIMA [33], which allows query results to be viewed at four levels of abstraction. For query visualization, Catarci et. al. [9] describe a powerful framework that uses both mixed and multiple metaphors; queries may be specified in a diagrammatic, form, icon, or hybrid fashion, with procedural transformations between them.

6.4 Tool Generators

In our discussion of tool generators (user interface tools), we view them as a means to define models and metaphors for the tools that they generate, and focus on the properties of their definitional power. We distinguish two categories of tool generators, those unrelated and those related to database systems. Most general user interface tools specify visual models procedurally, though a few, such as HUMANOID [35] and UIDE [34], allow description of visual models through expressions in formal modeling languages. HUMANOID does not allow any description of a data domain, and all data is encapsulated in the visual primitive types. UIDE permits declarative definitions of the data and visual domains, though its 'metaphor' is defined as correspondences between operations on visual model objects (such as a click of the mouse) and actions on data model object (such as incrementing a value).

There are several user interface tools designed to support a specific database system and hence a single data model. Binnacle [5] and R1 [18] are two similar systems for the nested Relational data model, which allow procedural specified metaphors, with interface behavior determined through an extensible description of a finite state automaton. Binnacle allows extensible descriptions of the visual model as well. There are also several user interface tools for object-oriented data models (DUET [26], FaceKit [22], O2Look/ToonMaker [7], Picasso [30]), which allow the procedural definition of a single, mixed metaphor, and associated visual model. There are notable exceptions, however, which can deal with several object-oriented and semantic data models by capturing their common characteristics [11], or deal with a specific object-oriented model and uses a single, hard-coded visual model and metaphor to automatically generate form and menu-based interfaces from the database schema [8].

7 Summary

In the natural world, the Opossum is an animal known for its acting ability; the expression "playing 'possum" originates from its inclination to feign death when startled or alarmed. The OPOSSUM system is also an actor: given the script of model and metaphor definitions, it can assume the role of a direct manipulation editor for schemas from virtually any data model visualized in diverse ways. Not only can OPOSSUM play many different roles, it can also combine them into a single, mixed role.

OPOSSUM is part of the user interface module of an ongoing effort to develop a desk-top experiment management system called ZOO. Our future work includes the completion of the implementation of the meta-creator through meta-modelling; expansion of the methodology used in OPOSSUM so that the system can be used for complete query and data visualization within ZOO; a comprehensive evaluation of usability of the system; and a study of constraint evaluation techniques that could help expand the capabilities of the system.

References

[1] R. Agrawal, N.H. Gehani, and J. Srinivasan. OdeView: The Graphical Interface to Ode. *SIGMOD Record*, pp. 34–43, 1990.

[2] A. Albano, L. Alfo, S. Coluccini, and R. Orsini. An Overview of Sidereus, A Graphical Database Schema Editor for Galileo. *Int. Conf. on EDBT*, 1988.

[3] M. Angelaccio, T. Catarci, and G. Santucci. QDB*: A Graphical Query Language with Recursion. *IEEE Transactions on Software Engineering*, pp. 1150–1163, October 1990.

[4] A. Auddino, E. Amiel, Y. Dennebouy, Y. Dupont, E. Fontana, S. Spaccapietra, and Z. Tari. Database Visual Environments Based on Advanced Data Models. In *AVI'92: Proc. Work. on Advanced Visual Interfaces*, pp. 156–172, 1994.

[5] J. R. Bedell and F. J. Maryanski. *Extensible Semantic Automata for Modular Design Systems*, pp. 43–56. Elsevier Science Publishers B.V., 1989.

[6] S. Benford and J. Mariani. Virtual Environments for Data Sharing and Visualization - Populated Information Terrains. In *Proc. 2nd Int. Work. on User Interfaces to Databases*, pp. 168–184, 1994.

[7] P. Borras, J.C. Mamou, D. Plateau, B. Poyet, and D. Tallot. Building user interface for database applications. *SIGMOD Record*, pp. 32–38, March 1992.

[8] R. Carapuca, A. Serrano, and J. Farinha. Automatic Derivation of Graphic Human-Machine Inferfaces for Databases. In *Proc. 1st Int. Work. on User Interfaces to Databases*, pp. 176–192, 1992.

[9] T. Catarci, S.K.Chang, and G. Santucci. Query Representation and Management in a Multiparadigmatic Visual Query Environment. *Journal of Intelligent Information Systems*, 3(3/4):299–330, July 1994.

[10] M. Consens and A.O. Mendelzon. Hy+: A Hygraph-based Query and Visualization System. Technical Report CSRS-285, University of Toronto, June 1993.

[11] R. Cooper. Configurable Data Modelling Systems. In *Proc. 9th Conf. on the Entity Relationship Approach*, pp. 57–73, 1990.

[12] Isabel Cruz. User-Defined Visual Languages for Querying Data. Technical Report CS-93-58, Brown University, December 1993.

[13] J. Davison and S. Zdonik. A Visual Interface for a Database with Version Management. *ACM Transactions on Office Information Systems*, 4(3):226–256, July 1986.

