Specification and Generation of Displays for Complex Database Objects

Belinda B. Flynn
David Maier

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 92-011

March, 1992

Specification and Generation of Displays for Complex Database Objects

Belinda B. Flynn
David Maier

Abstract

The overall objective of this research is to provide support for producing interactive displays of complex database objects. In particular, the generated displays should be able to express the semantics and behavior of the underlying data, in order to promote the feeling of directly affecting the data as concrete objects. Our approach is to move display management from the application program to a display system that generates and operates displays, and directly accesses database objects according to user requests made through the displays. Thus, the application program is only concerned with how and when displays are created, not how they operate.

Designing the display facility involves two major parts: 1) the specification techniques used to describe the desired display, and 2) a runtime system that produces the displays from specifications. A main issue to address in the design is providing capabilities to describe how displays reflect object semantics, while still maintaining modularity between display specifications and the specification of data processing and manipulation. We identify three design features needed to meet our goals, then describe the facility's design and how it addresses those features.

1 Introduction

As with user interfaces in general, database user interfaces are turning towards a style in which the main mode of interaction is editing and browsing data objects – the interface produces the notion of directly affecting the data as if they were concrete objects. Providing this notion requires that the interface be able to express the semantics and behavior of the underlying objects. Several visual database interfaces [Goldman85, Bryce90, Leong89, Larson86] have adopted this style to enable users to access databases more effectively without extensive training or knowledge of the database schema. Domain-specific database applications most likely would receive similar benefits by having such interfaces. The overall objective of this research is to develop support for creating this type of user interface in applications that deal with complex database objects.

Presently, database applications that support such visual interfaces do so without much help from the DBMS. With record-oriented data models such as the network, hierarchical or relational models, the responsibility for displaying structured objects is typically associated with the application because the data in the database has a "flat" generalized structure (e.g., a relation), and thus the application is required to impose the connectivity or "object structure" on the data. Since the application realizes structured objects from data records, it is best suited to create and manage the object displays for the user interface. The application is therefore performing two transformations: 1) translating between records and structured objects, and 2) translating between data objects and their display structure, or in other words, between an internal and an external representation. In addition, the application most likely will be responsible for maintaining any integrity constraints concerning object structure.

Object-oriented databases (OODBs) provide the capability to model data directly as structured objects so that object construction need not be managed in applications and constraints regarding object structure can be associated with object classes. Since object structure is imposed by the database rather than the application, complex objects can be displayed directly, without any intervention by the application. Since the user interface has a direct link to the database objects, the application only needs to specify how and when displays are created, not how the displays operate. A display system could generate and manage interactive displays using only information stored in the database.

In short, our approach is to move display management from the application to a display system that directly accesses database objects. The database acts as a central resource for both the application program and the user interface. This arrangement can improve productivity due to the benefits of modularity:

- Display specifications can be developed more independently of the application program, and once some basic requirements are established, development of the application programs and their associated displays may be done in parallel.
- Displays are reusable among different applications that operate on the same database.
- Display implementation is less tightly coupled with the rest of application processing. When the two are tightly coupled, it is difficult to know exactly what parts of the code to extract for handling the display execution and constraint checking among input values.
- It is easier to experiment with different alternatives for displays in an application.

We have developed the Object Display Definition System (ODDS) to investigate our approach.

Related Work

There are several existing approaches to providing support for displays of complex database objects. Many database systems include application development tools (sometimes called 4th Generation Languages) that support the creation of form-based user interfaces. These tools generally provide a set of display building blocks, such as text-entry fields, radio buttons and menus, which can be included in the form, to be used for entering data or invoking database operations or queries. An example is the Forms Application Development System [Rowe89] for Ingres databases. A related approach is to integrate a user-interface toolkit or class library with the programming facilities of an object-oriented DBMS [Schmidt90]. Its main difference is that it seeks to produce a broader range of user interface styles than just forms.

Another related area involves display generation tools that produce browsers for object-oriented databases. These systems generate default displays for objects, based on the structure information kept in the object classes. The generated displays include predefined techniques for browsing, such as switching between displays with different levels of detail. These system typically allow the designer to customize the default displays through a language or graphics editor. Examples include LOOKS [Deux91], which is part of the programming environment for the O2 OODBMS, and KIVIEW [Laenens89], used with the KIWI OODBMS.

A substantial amount of research has been done on User Interface Management Systems (UIMSs). In general, the goal of these systems is to support declarative, high-level specification of user interfaces,

either through a language or a direct manipulation environment. However, relatively few of these systems are targeted at applications dealing with data that are stored in a central repository and managed separately from the application. Examples of these *data-oriented UIMSs* include HIGGENS [Hudson88] and the Serpent UIMS [Bass90]. Some of these systems are based on a data-dependency view of interactive applications [Garrett82]. In that view, many of the application's activities are expressed in terms of dependencies among data parameters that come from input devices, control graphical output, or are values within application objects. In general, a dependency is the combination of an *assertion* and an *action* to be performed when the assertion is true. In constraint-based systems, assertions and actions are implicit; whenever a value taking part in a constraint is updated, the underlying system acts to resatisfy the constraint. The exact form for expressing dependencies differs with each system.

2 Design Issues

In designing our system, we have discerned three critical design aspects: 1) the database model used for defining objects, 2) the amount of behavior a display definition captures, and 3) the responsiveness of display format with respect to changes in the underlying object structure. This section discusses alternatives for each aspect and which alternative we feel is necessary to achieve the desired level of support for object displays.

