Is GUI Programming a Database Research Problem?

Nita Goyal HP Labs goyal@hpl.hp.com

Charles Hoch HP Labs hoch@hpl.hp.com

Ravi Krishnamurthy HP Labs krishnamurthy@hpl.hp.com

Brian Meckler HP Labs meckler@hpl.hp.com

Michael Suckow HP Labs suckow@hpl.hp.com

Abstract

Programming nontrivial GUI applications is currently an arduous task. Just as the use of a declarative language simplified the programming of database applications, we ask whether we can do the same for GUI programming? Can we then import a large body of knowledge from database research? We answer these questions by describing our experience in building nontrivial GUI applications initially using C++ programming and subsequently using Logic++, a higher order Horn clause logic language on complex objects with object-oriented features. We abstract a GUI application as a set of event handlers. Each event handler can be conceptualized as a transition from the old screen/program state to a new screen/program state. We use a data centric view of the screen/program state (*i.e.*, every entity on the screen corresponds to proxy datum in the program) and express each event handler as a query dependent update, albeit a complicated one. To express such complicated updates we use Logic++. The proxy data are expressed as derived views that are materialized on the screen. Therefore, the system must be active in maintaining these materialized views. Consequently, each event handler is conceptually an update followed by a fixpoint computation of the proxy data. Based on our experience in building the GUI system, we observe that many database techniques such as view maintenance, active DB, concurrency control, recovery, optimization as well as language concepts such as higher order logic are useful in the context of GUI programming.

1. Introduction

Programming nontrivial[1] GUI applications currently is an arduous task requiring significant time and effort to build and even more so to maintain and extend. Some reasons for this complexity are that the windowing system calls are idiosyncratic and interspersed with the logic of the application leading to spaghetti code that is hard to debug and to extend, the technology is continually evolving, not only through the set of building blocks such as widgets and the interactions between them, but also through innovative UI gestures and modalities, performance-driven decisions are implicit in the data structures and procedures and thus a major redesign of the system is required to make a change to the GUI.

Programming database applications also used to be an arduous task that was significantly simplified by the use of declarative query languages that relegated the responsibility for optimization, concurrency control, recovery, *etc.* to the DBMS. In this paper, we ask whether the use of a declarative language can similarly simplify the GUI programming problem? If so, can we apply database techniques to GUI programming and possibly extend these techniques to address the problems in the pervasive application area of GUI programming?

We answer these questions by describing our experience in building nontrivial GUI applications with traditional C++ programming and then with the use of a declarative language. We initially built a system called Forms-By-Example (FBE) in C++ that is being used in the catheterization lab at Johns Hopkins hospital for scheduling and looking up records of cardiac patients by more than 50 doctors and nurses. We then rewrote a large part of this system and extended it to build a system for Rendering-By-Example (RBE) [KZ95] entirely in a declarative language called Logic++ that is designed around a data centric architecture. In this paper our goal is to show the relevance of DB research to GUI programming based on this experience. Therefore, we do not attempt here to compare our approach to other approaches of GUI programming.

The following observation about GUI programs led to the data centric architecture and the Logic++ language. Abstractly, a GUI program can be viewed as a set of event handlers where an event is generated by the windowing system and the event handler is a transition from the old screen/program state to a new screen/program state. Thus, a GUI application starts off in an initial state and every event results in a state transition with the event handlers specifying the logic for transition. We use a data centric architecture for GUI applications where each state is viewed as data and each transition as updates on this data. Therefore, each event handler is modeled as a *query dependent update* that can be expressed in a declarative language that queries the current state and has the logic for updating the data to make the transition to the next state. To express complicated updates we use a higher order Horn clause logic language on complex objects with object-oriented features, called Logic++.

In the data centric architecture, we represent every entity on the screen with a proxy datum in the program. Since each

[1] Simple user interfaces that are done easily in GUI builders (e.g., Visual Basic, Tcl, HTML generators) are not our goal We are interested in building challenging user interfaces that would typically take many person-months to develop in Visual Basic or Tcl

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

screen entity could depend on other data in the program, either other screen entities or internal data, we express proxy data as derived views that are materialized on the screen. Each event handler is conceptually an update followed by a fixpoint computation of the proxy data. Querying and updating the proxy corresponds to querying and updating the screen, it is the responsibility of the "active" proxy data to make the appropriate calls to the windowing system and realize the necessary changes to the screen. Therefore, the DB techniques of derived view maintenance and active database will be very useful here. In our system, we also have use for techniques such as concurrency control, recovery and optimization and many declarative language concepts including higher order logic.

In section 2, we describe the scope and characteristics of the system that we have built in more detail. In section 3, we describe the data centric architecture and give the rationale for using a declarative language to build the system. In section 4, we give a brief overview of the declarative language Logic++ not only from a logic programming point of view but also from a C++ point of view. To answer the question "Is GUI programming a database research problem?", in section 5 we discuss the significant applicability of DB techniques to GUI programming problems. In section 6, we discuss some GUI programming research to show that the problems addressed through DB techniques are indeed important problems in need of good solutions.

2. System Experience

As part of the Picture Programming project at HP Labs we started by building a prototype for the ICBE system (Interoperation and Customization By Example) [ZK96] in C++. As a by-product of the ICBE system, we built a system called Forms-By-Example (FBE) using traditional C++ GUI programming. This system is currently being used in the catheterization lab at Johns Hopkins hospital for scheduling and looking up records of cardiac patients by more than 50 doctors and nurses. It has completely replaced their old process of drawing the schedules of operation theaters on a whiteboard in the hallway and of maintaining paper records of patients. The system is more than 60,000 lines large. At this stage the system became really immalleable and it became difficult to add any functionality to the spaghetti code. Therefore, we decided to redesign the ICBE system in a declarative language using a data centric architecture.

We are currently using Logic++ to develop the ICBE system. We have rewritten a large part of the FBE system entirely in Logic++ and have extended it to build a system for Rendering-By-Example (RBE) [KZ95]. RBE is a visual user interface builder for databases. The compiler for Logic++, also bootstrapped in Logic++, generates C++ code. The RBE system has more than 10,000 lines of Logic++ code generating over 170,000 lines of C++ code. The compiler was bootstrapped with less than 1000 lines of C++ code and more than 7,000 lines of Logic++ code generating over 75,000 lines of C++ code[2].

