18

Database object display definition and management with Moggetto

Pete Sawyer, Andy Colebourne, John A. Mariani and Ian Sommerville
Lancaster University
Lancaster, LA1 4YR, UK. tel. +44 1524 65201, fax +44 1524 593608,
[sawyer, andy, jam, is]@comp.lancs.ac.uk

Abstract

This paper describes a user interface framework called Moggetto for an object-oriented database system (OODB). Moggetto is proposed as an approach to the definition and management of tailorable object displays. The novelty of the approach lies in the use of embedded, inherently reconfigurable user interface components to realise object user interfaces which are modelled in the database. This obviates the need for close coupling between the OODB and the host windowing system through, for example, common language bindings and code libraries.

Keywords

User interface framework, database user interface management system, user interface components, MOG objects

1 INTRODUCTION

The object model is frequently espoused for database applications where richer semantic modelling capabilities than those offered by the traditional database models are required. Most of the older database models, exemplified by the relational model, are designed for applications where the data can be modelled using relatively few types but many instances; in other words the bulk of the data has a relatively uniform structure. Object-oriented database systems (OODBs), by contrast, are a response to the problems of supporting applications where the data is more heterogeneous. The consequent lack of uniformity of structure and semantics among the data has important repercussions for OODB user interfaces.

One of these is the mode of searching and retrieval of data. Querying via declarative query languages and form-filling is the dominant means of interacting with relational systems. With OODBs, querying alone is insufficient because a mechanism to facilitate the exploration of the interrelationships between objects is also needed. This is typically provided by a graphical browser.

A closely related issue is how the objects are visualised once they have been located by a query or by browsing. OODB objects are structurally complex compared to relational tuples. For example, they may be composed of other objects requiring a means to reveal or conceal multiple levels of composition as required. They also encapsulate behaviour in the form of

S. Spaccapietra et al. (eds.), *Visual Database Systems 3*

methods and perhaps constraints and triggers (e.g. (Sawyer 1994)). These factors mean that the structure of the data (or a stylised representation of it) needs to be made explicit to the user. Users need visual clues to recognise objects of different types and how to interact with, or reveal new or related information about, those objects.

The provision of effective browsers and means to visualise the objects of interest are necessary for end-users of OODBs. Equally important are tools for database designers. As observed by Almarode (Almarode 1991):

"*With the increased complexity of inheritance, behaviour, and complex objects in the database, building such tools for an object-oriented database management system will provide a challenge for all researchers and implementors*"

Traditionally, the schema of a database is defined by hand coding using a data definition language (DDL), but attention has shifted to the use of graphical interactive tools to assist with this (e.g. (Leong 1989), (Almarode 1991)). The dynamic nature of some OODBs, where the schema can evolve without recompilation of the database, has encouraged the development of such tools.

This paper describes an OODB user interface framework called Moggetto which has been developed for the Oggetto OODB (Mariani 1992) to explore these issues. Moggetto provides:

- A run-time environment to manage the display of, and interaction with, database objects.
- A means for the interactive definition of objects' displays.
- A means for the interactive definition of the object types which constitute the database schema.
- Tools for browsing and querying.

We focus on the first three points which are novel because they have been achieved through the integration of interactive user interface components with the database schema. These are MOG objects (Colebourne 1993) which embody editing as well as end-user even handling. When integrated with the database through the framework, the information about how objects are to be displayed can be stored within the database. The major advantages of this approach are:

- A computationally rich DDL such as would be necessary if objects' user interfaces were to be defined as methods is not required. Consensus on OODB DDLs, from persistent languages to more traditional and less complete DDLs (Andrews 1987, Atkinson 1988), is currently lacking. By minimising the dependence on DDL computational completeness, Moggetto provides a generic mechanism for OODB user interface management.
- Code to manage objects' user interfaces but defined in libraries external to, and outside the control of, the database management system is not required. Storing objects' user interface definitions in the database is safer because evolution of the schema and user interface can be more easily managed and kept consistent. Similarly, the definition of new object types and their user interfaces does not require that new code libraries are generated or that they be compiled and linked with the database management system.