2.1 Database Model

Although all OODBs by definition support complex objects [Zdonik90], they differ in the extent that object behavior can be defined. Some database models, called *semantic database models* [Hull87], concentrate only on structural abstractions, e.g., aggregation, set grouping, and relationships. Other models provide for the expression of rules or constraints concerning the objects' semantics. One kind of rule describes the creation of *derived data* computed from existing data. Other rules are one-way constraints that state how changing a particular data item will cause values in other data items to be re-evaluated. These rules produce the notion of *active data* that can react to changes. Certain aspects of the objects' behavior can be described using these techniques; however, some systems using such models, e.g., the Cactis [Hudson90] and HiPAC [McCarthy89] DBMSs, do not express rules about an object's connections to other objects.

Behavioral models use the concept of abstract data types to associate general behavior with complex objects. An abstract data type includes an internal representation (which may be a complex structure) and a set of operations or *messages* used to interact with objects of that type. Unlike the previous categories of models, behavioral models enforce a strong view of encapsulation, meaning that the messages of a type do not necessarily allow direct access to the objects' internal representation. Examples of systems using behavioral models are GemStone [Butterworth91] and O2 [Deux91].

The choice of database model affects the capabilities of the display system because it determines what information the DBMS can provide to the display system regarding the objects being displayed. In particular, the display system should know about an object's structure, and needs to be informed of how an object is being updated during the time it is displayed. We choose to interface our display system to an OODB with a behavioral model. This choice provides the advantage that the information regarding the objects' integrity constraints and behavior can all be managed within the database. With the non-behavioral models, this management must be split between the database and the applications

that use the data. Besides complicating matters for the display system, such a distribution introduces the possibility that one application may violate some constraints being maintained by another application.

Another advantage of using a behavioral database model is that the message paradigm provides a mechanism for the display system to manipulate database objects. Within a display specification, one may specify that some message should be sent to a database object after a particular event or event sequence has occurred. Consequently, the application program does not need to be involved in every update of the database objects on behalf of the displays.

The user-interface tools that deal with a relational or partially behavioral database model receive limited kinds of information on the objects being displayed. The schema in a relational database holds information on the attributes of relations, but none on the complex structure of the database entities, because this information can only be determined by examining the data values within the relations. Similarly, the database models in data-oriented UIMSs typically notify displays of changes to attributes, but not changes in an object's connections to other objects. It may be possible to encode connectivity information within attributes, but this burden would be placed on the database designer. In addition, encoding one-to-many relationships would be cumbersome because these database models generally do not support objects with variable number of attributes [Hudson88].

2.2 Scope of Display Descriptions

The second design aspect is the extent to which display definitions capture the activities in the display, which in turn affects how well display management can be separated logically from an application. Different tools for building user interfaces capture various levels of display activity. At the lowest level, toolkits provide predefined interaction techniques (e.g., buttons, menus, gauges) and require that the designer build up displays by using a procedural programming language to combine the interaction techniques. Toolkits give no support for a logical separation for the code that manages the displays, as that code is allowed to be interspersed with the rest of the application code.

UIMSs capture more activity, since the designer can specify compositions of interaction techniques. The display descriptions are defined via a special-purpose language or environment, and thus are separate from the application code. Many UIMSs are based on the Seeheim architecture [Green85], which separates the user interface and application in such a way that the user interface is concerned only with interpreting user input, providing feedback as the input is parsed, and sending an appropriate request to the application. In this architecture, only the application has direct access to its objects and there is no explicit means, such as a data management component, to let the the user interface directly access information about the structure of the application's objects. Therefore, these UIMSs have some difficulties displaying application objects with complex structure. Typically, a change in one subpart of a complex object will require that its entire display be redrawn. Because of this overhead, the application will manage the semantic feedback in object displays instead of having the UIMS do it. The underlying reason for the difficulty is that although the UIMSs can specify compositions of display components, the display definitions cannot express the mapping of display subcomponents onto object subcomponents.

A third level of support would include the description of display behavior that reflects how the underlying objects have changed. Basic display responses include feedback on scalar data values such as a strings, numbers, and enumerations. Here, the feedback is determined by a simple conversion from the value to a graphical object. More complex responses involve some interpretation or intermediate processing of the updated values in the underlying objects. For example, a display change may be

triggered by certain conditions in object state, such as a list being empty or non-empty. Another case is some display change that occurs in addition to feedback on data values. For example, when a course is assigned to a classroom whose capacity is too small, the display might reflect the assignment, but also bring up a notifier stating that the course enrollment exceeds the classroom size. The specification of these kinds of display responses requires some way to reference particular parts of object structure. Data-oriented UIMSs have addressed this level of support by introducing a data model where the objects' structure can be defined. Naturally, ODDS should try to achieve this level of support since its objective is to manage displays of complex database objects.

2.3 Display Responsiveness

The third design aspect is the system's ability to produce displays whose structure is responsive to changes in underlying objects. Some systems support *static* representation, where displays are limited to a fixed template for all instances of a class, and only the basic data values in the display may change. Some systems allow *variable* representation so that an object can be displayed in various ways based **only** on conditions evaluated at the time the display is created. *Dynamic* representation calls for displays whose format may change during their lifetime, either to reflect the state of the underlying object, to respond to some user request, or to meet some space requirements. A *format change* of a display means altering the number or arrangement of display subcomponents, the graphics connecting subcomponents, or the format of any subcomponent.

Our objectives require that dynamic representation be supported because displays of complex objects should reflect changes in an object's structure as well as changes to basic values in an object. An object's structure can be viewed as a set of connections or references to other complex objects, where such a reference represents the fact that a semantic relationship holds between the referrer and referenced objects; e.g., a Course object references a Classroom object, to capture the "taught in" relationship. Structural changes that create or delete an object's connections to other objects could result respectively in adding or deleting the subcomponents of a display. In addition, replacing one of the object's references can result in changing the format of a display subcomponent. Such a change occurs if the newly referenced object has a different type. In many OODB models, the referenced object in a particular object connection is restricted to a specific type or subtype of that type. A type and its subtype(s) may have different display requirements. For example, suppose the type Laboratory is a subtype of Classroom. A Laboratory display could include additional information not applicable to a regular classroom.