3. Data Centric Architecture

In this section we describe a data centric architecture for GUI applications that depends on the fact that these applications are mostly event-based. We discuss an extended example to clarify the architecture. We then discuss the problems associated with realizing this architecture and thereby argue that a declarative language is appropriate for programming a GUI application that has a data centric architecture.

3.1 Architecture Description

Data centric means that all entities on the screen are mapped to data in the system and the computation of what is on the screen or changes to the screen are viewed as querying or updating this data, respectively. *Event based programming* refers to the process by which an event, *i.e.*, an action on the screen by the user, is captured by the windowing system, the windowing system then calls the appropriate event handler defined by the programmer that determines what is to be done in response to that event. The logic of the event handler changes the *state* of the system including the contents of the screen.

Each event handler is conceptually a state transition function from the current state of the system to a new state.

Event handlers form all of the code for GUI applications, either called directly by the windowing system or indirectly by other procedures that are eventually called from the windowing system. If a piece of code in a GUI program cannot be executed in response to any event then that code must be useless.

The proposed data centric architecture is shown in Figure 1. It consists of two main modules, the Windowing System and the System Logic. The data in the System Logic can be the base (extensional) data or derived (intentional) data. The screen entities in the windowing system are represented in the System Logic as proxy data. The events are captured from the screen by the Windowing System that calls the appropriate event handler in System Logic. The logic in the event handler computes the required changes to the base data and to the proxy data. The changes to the data are carried out as updates, any update to the proxy data is reflected to the screen through calls to the Windowing System.

Conceptually, any event handler under this architecture is abstracted as follows:

1. Test the preconditions of the event by *querying* the data in System Logic. If the data being queried is proxy data then it might call the Windowing System to compute an answer to the query.
2. Determine the required changes to the data and *update* the data appropriately.

[2] The lines of code are mentioned here only to indicate the order of magnitude of the size of the system, *not* for the purpose of any numerical comparisons

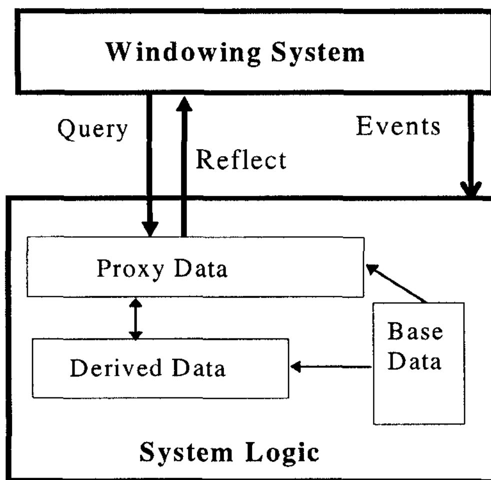

Figure 1: Architecture of the System

3. Compute the new state of the system as a fixpoint computation on the data in System Logic.
4. *Reflect* the new values of the proxy data to the screen through Windowing System calls.

"Data" in System Logic could be internal data (base or derived) or proxy data representing entities on the screen. The system logic of the event handler treats them all uniformly. Therefore, each event handler is a query dependent update.

3.2 The Handles Example

Let us consider a very simple example to illustrate the benefits of this architecture:

Most widgets in focus can have "handles" -- 8 small black squares one on each corner and edge -- that can be used for resizing or dragging the object and that usually provide a visual feedback to the user that this object has the focus. At most one object in our application can have handles. Consider a textbox widget as in Figure 2 and a click event on it. The widget will get the focus and the desired effect is that the handles must appear around it. When the textbox is resized, the handles must be repositioned around it. Programming of these handles has to deal with the problem of deciding when and where to put the handles on which widget and when to remove these handles.

Consider the program logic when we implemented the handles using traditional C++ programming with good encapsulation. There was a base class called *visual* of all widgets, that captured the behavior for handles on any visual widget. In this class we defined the event handler for the focusGot event that must be sent by the windowing system whenever a widget gets the focus. In this event handler there was logic to position the handles around the widget and to make them visible. In the event handler for the focusLost event there was logic to make the handles invisible. So far the logic for handles was encapsulated well and limited to just these two event handlers. But when we added an event handler for the resize event on the visual widget, we also had

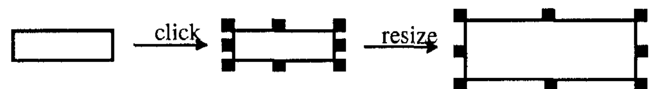

Figure 2: Events on a Widget

to add the logic to recompute the position of the handles and move them to their positions around the widget, breaking the clean encapsulation we had earlier. In fact there were many such events that forced us to break the encapsulation leading to the permeation of the logic for handles in many different places. For instance, the event to move the widget around within its parent window must reposition the handles, the event that moves the widget from one window to another must in addition reparent the handles, any event that deletes the parent window of the widget must reparent the handles before the deletion otherwise all the handles are deleted too *etc*. For all such events the position, visibility, parent or which object handles are on must be recomputed and redisplayed through the windowing system. To make the situation worse some specialized behaviors had to be put in. The handles could not be removed on every focusLost event; for instance, when the user is switching from the application to another one. Some widgets required the handles to not appear at all. Some event handlers mentioned above had to be redefined in a subclass of visual for some specialized behavior unrelated to the handles which meant that the logic for handles had to be replicated in the redefined event handler. This simple example provides a glimpse of how the interdependencies in a GUI system cause it to become complex, intertwined and immalleable.

Now consider the same program under the data centric architecture. Every entity on the screen such as the position, size, parent, visibility for the handles as well as for the widget has proxy datum in the program. The position of the handles is expressed as derived view over the position and size of the widget that is currently the focusObject (maintained by the programmer). Similarly for parent and visibility of the handles. The event handler for the click event on the textbox widget is defined in System Logic to just update the focusObject to be this widget. System Logic then computes the new state of the system by computing the fixpoint of all the derived views which will automatically recompute the positions, parent and visibility of the handles. The new state is actively reflected to the screen through the proxy data that corresponds to these screen entities of the handles. This reflection has the effect of putting the handles around the widget in the new state. The resize event handler for the widget just updates the size of the widget and is unaware of the logic for handles. The system detects that the derived position of handles has changed, recomputes the proxy data for the handle positions through the derived view and reflects this change to the screen. Similarly, for all the events that affect the handles, the logic for the handles is isolated to the handle class only.