[14] F. Ch. Eigler. Hy+ User's Manual. Technical Report CSRS-285, University of Toronto, June 1993.

[15] D. England and R. Cooper. Reconfigurable User Interfaces for Databases. In *Proc. 1st Int. Work. on User Interfaces to Databases*, pp. 338–352, 1992.

[16] B. Flynn and D. Maier. Supporting Display Generation for Complex Database Objects. *SIGMOD Record*, pp. 18–24, March 1992.

[17] E. M. Haber, Y. Ioannidis, and M. Livny. Foundations of Visual Metaphors for Schema Display. *Journal of Intelligent Information Systems*, 3(3/4):263–298, July 1994.

[18] G. Houben and J. Paredaens. *A Graphical Interface Formalism: Specifying Nested Relational Databases*, pp. 257–276. Elsevier Science Publishers B.V., 1989.

[19] Y. Ioannidis and M. Livny. Conceptual Schemas: Multi-Faceted Tools for Desktop Scientific Experiment Management. *Int. Journal of Intelligent and Cooperative Information Systems*, 1(3), December 1992.

[20] Y. Ioannidis, M. Livny, and E. M. Haber. Graphical User Interfaces for the Management of Scientific Experiments and Data. *SIGMOD Record*, pp. 47–53, March 1992.

[21] R. King and M. Novak. Freeform: A User-Adaptable Form Management System. In *Proc. Int. Conf. on VLDB*, pp. 331–339, 1987.

[22] R. King and M. Novak. FaceKit: A Database Interface Design Toolkit. In *Proc. Int. Conf. on VLDB*, pp. 115–123, 1989.

[23] M. A. Linton, P. R. Calder, and J. M. Vlissides. InterViews: A C++ Graphical Interface Toolkit. Technical Report CSL-TR-88-358, Stanford University, July 1988.

[24] R. Miller, Y. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. 19th Int. VLDB Conf.*, Dublin, Ireland, 1993.

[25] M. C. Norrie. An Interactive System for Object-Oriented Database Design. In *Proc. 1st Int. Work. on User Interfaces to Databases*, pp. 9–24, 1992.

[26] B.C. Ooi, C. Zhao, and H. Lu. DUET - A Database User Interface Design Environment. *Journal of Intelligent Information Systems*, 3(3/4):331–356, July 1994.

[27] N. W. Paton, G. al Qaimari, and A. C. Kilgour. An Extensible Interface to an Extensible Object-Oriented Database System. In *Proc. 1st Int. Work. on User Interfaces to Databases*, pp. 265–281, 1992.

[28] K. Radermacher. An Extensible Graphical Programming Environment for Semantic Modelling. In *Proc. 1st Int. Work. on User Interfaces to Databases*, pp. 353–373, 1992.

[29] M. Rapley and J. Kennedy. Three Dimensional Interface for an Object Oriented Database. In *Proc. 2nd Int. Work. on User Interfaces to Databases*, pp. 143–167, 1994.

[30] L. Rowe, J. Konstan, B. Simth, S.Seitz, and C. Lin. The Picasso Application Framework. Technical Report UCB/ERL M90/18, University of California, Berkeley, March 1990.

[31] L. A. Rowe. Fill-in-the-Form" Programming. In *Proc. 10th Int. VLDB Conf.*, Stockholm, Sweeden, 1985.

[32] P. Sawyer, A. Colebourne, I. Sommervill, and J. Mariani. Object-Oriented Database Systems: a Framework for User Interface Development. In *Proc. 1st Int. Work. on User Interfaces to Databases*, pp. 25–38, 1992.

[33] H. Schoning. A Graphical Interface to a Complex-Object Database Management System. In *Proc. 1st Int. Work. on User Interfaces to Databases*, pp. 193–208, 1992.

[34] P. 'Noi' Sukavariya, J. D. Foley, and T. Griffith. A Second Generation User Interface Design Environment: The Model and The Runtime Architecture. In *INTERCHI '93, Proc. Conf. on Human Factors in Computing Systems*, pp. 375–382, 1993.

[35] P. Szekely, P. Luo, and R. Neches. Beyond Intferface Builders: Model Based Interface Tools. In *INTERCHI '93, Proc. Conf. on Human Factors in Computing Systems*, pp. 383–390, 1993.

[36] T. J. Teory, G. Wei, D. L. Bolton, and J.A. Koenig. ER Model Clustering as an Aid for User Communication and Documentation in Database Design. *Communications of the ACM*, pp. 975–987, August 1989.

[37] J. L. Wiener and Y. Ioannidis. A Moose and a Fox can aid scientists with data management problems. In *Proc. 4th Int. Work. on Database Programming Languages*, 1993.

[38] H. K. T. Wong and I. Kou. GUIDE: Graphical User Interface for Database Exploration. In *Proc. Int. Conf. on VLDB*, pp. 22–32, 1982.

[39] M. Zloof. Query-by-Example, The Invocation and Definition of Tables and Forms. In *Proc. Int. Conf. on VLDB*, 1975.