In particular, the ability to reflect object structure is useful in database browsers that facilitate exploring, querying, and manipulating the database contents. For example, in the browsers generated by SIG [Maier86] and LOOKS [Plateau89], referenced objects are displayed using nested displays, so structural changes are shown when viewing data. Many browsers allow the user to control the level of detail at which objects are viewed. This "zooming" operation is another form of dynamic representation, although it is not tied to changes in the state of the object being displayed.

The declarative specification of dynamic representation has rarely been addressed. This aspect of display behavior generally must be specified procedurally, as part of the application program. To support dynamic representation, it is necessary to express 1) assertions about the connections between application objects and 2) actions that describe the rearrangement or replacement of subcomponents in a display. Data-dependency systems are mostly concerned with how display attributes are dependent on basic values, not how display structure depends on object structure. The assertion-action pairs in these

systems often constrain basic values in application and graphical objects, or maintain a link between an application object and graphical object.

To summarize, the features we desire for supporting interactive displays are:

- a behavioral model for defining complex objects,
- support for capturing display behavior that includes complex responses to database changes, and
- support for dynamic representation.

3 Towards Support for Direct Manipulation

As mentioned earlier, our objective is to support displays that provide a sense of direct interaction with the objects of interest. Two factors have been identified as being crucial to obtaining this feeling of directness in user interfaces: *direct engagement* and *reduction of distance* [Hutchins86]. An understanding of these factors provides a basis for seeing how the three desired features contribute towards producing a display system capable of supporting direct manipulation.

Direct engagement is the sense that the displays actually *are* the objects they represent, and that one is controlling the displayed objects through one's physical actions. This effect is produced by displays that exhibit some behavior, rather than merely being static output expressions printed on the screen. Direct engagement also requires a special relationship between input and output in which output expressions (i.e., the display images) may serve as components of input expressions. In other words, selecting or referencing an object's display is part of an input sequence that invokes an operation on the object. Since the input sequence results in changes to the display image, it appears to the user that referring to an object causes it to behave. Rapid feedback is also important to direct engagement, as it helps to hide the computer's role as an intermediary that is manipulating the display images on behalf of the user.

The other factor contributing to directness is the reduction of distance, where distance refers to the relationship between the user's task and the interface's mechanisms for dealing with that task. Thus, the notion of distance emphasizes that the directness of a display is always relative to the goals of the users. A short distance means that, for a given task, a small amount of cognitive effort is required to manipulate the objects as desired or to evaluate the results of one's actions. Since this effort is reduced, the interaction feels more direct.

Distance can be broken down into two levels. *Semantic distance* refers to the relationship between user goals and the meanings behind input and output expressions. With respect to input, the semantic distance of a user interface is bridged by providing commands that allow the user to accomplish a task in a concise manner. With respect to output, it is bridged by providing feedback information that allows the user to evaluate readily whether the desired goal has been achieved. *Articulatory distance* refers to the relationship between the meanings of expressions and their physical forms; i.e., it indicates how well the form of an (input or output) expression reflects its intended meaning. Examples of forms for input expressions include sequences of mouse movements, mouse button clicks, and key presses. Forms for output expressions may be character strings, bitmap images, sounds, or the composition of several output expressions. Articulatory distance is reduced by choosing the appropriate physical forms for requesting a given operation and for representing the concepts and objects of the application domain.

The three design features from the previous section are needed to support the specification of display properties that foster these two factors. In support of direct engagement, the use of behavioral models supplies a more complete account of the objects' behavior, which is necessary to update the display in such a way that it appears to be the object it represents. These display behaviors will most likely involve complex display responses (such as those described in Section 2.2), as well as changes in the display's format. Thus it is important to be able to specify such responses through a display facility.

The display capabilities that contribute to reducing distance are mostly dependent on the subject domain and the tasks to be performed. However, reducing semantic distance in a display's output often requires presenting information that is not part of any object's state, but is computed from values in the state of an object or group of objects. The message paradigm in behavioral database models can be used to provide access to the computation for deriving this required information. Some support for reducing articulatory distance can be provided if we consider a particular meaning for output expressions. When the intent of output is to reveal object structure, any display changes regarding the connections among the underlying objects contributes to reducing articulatory distance. Dynamic representation enables a display to adust its format to match changes in an object's connections. These changes may also be manifested through notifiers or as effects on the set of objects currently displayed.

4 The Display Facility and Its Architecture

Having discussed key features concerning the expressiveness of the display specifications, we turn to design decisions concerning their form. First, we chose to have the displays described by declarative specifications, and to generate the displays from the specifications. With declarative specifications, the designer can indicate the desired functionality without being overly concerned with how it is accomplished. Since our primary motivation is to improve productivity and reduce extraneous effort, the designer should be able to work at a level of abstraction that matches the design task, rather than being concerned with the details of a conventional programming language. For this reason, we did not take the alternative of defining displays using a procedural specification, i.e. source code. In addition, choosing a particular programming language can introduce restrictions on what implementation language may be used for the application program.

Another alternative was to create displays within a direct manipulation environment, by graphically composing display components, and possibly using graphical symbols to represent computation or inter-communication between display subparts. Fabrik [Ingalls88] and HyperCard are examples of such environments. A problem with this approach is that temporal aspects of the display are not easily expressed, making it difficult to describe how displays will reflect changes in object structure during execution.