3.3 Problems in Architecture Realization

The data centric architecture provides a very simple abstraction of the data and events in GUI applications. This simplicity of the architecture translates to complexity in its implementation. The realization of the data centric architecture posed a number of challenges, some of which we discuss below:

1. The data, whether base, derived or proxy, have a large number of interdependencies. In the data centric architecture, the system is responsible for keeping track of all the data dependencies and providing notifications to the data if and when its values change.
2. One of the biggest challenges of the data centric architecture was to keep the proxy data always synchronized with the screen entity that it corresponds to. This "active" reflecting of the data is in fact what makes this architecture truly data centric. The implementation of the reflect mechanism raises the question of *when* and *how* the proxy data must be reflected. We will discuss reflecting proxy data in significant detail in section 5.2.
3. The data in GUI applications is usually complex and therefore is better represented as "objects" rather than as relational data. Therefore, languages such as Datalog, LDL and Prolog [Ull89, NT89] are not sufficient for the purpose of modeling the GUI objects.
4. The query dependent updates in an event handler require a powerful language to represent them since they are quite complicated. For example, if we want to allow a widget to be resized on the screen such that the other widgets close to it are pushed around to make space for it, then the queries and updates required to express this logic is nontrivial. Therefore, we do want a powerful declarative language to express the powerful event handlers. In [KN88, KLK91], a complex data language based on higher order logic was proposed. In section 4, we describe an enhanced version of that language (called Logic++) that we used in the implementation of the data centric architecture. Constraints have been used in GUI programming research, in most cases their expressive power is less than Logic++ and there are performance problems. We will revisit this issue in section 6 on related work.
5. For good performance of the application, the declarative language used for implementation must have provisions for optimization *after* the logic of the program is working. In Logic++ we have many such provisions including materialization of derived views that is discussed in detail in section 5.1.

4. Declarative Language Logic++

[KN88, KLK91] proposed a conceptual structure of the Logic++ language, under a different name, based on objects and expressions on objects. We briefly describe the language here; its logic programming features in the first subsection and its object-oriented features in the next[3]. The reader is referred to [KN88, KLK91, GH*96] for a detailed discussion.

4.1 Logic-Oriented Overview

An object in Logic++ can be classified into one of three categories: an atomic object, a tuple of objects, or a set of objects. Examples of atomic objects are integers, characters, *etc.* A tuple object is recursively defined as a collection of attribute/object pairs, in which the object corresponding to any attribute can be atomic, tuple or set object. A set object is a collection of atomic, tuple or set objects. For example, {(name:john, sal:10K),...} is a set object whose elements are tuple objects. The language also allows aggregates such as lists and arrays that are similar to the sets but for brevity we omit that discussion here.

Elements of a tuple can be accessed by reference to the unique attributes, serving to "name" objects, whereas a set is queried by the contents of the object alone. Attribute of a tuple or an element of a set can be either extensionally specified to be a particular object or can be intentionally specified by a rule as in any Horn clause language (*e.g.*, Datalog, LDL) [Ull89]. Extensionally specified attributes are called *base* (facts) and intentionally specified attributes are called *derived.* In addition, there are two other categories of attributes: *foreign* and *procedure*.

We present an overview of the language through an example. Consider a database containing the emp and dept relations. Let the database be represented as a tuple containing the two attributes named emp and dept, the value of each being the corresponding set of tuples. Consider the following derived view (*i.e.,* rule denoted by the left arrow :-) of experienced managers who manage older employees (age>50) in California.

```
db.expMgr{M} :- db.emp{(.age>50,.department=D)}
                & D.loc="CA" & D.mgr = M .
```

The derived view expMgr is an attribute of the db tuple. Note that db is a globally known constant. An intuitive way to read the above rule is to view the right hand side of the rule as a conjunct of 3 path expressions. The first conjunct binds a set of references to emp tuples with employees aged greater than 50; note that the value bound to **D** is a reference to a dept tuple. The next conjunct states a condition on the loc attribute of these dept tuples. The last conjunct binds the set of managers from these tuples. As usual, variables are capitalized. Note that each prefix such as db.emp, **D** or **D**.mgr refers to one or more objects that can be subsequently selected by an appropriate expression or bound to variables for subsequent use such as selection or join with other objects. Therefore, by defining the three types of expressions

1. atomic expressions, *e.g.,* >50, ="CA", = **M**

[3] As our intention in this paper is *not* to propose yet another language but only to use the language in building challenging GUI applications, we do not attempt to compare this language with other proposals for higher order logic languages and complex object languages

2. tuple expressions; *e.g.*, .age, .loc
3. set expressions; *e.g.*, { .....}

the language has significant expressive power. We will elaborate on many of these points in the next subsection.

Procedure is the third category for an attribute and is needed to model actions and updates to base data. Intuitively, procedures can be viewed as query dependent updates allowed in QBE or SQL. The notation for procedures is similar to the rules but with a right arrow "-:" instead of the left arrow. The head has the calling procedure name with parameters and the body has the definition of what is to be done. The semantics are close to the LDL proposal [NT89, KLK91] which is functionally similar to the NAIL! [Ull89] and Coral [RS*93] approach. That is, procedures can use derived rules in their definition but rules cannot use procedures.

The updates in Logic++ are denoted by ":=" for atomic data and by "+" or "-" for insertion and deletion, respectively, into a set of data. For example, **D**.loc := "CA" means that the location of department **D** should be CA; db.emp+{(.name="John")} means that a tuple with name John must be inserted into the set of employees; and db.emp-{(.age>60)} means that all tuples with age more than 60 must be deleted from the set of employees.

Negation is denoted by "!" for negation of an expression. For example, **D**.loc != "CA" means that the location of department **D** is not CA; db.emp!{(.age<20)} means that there is no tuple in the set of employees with age less than 20; and **E**!.loc means that the object bound to **E** does *not* have a loc attribute. In effect, any expression can be negated and for aggregates (*i.e.*, tuple and set) not-there-exists meaning is given. We use the traditional stratified negation semantics appropriately modified to deal with complex objects. In fact our experience showed that we needed locally stratified negation as induced by the acyclic nesting of objects. Informally, if O_a.P (the attribute P of the object instance O_a) depends on O_b.P then $O_a < O_b$ must be implied by the acyclic nesting of objects.

We also needed the capability to nondeterministically choose one amongst many different values in a rule. Suppose we needed to call someone in CA, anyone as long as that person is in CA, to find out if CA has a particular state regulation. Then the rule has to choose nondeterministically anyone in CA and this is provided by nondeterministic choice as proposed in LDL [NT89].