The remainder of this paper is structured as follows. The next section discusses the user interface tool and run-time features to be expected of an OODB and introduces the notion of an OODB user interface management system (UIMS). Section three describes the Moggetto framework in terms of an OODB UIMS. Section four describes how user interfaces are defined in Moggetto, including a description of the MOG augmented widget set and how it has been integrated with the Moggetto display manager. Section five describes object type definition in Moggetto and section six describes how Moggetto user interfaces are modelled in the database. Section seven concludes the paper.

2 DATABASE SYSTEM USER INTERFACES

In many database systems, the dominant means of user interaction is through a generic user interface where common display and interaction mechanisms are applied to every entity. In the case of relational systems, for example, every relational table can be easily mapped onto a tabular form on the user's display. Example-based user interfaces exemplify these (Özsoyoglu 1993) and many variants on the original QBE (Zloof 1975) are in existence. Tabular representation of database entities have the advantage of reducing the syntactic information which users must retain in order to interact with the database (Yen 1993). Visual query languages (Batini 1992) represent an attempt to ease the user's semantic load as well. An iconic query language, for example, will display entities according to entity-type-specific graphical icons which are deemed visually meaningful in the context of the user's application domain.

The approach of mapping every entity to a graphical representation according to a fixed set of rules has also been used for OODBs (e.g. dynamic forms (Sawyer 1988) and FO2 (Collet 1992)). To be effective where the schema and objects' structural complexity is relatively high, however, these must support browsing in addition to querying. Hence, for example, the means to follow relationships between objects and components, and types and sub/supertypes are needed.

Entity-type-specific representations are especially useful in OODBs where there are a relatively large number of different object types, each embodying different structure and semantics. Iconic representations alone are inadequate because they are atomic graphical entities and integrating navigational operations at the level of object components (e.g. expanding an attribute of a complex type) is difficult. Displays which visually distinguish the differences between object types but which allow the user to inspect and interact with the properties of which the objects are composed are needed to visualise objects.

In summary, an OODB user interface system should provide tools for the definition of objects' user interfaces and a run-time environment. Such a system may be described as an OODB User Interface Management System (UIMS).

2.1 UIMS and OODBs

It is interesting to consider user interfaces to database systems in relation to the Seeheim run-time model (Pfaff 1983) (figure 1) consisting of presentation, dialogue control and application interface components. Broadly, these components' responsibilities are as follows:

- Presentation refers to the display of the entities with which the user is interacting and the trapping of user input.
- Dialogue control manages the user interface-specific functionality (as opposed to that of the underlying application functionality). For example, triggering the display of a pull-down menu in response to a particular input event.
- The application interface forms the interface between the application and the user interface. It often takes the form of the sub-set of application entities (objects, functions) which need to be accessed by the user interface code.

Communication between these components is linear in nature so all input and output is routed between the presentation and application interface components via the dialogue controller.

For many application domains, the Seeheim model, while providing a useful conceptual decomposition of roles within the UIMS, has proven to be unsatisfactory as a run-time architecture. In particular, the strict decoupling of application and user interface functionality imposes performance problems where high levels of semantic feedback are required (the so-called bandwidth problem). This is a serious impediment to highly interactive, direct

manipulation user interfaces (such as drawing packages) where every mouse movement event might require a user interface - application - user interface communication cycle. In terms of UIMS to databases, however, the granularity of communication is typically much coarser. In any case, the practicalities of the secure management of large volumes of data mean the database management system is a far more significant performance bottleneck than the user interface's run-time architecture.

Figure 1 The Seeheim model.

In a UIMS supporting graphical user interfaces to an OODB, the three Seeheim components may be broadly characterised as follows: The presentation and dialogue control components are much as described above, where the entities with which the user interacts are database objects. The application interface is the set of query operations provided by the OODB management system and through which the user interface communicates with the database. Hence, user events are mapped by the dialogue controller onto queries on the state of the database or requests to update the database's state.