Because of our choice to use declarative specifications, designing the display facility involves two major parts: 1) the specification techniques used to describe the desired end-product, i.e., the display, and 2) a runtime system that produces the displays from specifications.

A second key decision was to store display specifications in the database along with the objects being displayed. Thus the display specifications are themselves database objects, belonging to the class *Outline*. An Outline is used to generate displays for objects of a particular class, called its *source class*.[1] We call our specifications Outlines because they do not completely describe displays of individual objects.

[1] The specification can also be used to display instances of the subclasses of its source class.

Rather, they are templates or partial descriptions that lack the actual data values that would come from the object being displayed. An Outline basically consists of two kinds of information: 1) a description of the display's graphical image and 2) a behavioral description; i.e., the display's responses to user-input events, including visual feedback and message passing to underlying database objects. Accordingly, there are two hierarchies of specification classes, the *Layout* and *Interaction* classes, whose instances are used within an Outline and are called Layout and Interaction *specs*.

The main reason for making specifications database objects was to make them accessible to, and sharable by, all database applications as well as the display runtime system. The specifications need to be available to the runtime system during display execution, so it can obtain information on how to handle the displays' format changes. If each application were to hold its own specifications and provide them to the runtime system for interpretation, then the specifications would not be available to other applications. A possible alternative would be to store the specifications in files so that they are sharable among applications. However, this approach would duplicate some data management capabilities already present in the DBMS. One significant capability is being able to model complex objects that can be referenced from several other objects. This capability is often needed to express the complex structure of display specifications, and it also allows reusability in the sense that different display specifications may share a common subpart.

Several other advantages are gained by making display specifications part of the database. The specifications may be associated with class definitions, thus providing a more comprehensive view of the objects' semantics. Multiple specifications may be assigned to a particular class so that an instance can be displayed differently depending on the context of the display. This capability is often needed for objects with complex semantics, because a single representation generally will not be appropriate for all possible operations on an object.

Another advantage is that the specifications can be examined, modified, and displayed just as other objects are [Anderson86]. An interactive editor for creating new specifications could be built as a database application that uses the display specifications associated with the Outline, Layout and Interaction classes. In addition, the specification objects will be available to other database tools that assist with application programming.

The remainder of this section gives an overview of the specification techniques and the runtime system, placing emphasis on those elements that serve in achieving the three desired features listed in Section 2.

4.1 System Architecture

Figure 1 shows the architecture of ODDS's runtime system in relation to the database and the application program. The runtime system is made up of the components shown in ovals. Our prototype implementation uses the GemStone DBMS, which has a behavioral database model that is similar to the object model in Smalltalk. The runtime system is implemented in Smalltalk, with the exception of the Source-Update Manager, which is implemented within GemStone.

The database acts as a central resource and a means of communication for the display runtime system and the application program. Since both sides have read and write access to the database objects being displayed, the modifications made by one side are observable to the other side. The application might act based on changes made by the runtime system, and the runtime can update the displays to reflect changes made by the application. As a result, the two sides communicate indirectly through modifications to

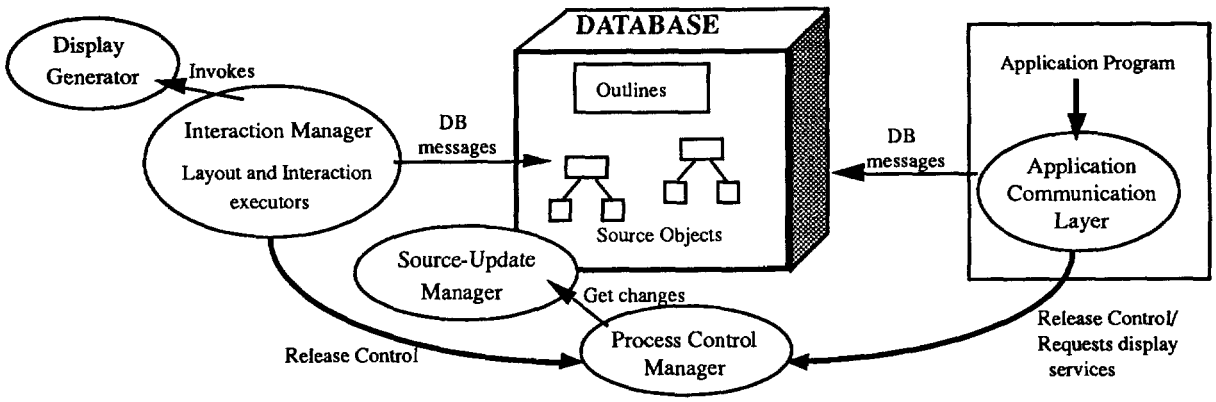

Figure 1: Components in System Architecture

the database objects.

With respect to control flow, the runtime system and the application program cooperate as coroutines; i.e., each will voluntarily give up control to the other at certain points in its execution. The Process Control Manager handles this exchange of control, hiding the details of process communication from the other components. Thus the rest of the runtime system is not concerned with whether the application is a local process or on a remote machine, or whether it is a C or Smalltalk application.

The runtime system cedes control as directed by a display's specification. The specification framework provides a special object to be used in Outlines to indicate that control should return to the application. The application grants control to the runtime system for several purposes: to create or close displays, to hide or unhide displays, or to resume operation of the displays. The application can also temporarily release control to display some intermediate effects of a computation, and regain control after the displays have been refreshed, without any further action by the runtime system. The application makes these requests by sending messages to the Application Communication Layer, which forwards the requests to the Process Control Manager. The services provided through the Application Communication Layer could be viewed as an extension to the existing set of functionalities that a DBMS provides to applications. Thus, the ability to display objects becomes another aspect of data management.