Higher order variables in Logic++ are mostly used to bind attribute names of a tuple; *e.g.*, consider the expression emp{**E**} & **E**.**X** = **D** & **D**.loc, where **E**.**X**, binds to **X** the set of attribute names in the employee tuple and **D**.loc tests whether the (department) object that is bound to **D** has an attribute called loc. In some sense, the higher order variables allow the program to reason over the itself. We will elaborate on this in the next section.

The fourth category of attributes is *foreign* and refers to datum in the Logic++ program that is proxy for some entity in an external system. Windowing system is the external system in GUI programming, and the screen components are the entities of this system. A foreign attribute could be an atomic or a set. The programmer must define the updater (:= or + or -) and accessor (= or { }) methods for these attributes that are the means of communication between the proxy datum and the screen entity. We will elaborate on the usage and benefits of these attributes in the following sections.

4.2 Object-Oriented Overview

We adhere to the C++ semantics for all the object-oriented capabilities for the sake of expediency and acceptability. Therefore, this subsection is a discussion of how the C++ capabilities are combined with the logic-oriented capabilities of Logic++ discussed earlier.

Conceptually a C++ program is a collection of classes, each with its data members and method definitions as well as information about inheritance, encapsulation, overloading, polymorphism and other OO capabilities. In such a context, view an instance of a class to be a tuple in Logic++ whose attributes are the data members and methods. We can map from C++ classes such that each data member is a base attribute, each const function (*i.e.*, method with no side effects) is a derived attribute and all others (*i.e.*, methods with side effects) are procedure attributes in Logic++.

As in C++ we require data declaration. Each class declaration has the name of the class, its parent(s) in the inheritance hierarchy and protection/scope specifiers (*e.g.*, public, protected, static, virtual). For every attribute of the class, its category (base, derived, foreign, procedure) and type (including whether atomic/set/array/list) are also specified. The data declaration for the handles example is shown in Figure 3.

The semantics of all the OO capabilities is their meaning in C++ with some extensions. The following guidelines may be useful to the reader:

1. All attributes (from all 4 categories) can be inherited and redefined. A base/derived/foreign attribute can be also be redefined as some other category from base/derived/foreign. The redefinition of a foreign attribute as a derived attribute will be shown to be particularly useful in GUI applications.
2. An attribute of a class can be referenced using the class specifier. For example, if **H** is an object of type *handle*, then **H**.visible refers to the visible attribute in handle class, whereas **H**.*visual*::visible refers to the visible attribute in *visual* class (which is a superclass of handle).
3. Globally accessible, statically created objects (*e.g.*, *universe*::theUniverse) can be used in any rule, otherwise, only path expressions leading from bound variables in the head of the rule or procedure are valid.

4. External C++ procedures can be called from within the Logic++ procedures.
5. A base/derived/foreign attribute can be declared to be either an embedded object or a pointer. All pointers are declared by adding a "*" to the end of the class name (as in C++). In the example in Figure 3, base attribute focusObj of *universe* class is a pointer to a *visual* object, base attribute handles of *universe* class is a set of pointers to *handle* objects, and, foreign attribute top of *visual* class is an embedded *coord* object.
6. Logic++ syntax allows attributes that are specific to a particular attribute of a class. For example, *handle*::onobject@initialize denotes a procedure named initialize in the *handle* class that initializes the object that the onobject attribute points to. It can be applied *only* to the onobject attribute; *not* to an instance of *handle* or *visual* classes. We can also use this syntax to reason about the program itself. For instance, H.*handle*::onobject@category=derived means that the category of the onobject attribute for the *handle* class has the value "derived" where @category is a system provided method on any attribute of any class and derived is a constant. Using this syntax to reason over the program will be exemplified in the next section.

The rules and procedures for the handles example will be as follows in Logic++.

```
Data Declaration in Logic++

class universe
Base:[      public focusObj      :visual* .
     static public theUniverse   :universe* .
            public handles       :{handle*} .  ]
endClass

class visual
Foreign:[ virtual public top      :coord .
          virtual public left     :coord.
          virtual public width    :coord.
          virtual public height   :coord.
          virtual public visible  :boolean .  ]
Base:[ public canNotHaveHandles:boolean . ]
Procedure:[
     virtual public click() . ]
endClass

class handle isa { public visual }
Base:[        side            :handleSide .
     static   handle_height :=  coord(96.0) .
     static   handle_width :=   coord(96.0) .  ]
Derived:[ static    onobject:visual.
          virtual   visible   :boolean .
          virtual   top       :coord
          virtual   left      :coord.   ]
endClass

Note: static, virtual, and public are keywords with same
meaning as in C++
```

Figure 3: Example of Logic++ Program

Example: *handle* class has a derived static atomic attribute onobject that is the object on which the handles must appear. It depends on two base facts; the widget with the focus and whether this widget is allowed to have handles. The following rule defines onobject for *handle* class.

```
handle::onobject = O :-
    (universe::theUniverse).focusObj = O &
    if O.canNotHaveHandles then O = NULL fi
```

Using this derived attribute the visibility of a handle can be defined as follows[4]:

```
H.handle::visible :- handle::onobject=O & O !=NULL
```

Note that focusObj could be changed by any number of events but the logic for the visibility of the handles remains unchanged. Same will hold for the rules describing the coordinates and parent of the handles. Thus we have managed to separate the logic of the handles from its usage. An example of an event that changes focusObj is the click event on a widget. We describe below the click event handler on a widget as a procedure that queries the current data and updates the base facts[5]:

```
V.visual::click() -:
            (universe::theUniverse).focusObj := V
```

The click event handler is really simple in Logic++ because all the logic related to handles can be removed from the click procedure. This is possible because the system takes the responsibility of reflecting any resulting changes in the proxy screen data (foreign attributes) such as *handle*::visible to the screen. In the next section we will see how the system accomplishes this data reflection. We will also discuss the applicability of database features to GUI programming in Logic++.

5. Applicability of DB Features to GUI

In this section we describe the features that were derived from the database literature and that were found to be useful in developing the GUI system. In general, logic was extremely useful for a number of reasons: it allowed for programming independent of data structure, particularly for aggregates; derived views provided a transparent way of accessing stored as well as computed data; materialized derived views provided the optimization capability; higher order logic made it possible to reason over the program and provided generalized features; the use of negation (particularly locally stratified negation) was useful; and, nondeterministic choice was useful in choosing a subset in a rule. The object-oriented feature that was particularly useful (apart from the usual benefits of inheritance, redefinition, overloading, polymorphism, encapsulation *etc.*) was the ability to redefine a foreign

[4] The "this" object in a C++ method is explicitly named as H in this rule, *i.e.*, H is the object on which this rule is invoked

[5] A procedure has the right arrow -: whereas a rule has the left arrow :-.

attribute as a derived attribute. We discuss some of these features in detail.

5.1 Materialized Derived Attributes

In large applications most of the data are interdependent. These dependencies make programming and maintenance of programs really hard because change in one datum might require changes in other data with a ripple effect that is extremely hard for the programmer to keep track of. The derived attributes in Logic++ programs are meant to alleviate this problem. Ideally, all base attributes must be independent data and any data that are dependent on others are defined as derived attributes. Then the programmer only has to update the appropriate base attributes and the system automatically computes the right value of derived attributes on demand. We saw an example of this in section 4 where focusObj and canNotHaveHandles are base attributes in *visual* class and onobject in *handle* class is derived.

If derived attributes are always computed on demand, it may result in poor program performance. Therefore, Logic++ allows derived attributes to be materialized, *i.e.,* the first time that the value is computed it is cached away and on subsequent requests the cached value is used, thereby amortizing the computation over multiple usage. The only effort required by the programmer is to add the keyword "materialized" to the declaration of the derived attribute. The system then takes the responsibility for materializing and maintaining the derived views even when the data that it depends on changes. For instance, if onobject is materialized and then subsequently focusObj changes, then onobject must be recomputed. This means that all derived views that depend on onobject must also be recomputed. This is a nontrivial view maintenance problem since not only can a large number of attributes be materialized but also an attribute can be defined by a complex rule involving a large number of attributes.

A solution to this view maintenance problem is to manage the dependencies so that when focusObj is changed then onobject get notified. Such "data_changed" methods are provided in languages such as SmallTalk where the programmer must explicitly set up the dependencies for the data_changed notifications. Such a responsibility on the programmer invariably results in bugs that are not easy to track down. In Logic++, where the body of the derivation rule can be analyzed, such a propagation of the updates can be compiled automatically. Thus, if a derived attribute is materialized, any change in the data that it depends upon, either directly or transitively, will result in a notification to the derived attribute. Therefore, next time that this attribute is used it will be recomputed and rematerialized.

Our current approach of data_changed notification and complete rematerialization could be improved using active DB research [WC96] and incremental view maintenance techniques from DB research [GM95]. Since our situation is different from traditional DB problem areas in that the data is in the form of complex objects, is usually smaller in size and the memory resident property allows the use of pointers, it offers a new direction of research to adapt the current DB techniques to these new situations.

In our system-building experience, both for the Logic++ compiler and for the RBE system, the derived views helped us immensely in programming as well as program maintenance. The materialization of many of these views *post facto* significantly improved the performance. In fact, for reflecting the proxy data to the screen in GUI applications, view materialization not only enhances performance but is also a necessity.

5.2 Actively Reflecting Proxy Data to Screen

In a nontrivial GUI application there are a large number of screen entities and most of them depend on other screen entities and on data throughout the program. For example, the visibility and coordinates of a handle depend on the object that it is on and the coordinates of this object. When programming GUI in C++, the explicit updates to these screen entities are sprinkled all over the program and the programmer is responsible for ensuring that the appropriate updates are percolated through to all the affected screen data, leading to spaghetti code that is hard to write and maintain. In the data centric architecture for GUI applications, corresponding to each screen entity we have proxy datum in our program as a foreign attribute. Most of the screen proxy data are derived from other internal data in the program as well as from other proxy data. As the proxy data are defined as derived rules, in effect the screen entities are a materialization of these rules on the screen. We describe here how this materialization of the proxy data, that we term "active reflecting", takes place so that the screen is always synchronized with the internal program state.

The problem of active reflection occurs not only with GUI systems but also while interacting with any external autonomous system such as a set of instruments, a database system, workflow system, or any hardware/software system. In all these cases the information, obtained as a result of computation, is conceptually recorded as proxy data that must always be synchronized with the external system. In this paper, we restrict our attention only to the case of a windowing system. The general case, albeit more difficult, is an extension of this approach.

We first discuss why active reflecting is hard and derive from that a desiderata for any solution. Then we discuss our solution implemented in Logic++. The two main problems with active reflecting are deciding *how* and *when* to reflect proxy data.

How: We do not want the windowing system calls to be sprinkled throughout our code. Therefore, windowing system calls to push proxy data to the screen or to get screen data into the program must be well *encapsulated.* To reflect the data we must also be able to determine the current value that should be

pushed to the screen. Depending on the semantics of the application, this value could conceptually be base data available by querying the windowing system or could be derived data obtained by computing the corresponding rule. In fact, it could be base data in a superclass (*e.g.*, visible attribute in *visual* class) and be redefined as derived data in a subclass (*e.g.*, visible in *handle* class). The reflect mechanism must be *transparent* to the programmer so that they can express the computation of the data without external considerations in a truly declarative manner. Furthermore, the reflect mechanism must be *general* enough so that the redefinition of data does not require redefining the reflect for that data.

When: As discussed in the last subsection, derived attributes can be (re)materialized lazily whenever someone demands the data. In contrast, in the case of foreign attributes, proxy data must be (re)materialized *eagerly* on the screen whenever the data changes value since the screen is seen by the user of the application. But the problem is more complicated than just actively reflecting a foreign attribute as soon as it changes value. The *order* in which the proxy data is reflected is also very important semantically. For example, for a given window in Microsoft Windows, changing its parent window followed by changing the top coordinate of the window has a different effect than changing the top coordinate followed by changing its parent window. Therefore, depending on the desired semantics, one or the other order of reflection must be chosen. Another problem is that in some event handlers it might make sense semantically to "batch" the internal data updates and reflect the updated proxy data only after the whole batch is completed. That is, reflecting proxy data immediately after one data update might lead to a wrong result. Therefore, we must allow for a *commit* to the screen updates only after a batch of data updates are completed.

Intuitively, reflect is similar to the DB notion of forcing updated data to the disk, and batching of data updates is analogous to buffering in DBMS. The data updates are buffered in memory and reflected to the screen at the appropriate time. The appropriate time may be immediately (*i.e.*, write through to the screen) or at commit time. Buffering of updates also improves the performance of reflecting since multiple updates to the same data before commit time are reflected only once. To draw the analogy with DB techniques further, the need for recovery from semantic errors in the GUI application (*i.e.*, avoiding any effect of that event on the screen) has identical problems to that of database recovery.