When the application requests that a display be created, it supplies the name of an Outline and the database object to be displayed, which we call the *source object*. This information is passed to the Display Generator, which generates a set of Smalltalk objects called *executors* that draw the screen images and carry out the behaviors as defined in the Outline. To construct executors, data from the source object is merged with the Outline's partial descriptions, producing a complete description of the source object's display. The structure and content of the generated executors essentially parallels that of the specs in the Outline. Thus, the classes of executors, called *LayoutExec* and *InteractionExec* classes, have an exact correspondence with the Layout and Interaction specification classes. A key difference between the executors and the specs is that the executors are non-persistent objects known only to the runtime system.

The Interaction Manager is responsible for the overall execution of the displays. Most of its work is performed by the executors produced by the Display Generator. The methods of the executors are designed to interpret information copied from its spec counterpart so that the executor will carry out the semantics expressed by the spec object. Work that is performed outside of the execution objects includes

processing the events originating from input devices and from database changes, and routing them to the appropriate Interaction executor, which will act based on the new event. Finally, the Interaction Manager will need to activate the Display Generator when a display format is to be altered. The Display Generator evaluates the tests that choose between formats of the display. It obtains the appropriate spec describing the new format, then reconfigures the display.

The Source-Update Manager monitors the source objects currently being displayed, and produces a list of updated source objects, the particular instance variables that were affected, and the object identifiers for the new values of those instance variables. Thus, the Source-Update Manager supplies information that the display needs to produce semantic feedback. The Process Control Manager requests this information whenever there is an exchange of control between the runtime system and the application, and passes the information to the one that has just regained control, so that it may react to the changes. In addition, this information is requested and passed to the Interaction Manager after a database message has been sent, so that changes that occured during the message's execution may be reflected in the displays.

4.2 Specification Capabilities

Since an Outline does not contain the actual data values to be displayed, it instead contains special 'placeholder' objects called *Paths*, which indicate a message or message sequence that will return a particular part of a source object's state. An object class may have several Outlines that display its instances differently and may also indicate a default Outline to be used in cases where no specific Outline is requested when an instance is displayed.

An Outline may contain objects called *Deferments* to specify that another Outline will be used to generate a subpart of the display. Rather than having an Outline refer directly to another Outline, Deferments provide more flexibility in indicating which Outline is to be used. A Deferment can either indicate a particular Outline by name, provide a test to choose among several Outlines, or just use the default Outline for the appropriate class.

4.2.1 Classes for Layout Specification

Layout specs define a display's visual image on the screen. Each Layout spec holds a set of attributes that determine graphical details such as foreground and background colors, text fonts, and spacing. At present, the primitive components available for display images are text strings, bitmaps, lines, and rectangles; different types of Layout specs exist for each of these primitives. A *ComposerLayout* is a Layout spec that indicates a composition of other Layout specs. Specific kinds of ComposerLayouts, such as *Above*, *Beside*, and *Around* specs, specify how the display image is spatially composed from its subparts.

A *Correspondence* spec is a Layout spec whose instances describe the visual representation for object relationships. We say that a *domain* object `x` is related to a *range* object `y` if there is some message in `x`'s protocol that when sent to `x` returns `y`. A Correspondence is used to present a mapping from a collection of domain objects to another collection holding potential range objects with respect to a particular message. An example is a mapping from a set of Course objects to a set of Classroom objects, where the domain and range objects are related by the `taught_in` message. The visual representation for a relationship may be in the form of lines connecting the displays of the related objects, or the displays might be positioned next to each other to symbolize the relationship. Correspondence objects provide

an alternative to the customary way of expressing relationships, which is to nest the display of the range object within the display of the domain object.

4.2.2 Classes for Behavioral Specification

An Interaction spec can reference Layout specs and define what changes will take place in the executor counterparts for those Layout specs. In effect, an Interaction spec defines new images to replace the present image of the display, or parts of it. There are three general kinds of behavior to be specified: 1) image changes based on user input, 2) image changes based on activity in the database, and 3) the composition of the first two kinds to form more complex behavior.

Event Types and Response Actions

In general, behavior is expressed in terms of *event types* and *event responses*. Examples of events are the arrival of data through input devices, the occurrence of a certain condition or state in a display component (e.g., when a button is designated as being selected), or an update to some source object. Events can be generated by user input, by database changes, or by Interaction executors.

An event response describes the action or sequence of actions that will take place whenever a certain event occurs. The types of actions described in an Interaction spec include: 1) updates to a Layout executor, 2) updates to its Interaction executor counterpart, which will affect the future behavior of the display, or 3) the generation of *internal events* that will be forwarded to another Interaction executor.

A *MatchMaker* spec describes a response action that operates on a Layout or Interaction executor. The operation is defined using in two parts: the Matcher includes one or more templates defining the arguments participating the operation, and the Maker is a template defining the objects to be modified. The Matcher includes at most three templates: an Interaction template, a Layout template, and an Event template. These templates are used to express that during display execution, the arguments for the operation will come from within the Interaction executor, its Layout, or the event that triggered the response action, respectively. A third component in a MatchMaker spec is a map connecting the two Matcher and Maker, which describes how data values or objects in the arguments are placed into the modified objects, or are used in some computation whose result is then placed in the modified objects.

To illustrate, the MatchMaker below defines an operation where the foreground and background colors of a Layout executor are reversed. The underlined class names signify an object template is being represented, rather than an instance of that class. The *object tags* in our notation (**LA**, **FC**, and **BC**) represent the map connections in the MatchMaker; an object tag present in both the Matcher and Maker represents a connection between the two.