Our approach to active reflecting, implemented in Logic++, addresses most of the aforementioned problems. The approach to encapsulate windowing system calls has been taken in VisualWorks by ParcPlace. In Logic++, the *encapsulation* problem is addressed by putting all the windowing system calls in the accessor and updater methods for foreign attributes that the programmer must define (as mentioned in section 4.1). For example, the visible attribute for the *visual* class will have the following two definitions:

V.*visual*::visible = **X** -: windows calls to GET the visible of the widget corresponding to **V** ...... (A)

V.*visual*::visible := $**X** -: windows calls to SET the visible of the widget corresponding to **V** ....(B)

$**X** indicates that **X** is a bound variable. The calls to get or update the value of visible are same as that for any other base/derived data. The caller not only is unaware that windowing system calls are being made but also is unaware that the data is proxy for a screen entity. Similar rules can be defined if the foreign attribute is an aggregate object.

To have a *general* and *transparent* semantics for active reflecting of each foreign attribute, we define a reflect procedure for each attribute[6] as follows:

V.*visual*::visible@reflect -: **V**.visible = **X** &
V.*visual*::visible := **X** .

This procedure first computes the visible attribute virtually and then propagates it to the screen using the update method (B). This reflect procedure is *general* because the programmer can redefine visible in any subclass as a derived view without affecting the reflect procedure. If visible is not redefined for the object **V** then by default it calls accessor method (A) in the *visual* class. The reflect mechanism is *transparent* because the redefinition of visible in any subclass need not know that it is being reflected as a foreign attribute in the *visual* class. Therefore, *handle* can redefine visible as a derived rule in a purely declarative manner and the reflect procedure will ensure that the right value is computed and propagated.

To address the issue of when and in what order to reflect the foreign attributes, we define a ReflectMgr class. An instance of ReflectMgr corresponds to a specific external system (*e.g.*, Microsoft Windows) that it is managing the reflections for. Conceptually, an instance of ReflectMgr has the list of foreign attributes to be reflected at any time in the appropriate order and a commit procedure that initiates the process of forcing the updates to the screen. Each foreign attribute that is changed, either explicitly by the programmer or indirectly as materialized derived data, is added to the list in the ReflectMgr. The commit procedure reflects all the attributes on the ReflectMgr list in the appropriate order, by calling the reflect method on each attribute. For batched update to the screen, the commit is called typically at the end of each event handler. The programmer can also just commit an attribute individually at any time rather than all the attributes on the list. To address the problem of ordering the attribute reflects, we assume a default ordering and it is possible for the programmer to change that ordering at the class level or at the attribute level if they so desire.

The reflect mechanism described above can be extrapolated from reflecting onto a windowing system to reflecting for any external autonomous system. This will require defining another instance of ReflectMgr for that external system. Also

[6] *visual*::visible@reflect is a procedure on the visible attribute of the *visual* class as described in section 4.2

in the declaration for each foreign attribute, the corresponding ReflectMgr instance must be specified so that the system knows which manager has the responsibility for reflecting it.

Since the reflect procedure is the same for all attributes, it can be stated by the following higher order logic procedure for all the foreign attributes.

```
V.visual::W@reflect -: V.W@category = foreign &
                       V.W = X  &  V.visual::W := X .
```

The above procedure is "safe" only if the value for the variable **W** is bound and there is an appropriate condition to ensure that **W** is a foreign attribute. In the next section we describe how we can use the higher order capability to express such meta-procedures.

5.3 Higher Order Language Capability

LISP, SmallTalk and other programming languages have made an effective case for the ability to write programs that can reason about themselves. Templates in C++ are a meager attempt at this. Logic++ uses the higher order variables (*i.e.*, variables quantified over attribute names) and the ability to reason over properties of these attributes to write rules and procedures. This feature was found to be quite useful not only for its succinctness but also for ensuring strong encapsulation. Efficient execution is usually not associated with such higher order capability or any language that can reason about itself. We show that efficient implementation is possible using DB techniques such as magic sets [Ull89].

As we mentioned in the last subsection, the reflect procedure can be defined using a higher order variable that is quantified over attribute names. Similar procedures can be written for other functions such as save, load, copy, cut *etc.* Consider the save procedure (saving to some persistent storage) for any object **V**. Saving it requires the recursive saving of each its attributes. This procedure, simplified for expository reasons here, can be expressed through the following higher order procedure:

```
V.save -: V.X@category = base & V.X.save .
       -: V.X@category=foreign & V.X=Y &
          Y.save .
```

The base attributes of **V** are saved by recursively calling the save procedure on these attributes. The base case of the recursion is saving elementary types such as integer, string *etc.* Each foreign attribute is saved by first computing its value and then saving that value. If no new base or foreign attributes can be defined at run time, then the above recursive save procedure can be unfolded at compile time for a specific class by instantiating the attributes of that class. This will result in a first order logic procedure that can be compiled efficiently.

In general, the process of computing the set of values that will be bound at run time was proposed in the magic set [Ull89] rewrite rules. Here we observe that when the set of values that can be bound to the higher order variable is invariant for all possible data values at runtime, then a higher order rule/procedure can be compiled efficiently.

Consider another example of higher order logic. Foreign attribute definition is intended to encapsulate the calls to windows. The use of higher order procedures was found to be useful in these foreign attribute definitions. This is because many properties of widgets are propagated to the windows using similar call structure. This can be conceptually viewed as follows:

```
V.visual::visible = W  -:  get widget id for the object V
        and then getProperty(id, "Visible", W)
V.visual::visible : = $W -:  get widget id for the object V
        and then setProperty(id, "Visible", W)
```

Aside from the constant associated with the visible attribute (denoted above as "Visible") and the attribute name visible, the code is replicated for many such attributes. So the use of higher order procedures for these attributes significantly simplifies the code and once again they can be compiled efficiently in most cases.

Even though the above two cases of save and foreign attribute definition occur widely in the system, they are a somewhat simple usage of higher order logic. A more interesting case is the use of higher order logic to provide multiple inheritance and more complicated inheritance of classes.