Matcher:
Layout(attributes →
    LayoutAttribute **LA**:(foreColor → **FC**: Object
        backColor → **BC**: Object))
Maker:
LayoutAttribute **LA**:(foreColor → **BC**: Object
    backColor → **FC**: Object)

In this example, the object tags in the Matcher specify the values from the Layout executor which will be involved in the operation. The Maker specifies that the LayoutAttribute executor bound to

object tag `LA` will be updated. Specifically, the `foreColor` and `backColor` fields of the LayoutAttribute executor will be updated with the executors bound to tags `BC` and `FC`, thus reversing the background and foreground colors of the Layout executor.

MatchMakers may optionally contain a computation component, specified by a *ComputeAction* spec. A ComputeAction holds a name associated with a block of code that must be registered with the runtime system, where it is kept in a table called the Computation Library. The ComputeAction also defines the values or the locations of the arguments to the computation block. An argument location is specified by a reference into the Matcher. Similarly, the ComputeAction specifies the location for the computation's result, using a reference into the Maker template.

Another type of activity described in Interaction specs is the change in a display's format during its execution, expressed through a combination of specs. A *ChoiceMap* spec defines the set of alternative formats, while a *FormatAction* determines how an alternative is selected. The concept of interpreting specifications to execute format changes is based on an earlier system, the Smalltalk Interaction Generator [Maier86].

A ChoiceMap holds a dictionary that maps a *lead value* to a particular format alternative. The ChoiceMap also defines which alternative to use when the executors for a display are initially generated, either by a direct reference to the desired alternative, or indicating a subpart of the source object, whose value will be the lead value used to select an alternative. The change in format is specified using a FormatAction spec, which describes a response action in an event mapping. A FormatAction defines a computation or database message whose result produces a lead value into the ChoiceMap's dictionary.

The different format alternatives may share subcomponents specs; for example, they may arrange the same subcomponents in different ways, or they may differ only in the graphics that connect the subcomponents. Thus, there is an opportunity for re-using executors at runtime if the subcomponents of a format alternative is shared with one for which executors have already been generated.

Interaction Classes

The Interaction subclasses provide a framework in which to structure or organize the collection of events and event responses that make up a display's behavior. Each Interaction spec holds a mapping from expected event types to the responses for events of a given type. There are three main Interaction subclasses, described below.

ImageOp specs group together the events and responses of interaction techniques such as menu buttons or gauges. The responses they describe consist mainly of updates to Layout executors.

DBConnect specs cover behavior that reflects the activity in the database source objects, as well as behavior where the display initiates operations on database objects. Their event mappings hold event types related to changes in source objects. The response actions in DBConnect specs specify how the information on database changes is used in updating Layout executors. A response action in a DBConnect spec may also be a *MessageAction* spec, which holds the name of a database message to send to a source object, plus the location of any arguments required.

Coordinator specs group the events that define the communication and data transfer among several ImageOp or DBConnect specs. Thus, Coordinators describe the behavior of a complex interaction technique, e.g., a panel of radio buttons, by composing the behaviors of basic interaction techniques. A Coordinator holds a collection of *sub-Interaction* specs whose executors will be communicating at runtime. The event mapping of a Coordinator specifies which sub-Interaction executor will receive

internal events of a particular type. Coordinator specs can also be used to compose a collection of DBConnects as well as a combination of DBConnects and ImageOps.

4.3 Addressing Desired Features

We point out characteristics of the Outlines and the system architecture that address the desired features set forth in Section 2.

Support for and advantages of using a behavioral database model:

- Within the system architecture, the Source-Update Manager obtains the information that the displays require to reflect the source objects' state, including information on changes to an object's connections to other objects.
- MessageAction specs express how the source objects are to be manipulated through the displays.
- Information about how to display database objects is also within the database, and becomes part of the objects' semantics, as opposed to being isolated in individual applications. As a result, the ability to display objects becomes another aspect of data management.

Support for specification of display responses to database changes:

- Through the use of Paths, Outlines can refer to specific parts of a source object, allowing the specification to indicate how the subparts of a display are mapped to the objects' subparts.
- DBConnects specify how updates in source objects will affect the Layout executors that draw the screen images. The ways in which the Layout executors can be affected are not limited to the replacement of basic values; e.g., updating text strings. The updated values may be used as arguments for some computation whose results in turn are able to affect various aspects of Layout objects, such as colors or the positioning of display elements.
- The Correspondence specs provide the ability to associate some graphics or other display aspects with the semantic relationships among the source objects. This capability gives the display designer more flexibility in presenting object structure.

Support for dynamic representation:

- FormatActions are used to describe the possible formats of a display, the conditions for switching between formats, and in some cases, the criteria for choosing among several possible formats.
- The Display Generator serves to rebuild displays according to directions in the FormatAction objects during the execution of the display.

4.4 Example Outline Specification

A small example is given here to provide a more concrete picture of the specs in an Outline. This Outline defines a display for a Course object, showing its title and number of credit hours (see Figure 2). Each boxed area containing a data value is to be highlighted by switching the foreground and background colors whenever the cursor enters it. Another behavior within the display is that a mouse click within the