As the name suggests, multiple inheritance for a class (say *userButton*) occurs when *userButton* inherits from multiple classes (say *button* and *user*). Multiple inheritance problems occur when any two parent classes have a common ancestry class in their inheritance. As a result, reference to all attributes in that common class are ambiguous. This ambiguity is usually resolved by explicitly redefining each of these attributes in the *userButton* class by a wrapper method that explicitly denotes the parent class from which to inherit that attribute. This poses two problems: first, there may be many attributes that have to be redefined in this manner; second, every time the common parent class changes or a new attribute is added, the class with multiple inheritance needs to be modified, resulting in the violation of encapsulation.

Multiple inheritance is obtained using higher order procedures as follows. Let the *button* class be the primary inheritance parent. Then define the following rule

```
C.userButton::X :- C.button::X
                :- C.user::X & C!.button::X
```

The above rule defines all the attributes in classes *user* and *button* (and therefore in their common parent) to be attributes of *userButton* with the caveat that if the same attribute occurs in *user* and *button* then the one from *button* is inherited. An efficient compile time code generation is possible by generating the code that eliminates all references to higher order variables. If we want a particular attribute to come from class *user* even if it is an attribute of *button*, then programming that is possible because the language allows the program to reason over itself.

The above kind of multiple inheritance is a simple case of a more complicated inheritance that is often needed in large applications. One such inheritance is what we term XOR inheritance. Conceptually, this requires the inheritance of the *widget* class from an exclusive OR of two classes *configurer* and *user* such that only a lowest subclass of *widget*, a leaf class *userButton*, specifies which one of the two classes, *configurer* or *user*, is the parent class. Writing code in C++ to handle such inheritance was quite convoluted and posed difficult programming and debugging problems in the Johns Hopkins application. The ability to reason over the inheritance hierarchy allows the Logic++ language to specify the particular kind of inheritance required by the application. The case of XOR inheritance is discussed in detail in [GH*96].

In summary, we found many uses for higher order logic; in particular, modeling nontrivial semantics that vastly simplified the code.

5.4 Concurrency Control and Recovery

The responsibility of concurrency control and recovery in database applications was relegated to the DBMS. As a result, any application could avail these capabilities with very little overhead. Many of these applications could not otherwise have economically justified having to program these capabilities from scratch. Similarly, most GUI applications today provide very limited, if any, concurrency control and recovery functionality other than for data stored in a DBMS.

Concurrent updates to the screen of a GUI application simultaneously done by multiple users with the correctness criterion that all these interleaved human-computer interactions be "serializable", are the realm of custom programming. The Johns Hopkins application has such a cooperative work requirement; *i.e.,* many doctors and nurses are viewing the schedule on their screen and by drag-drop gestures moving the patient from one room to another, changing the status of the patient and jotting notes for one another. Conceptually, this is a WYSIWYG edit of the schedule simultaneously by tens of doctors and nurses such that every update to any screen gets propagated to all the other screens. This poses two major problems: 1) Determining the correctness criterion for such an interleaved, concurrent WYSIWYG edit; and 2) assuring interactive response time.

Firstly, the concurrent human-computer interactions to any applications are to be analyzed for correctness based on some "serializability" criterion. Such criteria are typically application dependent. In our opinion, the intuitions from database serializabilty research can be extrapolated to devising such correctness criteria for interleaved human-computer interactions.

Secondly, such an application poses concurrency control and recovery requirements that cannot be met by relegating this responsibility to a DBMS because most commercial DBMS' do not have active capability. Even if active DBMS were available, this would be impractical. For instance, resizing of a widget cannot be done by propagating the data to a DBMS and then letting the DBMS propagate it to every other screen, without sacrificing response time. In our data centric architecture, concurrent human-computer interactions map to data queries/updates. Therefore, traditional DB concurrency control and recovery techniques can be extrapolated to this new problem.

For instance, in our implementation of the Johns Hopkins application, we needed the following to assure interactive response times:

1. Semantic checks for conflict of updates;
2. A combination of optimistic and pessimistic concurrency control;
3. A combination of log based and shadow based recovery schemes.

In summary, there are many interesting concurrency control and recovery problems posed by the current GUI applications that can be solved using intuitions gained from the database research.

5.5 Access Methods

Disk I/O was (and could still be) the bottleneck for most DB applications. In a DBMS, efficient access to physical data on disk was the responsibility of indexing techniques that were independent of the application and disk I/O idiosyncrasies. Declarative queries were not concerned with such performance issues and as a result provided physical data independence. The compiler was responsible for choosing the appropriate *access* methods to assure the performance.

In contrast, GUI applications spend most of their time executing windows code; *i.e.*, output to the screen is the most time consuming operation. So it is fair to ask whether any generic *output* methods to windows can be designed that are application independent and whether our intuitions from building access methods can be used here.

In Logic++, the output to the external system is done through procedures on foreign proxy data. We saw an example in section 5.2 (rule (B)) where the operator **V**.visible := **X** was redefined to reflect the value of the visible attribute to the screen. This operator redefinition can also encapsulate any generic *output* methods that improve the performance of screen I/O, analogous to the access methods that improved the performance of disk I/O.

In today's GUI applications, as it was in 1960's for DB applications, custom output methods to screen are written that are application and windowing system specific; *i.e.,* dependent on the idiosyncrasies of the application and windowing system. In this sense, culling generic output methods independent of the application and windowing system would be of immense value, particularly if the compiler can automatically make use of such methods.

To understand the concept of output methods, consider the following GUI application. On a map of a city, the set of for-sale houses that are within a specified price are shown as dots on the map. The maximum price is specified in a slider

widget. When the price is increased by $10K, the number of dots on the city map will increase monotonically. If we are to repaint the screen by drawing *all* the dots again then the application will be quite inefficient as repaint is an expensive operation. On the other hand, if we can deduce the monotonicity and have a generic output method that can incrementally repaint only the new dots on the screen, then the performance will improve considerably. At a metalevel, the following two observations are analogous:

- B-tree reduces I/O by avoiding sequential scan and accessing a small subset.
- Incremental repainting reduces screen painting by limiting it to a small subset.

The availability of a declarative program enables the compiler to deduce properties such as monotonicity and allows the use of generic output methods such as incremental repainting for performance improvement.

6. Related Work

In this paper, we have argued that GUI programming is a DB research problem. We validated this argument by describing our experience in building a GUI system using many of the concepts and techniques from DB research. This experience showed us the relevance of DB techniques to many GUI programming problems and brought to light the need for further research on these techniques to solve the problem of building GUI applications. In this section we discuss some GUI programming research where the researchers have independently observed these GUI programming problems and have made a case for the importance of finding good solutions.

Extensible constraint based UI toolkits such as Amulet from CMU [MM95] enable the use of custom constraint solvers that are application dependent. Myers observes that "a disadvantage with constraints is that they require a sophisticated runtime system to solve them efficiently" [M95 p.23]. Derived views (*i.e.*, views defined using other views), as a constraint specification language and view maintenance as a constraint solving mechanism, show the applicability of view maintenance technology in the constraint based UI tools. Extrapolating from the DB experience and our system, maintaining materialized views can significantly reduce the runtime overhead observed by Myers.

Automatic redisplay of modified data to the screen is useful as evident from the following quote: "The automatic redisplay relieves the application programmer of the responsibility of implementing potentially complicated data structures and algorithms for display maintenance." [vZ94] In Amulet the automatic redisplay is implemented as daemons that handle redisplay of changed graphical objects by queuing them for later redrawing. We observe that such a queuing of output is analogous to the capabilities of a buffering manager in DBMS. Therefore, DB experience will be valuable in this context.

"As user interface applications start to become network aware and support collaborative work, they will be increasingly be called upon to operate in concurrent and distributed fashion." [BH95] This requires that concurrent, interleaved, distributed human-computer interactions to an application has to be logically consistent. Some of the guarantees that need to be met include that each user sees changes in the order they occur, always sees the cause of a change before the effect, and whenever a colleague communicates with you, your view shows every change that theirs does. [BH95] This we observe as requiring a new notion of serializability of human-computer interactions and insights from DB research can be of much value. Further, the user interactions such as 'OK' and 'Cancel' are analogous to commit and abort. This makes the case for importing recovery technology to provide these features generically in lieu of implementing them in an application dependent manner.

In [vZ94] it is observed that the necessity to compute the delta changes to the screen is a necessary operation for achieving acceptable performance of the system. "Update plans require information about which objects an operation modifies, what regions of the screen these objects inhabit, and what kinds of objects these objects may overlap. [vZ94] Striving for application independent solution for update plans is expected to be impractical. We observe that efficient output to the screen is analogous to efficient I/O to disk. We argued that techniques for efficient output to screen can be devised similar to the indexing technology that reduced the disk I/O time. In fact, an example of such an output method was discussed in the context of the real-estate example.

In summary, GUI researchers have independently observed these problems that were faced by DB researchers a couple of decades ago, and as argued previously, some of the solutions to these problems are applicable in this new context.

7. Conclusions

Just as database research addressed the software engineering problems in the limited area of database applications, we seek to address the software engineering problems of GUI applications. Imitating the database experience, we use a declarative language in a data centric architecture to program GUI applications. We report here from our experience with the nontrivial RBE system built both in C++ and in Logic++. We observe that programming in Logic++ simplified the system development that resulted in significantly reduced development time. This simplification is not only due to declarative programming but also due to the fact that the strong encapsulation of all classes reduced the inter-class dependencies, idiosyncratic windowing system code was completely separated in the encapsulated code, and the resulting program was significantly smaller than in C++ that facilitated its human comprehension.

Based on the system experience we also conclude that results/intuitions from many areas of database research can

significantly contribute to solving the GUI programming problem. Some of these areas are:

1. Derived views and active database capabilities for proxy data objects can provide a concise and intuitive way of writing GUI programs.
2. Materialized view maintenance can provide a way to enhance performance *post facto*.
3. Declarative language: negation, recursion, nondeterministic choice, higher order logic, *etc.* were found to be quite useful.
4. Concurrency control and recovery over concurrent interactions with the GUI by multiple users will be beneficial.
5. Output methods, analogous to access methods in database research, could be designed to enhance performance.

Furthermore, GUI programming problem poses new challenges in each of these areas. By identifying these areas we have shown that GUI programming is indeed a database research problem.

Acknowledgment: We sincerely thank Surajit Chaudhuri, Umesh Dayal, Ashish Gupta, Waqar Hasan, Bill Kent and the reviewers for feedback on the paper.

References

[BH95] Krishna A. Bharat, and Scott E. Hudson. Supporting Distributed, Concurrent, One-Way constraints in User Interface Applications. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, pages 121-132. Pittsburgh 1995.

[GH*96] Nita Goyal *et. al.*. Logic++: A Higher Order Logic Language with Object Oriented Features. In preparation.

[GM95] Ashish Gupta and Inderpal S. Mumick.. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *IEEE Data Engineerıng Bulletin, Special Issue on Materialized Views and Data Warehousing*, Vol 18, no. 2, June 1995.

[KLK91] Ravi Krishnamurthy, Witold Litwin and William Kent. Language Features for Interoperability of Databases with Schematic Discrepancies. In *International Conference of SIGMOD*, pages 40-49. Denver 1991.

[KN88] Ravi Krishnamurthy and Shamim Naqvi. Towards a Real Horn Clause Language. In *Proceedings of VLDB*, pages 252-263. Los Angeles 1988.

[KZ95] Ravi Krishnamurthy and Moshe Zloof. RBE: Rendering By Example. In *International Conference on Data Engineering*, pages 288-297. Taipei 1995.

[MM95] Rich McDaniel and Brad A. Myers. Amulet's Dynamic and Flexible Prototype-Instance Object and Constraint System ın C++. CMU-CS-95-176, July 95. URL: http://www.cs.cmu.edu/~amulet

[M95] http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ garnet/doc/papers/uimssurvey.ps; an earlier version published as "State of the Art in User Interface Software Tools," in *Advances in Human-Computer Interaction*, Vol. 4, pages 110-150. Edited by H. Rex Hartson and Deborah Hix. Norwood, NJ: Ablex Publishing, 1993.

[NT89] Shamim Naqvi and Shalom Tsur. A Language for Data and Knowledge Bases. W. H. Freeman, 1989.

[RS*93] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan and P. Seshadri. Implementation of the CORAL Deductive Database System. In *Proceedings of SIGMOD*, pages 167-176. Washington D.C. 1993.

[WC96] Jennifer Widom and Stefano Ceri (ed.). Active Database Systems. Morgan Kaufmann Pub., 1996.

[Ul189] Jeffrey D. Ullman. Database and Knowledge-Base Systems. Vol. II, Computer Science Press, 1989.

[vZ94] Bradley T. Vander Zanden. Optimizing Toolkit-Generated Graphical Interfaces. . In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 157-166. Marina del Rey 1994.

[ZK96] Moshe Zloof and Ravi Krishnamurthy. ICBE: Interoperation and Customization By Example. In preparation.