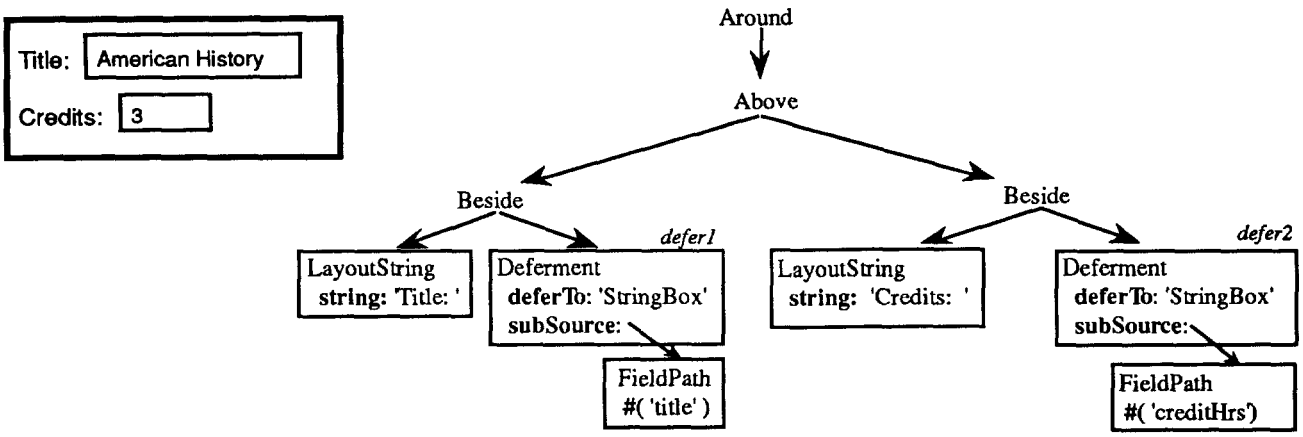

Figure 2: Sample Display and Layout Specification

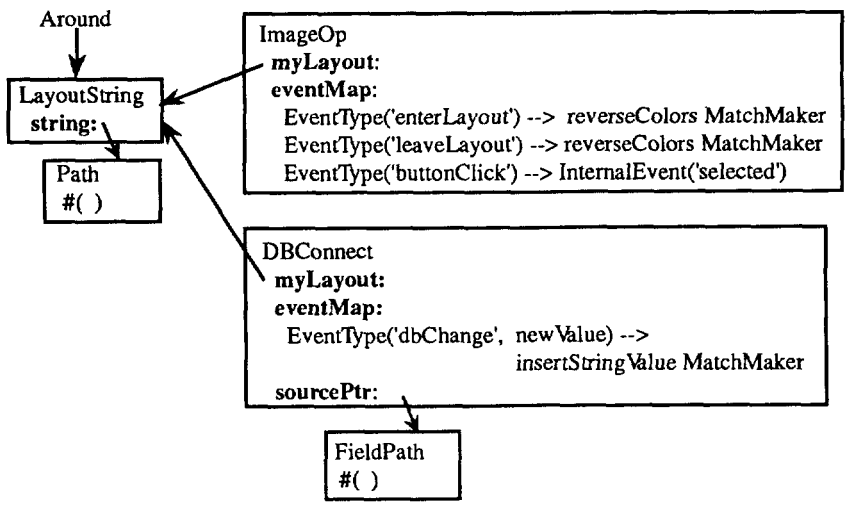

Figure 3: Outline for StringBox Display

Credits subdisplay causes the value there to be incremented, modulo some maximum allowable number of hours.

Figure 2 also shows the basic structure for the Outline's Layout spec. (Layout attributes have been omitted for simplicity.) The arrows in the diagram indicate the subcomponents for a ComposerLayout. The two subdisplays holding the data values are generated from a separate Outline named StringBox, as specified by the Deferments. Deferments are defined with two parameters: `deferTo` holds the name of the Outline to use for the subdisplay and `subSource` indicates which object subpart should be displayed.

The Layout and Interaction specs for the StringBox Outline are shown in Figure 3. Within the context of a LayoutString, a Path determines which data value from the source object is to be inserted into the Layout executor at runtime. The behavioral specification for the Outline consists of two parts:

- An ImageOp specifies the highlighting behavior and the response for a mouse click via the three associations in the event mapping.

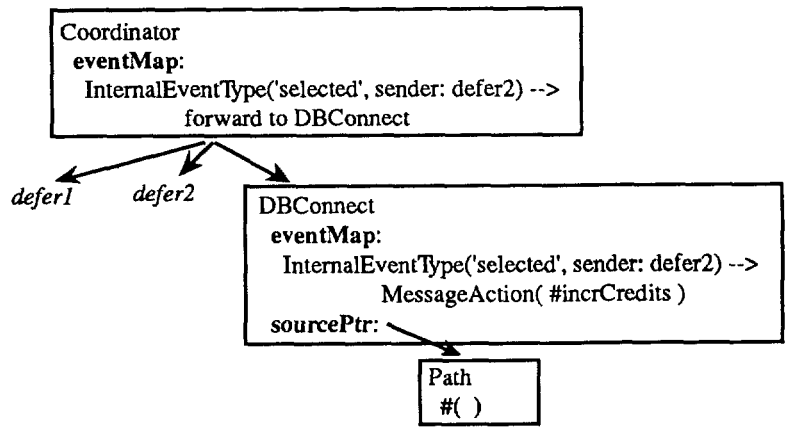

Figure 4: Interaction Objects for Course Outline

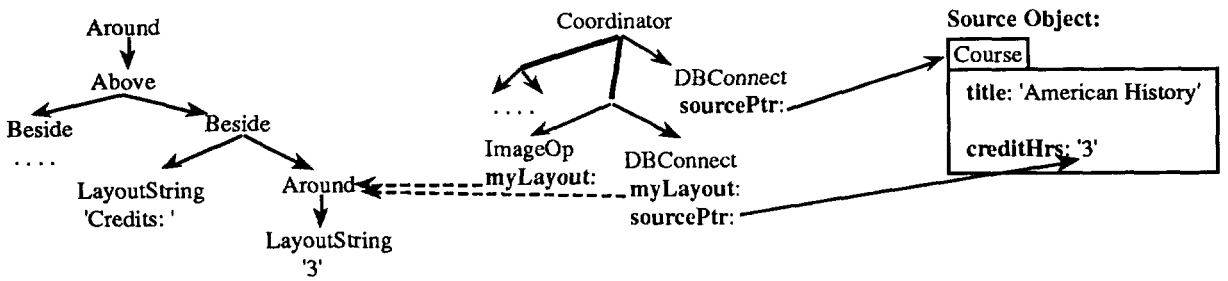

Figure 5: Source Object and Runtime Objects for Course Display

- A DBConnect defines the connection between the LayoutString spec and the source object being displayed. Its event mapping specifies that when an arriving event indicates a change in the source object, the new value (carried in the event) is placed into the `string` field of the LayoutString executor.

The behavior portion for the Course Outline (Figure 4) must specify that the number of credit hours of the Course is incremented when the Credits subdisplay has been selected with a mouse click. The Coordinator's event mapping states that at runtime, the 'selected' event from the that subdisplay will be forwarded to the DBConnect executor that holds a reference to the Course object. The DBConnect spec specifies that the response to this internal event will be to send a database message that updates the number of credit hours.

Figure 5 shows the basic structure for the runtime executors generated from the Course outline. The Deferments in the Course Outline are replaced with executors generated from the StringBox Outline. Paths in the DBConnect specs are replaced with references to the source objects, which are needed to send database messages to those source objects.

5 Summary and Current Implementation

To summarize, we have designed the specification framework and runtime architecture of ODDS to meet the desired features set forth in Section 2. ODDS provides mechanisms for interacting with a behavioral

database model to manipulate source objects and to obtain information for semantic feedback. The Path, DBConnect, Coordinator, and Correspondence specs are particularly useful in specifying complex display responses to database changes. Dynamic representation is supported through the FormatAction specs and the execution of format changes by the Display Generator.

We have devised a sample set of displays which will be specified and generated through ODDS, so that we may evaluate the expressiveness of the specification classes as well as the performance and resource usage of the runtime system. We also plan to examine the extent to which Outlines are reused through Deferments or by borrowing Layout and Interaction specs from existing Outlines.

The prototype implementation for ODDS is near completion. The specification classes and corresponding executor classes describe in this paper have been implemented and used to produce a number of the test displays. One of the remaining tasks is implementing mechanisms to set and use the default Outline for a class. Another area requiring further work is the set of services available to application programs. Currently, the runtime system serves only one application at a time. Additional services to provide to the application include controlling the visibility of displays and arranging for certain application procedures to be invoked from Interaction executors.

References

[Anderson86] T.L. Anderson, E.F. Ecklund, Jr., and D. Maier, "PROTEUS: Objectifying the DBMS User Interface", *Proceedings of the International Workshop on Object-Oriented Database Systems*, ed. D. Dittrich and U. Dayal, Pacific Grove, California, September, 1986.

[Bass90] L. Bass, E. Hardy, R. Little, and R. Seacord, "Incremental Development of User Interfaces", *Engineering the Human-Computer Interface*, A. Cockton, ed., North Holland, 1990.

[Bryce90] D. Bryce and R. Hull, "SNAP: A Graphics-Based Schema Manager", *Readings in Object-Oriented Database Systems*, ed. S. Zdonik and D. Maier, Morgan Kaufmann Publishers, 1990.

[Butterworth91] P. Butterworth, A. Otis, and J. Stein, "The GemStone Object Database Management System", *Communications of the ACM*, Vol. 34, No.10, October 1991.

[Deux91] O. Deux et. al., "The O2 System", *Communications of the ACM*, Vol. 34, No.10, October 1991.

[Garrett82] M. Garrett and J. Foley, "Graphics Programming Using a Database System", *ACM Transactions on Graphics*, April 1982.

[Goldman85] K. Goldman, S. Goldman, P.Kanellakis, and S. Zdonik, "ISIS: Interface for a Semantic Information System", *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data*, Austin, Texas, May 1985.

[Green85] M. Green, "Design Notations and User Interface Management Systems", *User Interface Management Systems*, Eurographics, 1985.

[Hutchins86] E. Hutchins, J. Hollan, D. Norman, "Direct Manipulation Interfaces", *User Centered System Design*, ed. D. Norman and S. Draper, Lawrence Erlbaum Associates, Inc., 1986.

[Hudson88] S. Hudson and R. King, "Semantic Feedback in the Higgens UIMS", *IEEE Transactions of Software Engineering*, Vol. 14, No. 8, August 1988.

[Ingalls88] D. Ingalls, S. Wallace, Y. Chow, F. Ludolph, K. Doyle, "Fabrik: A Visual Programming Environment", *OOPSLA '88 Proceedings*, September 1988.

[Laenens89] E. Laenens, F.Staes and D. Vermeir, "Browsing a la carte in Object-Oriented Databases", *The Computer Journal*, August 1989.

[Larson86] J. Larson, "A Visual Approach to Browsing in a Database Environment", *IEEE Computer*, June 1986.

[Leong89] M.K. Leong, S. Sam, and D. Narasimhalu, "Towards a Visual Language for an Object-Oriented Multi-Media Database System", *Visual Database Systems*, ed. T. L. Kunii, Elsevier Science Publishers, 1989.

[Maier86] D. Maier, P. Nordquist and M. Grossman, "Displaying Database Objects", *First International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986; also OGI Technical Report CSE-86-001.

[Myers90] B. Myers, ed., "The Garnet Compendium: Collected Papers, 1989-1990", CMU-CS-90-154, Carnegie Mellon University, August 1990.

[Rowe89] L. Rowe, "Database Representations for Programs", *Proceedings of 1989 ACM SIGMOD Workshop on Software CAD Databases*, Napa, California, February 1989.

[Schmidt90] D. Schmidt, "From Object-Oriented Database Systems to High Productivity Software Development Environments", OGI Technical Report, July 1990.