DBface (King 1993) is a good example of a Seeheim-based database UIMS. Here, the display of database entities is managed by a *representational* component and the user interface functionality is managed by an *operational* component. These correspond approximately to the Seeheim presentation and dialogue control components. A novel feature of DBface is that the database acts, not only as the repository of the underlying application data, but also as the repository of the information used by the representational component to display the data.

DBface represents an attempt to provide a run-time architecture for the support of OODB user interfaces. This is broadly the same goal as Moggetto. Where Moggetto differs is in the focus of our work within the general area of OODB user interface support. We are interested in providing highly flexible, tailorable user interfaces and user interface development tools for database users.

3 THE MOGGETTO USER INTERFACE FRAMEWORK

Like DBface, Moggetto adopts a Seeheim-like architecture (figure 2). The display manager corresponds to the Seeheim presentation component, rendering objects' displays and capturing user events. The definition of objects' displays are contained in the database and consulted when a request to display an object is received. The object store interface corresponds to the Seeheim dialogue control component, maintaining a model of the state of the user interface. Valid events are mapped onto database queries and the results returned from the database are mapped onto appropriate update messages passed to the display manager to ensure that consistency is maintained between the display and the database.

Figure 3 illustrates Moggetto being used to view an Oggetto database modelling the components of a computer network. It is presented here as context for the discussion on defining and manipulating object displays which follows, not to suggest that, for example, the browser illustrated is appropriate for every OODB. Three important features of the Moggetto framework are illustrated in figure 3:

1 The background window labelled "Type hierarchy" is the top-level view of the database provided by the display manager browser. When invoked, this consults the database to construct the representation of the Oggetto inheritance hierarchy. The nodes represent

Oggetto object types; types illustrated are subtypes of the type to which they are connected immediately to their left. Object is the object type at the root of the hierarchy. Interaction with objects is through this top-level database view. For example, by selecting a node, the user is presented with the options of viewing instances of the type (see 2 below), viewing the type itself (see 3 below) or deriving a new type as a subtype of the currently selected node.

2 In the example, the user has opted to view instances of the type "SE30" and the small window labelled "Instance list" lists the instances of this type which exist in the database. Selecting one of these, as has been done to "PetesMac", causes the object to be displayed. The display of this object is managed at run-time by an object librarian which is created to collate the object's structure (type), value (instance) and display (user interface) details from the database. This information is then forwarded to the display manager for the actual rendering of the object on the screen.

3 Also displayed in figure 3 is the *type* SE30. This is actually the template used for displaying instances of the type. As the user interface of an Oggetto type may be a projection over a type, the properties displayed may be a subset of those belonging to the type. Hence, it can be thought of as a graphical visualisation of that subset of the type which is required to be made visible to users of instances of the type.

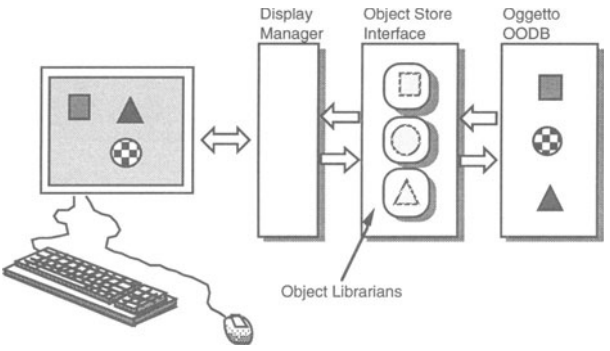

Figure 2 Moggetto's architecture.

Note that the SE30 type includes attributes of both primitive and Oggetto types, and singleton and multi-value (set) attributes. For example, "cpu" is a simple string while "pic" is a sequence of video images. The mechanisms used to visualise these are different with the value(s) of pic being mapped onto a composite user interface object incorporating a window for images' display and buttons to play and stop the sequence. Similarly, "network" is a set of instances of the Oggetto "Network" type and again, the user interface component to display this a composite one and includes a means (invoked through the button labelled "Select") to select individual values (instances of Network) for display.

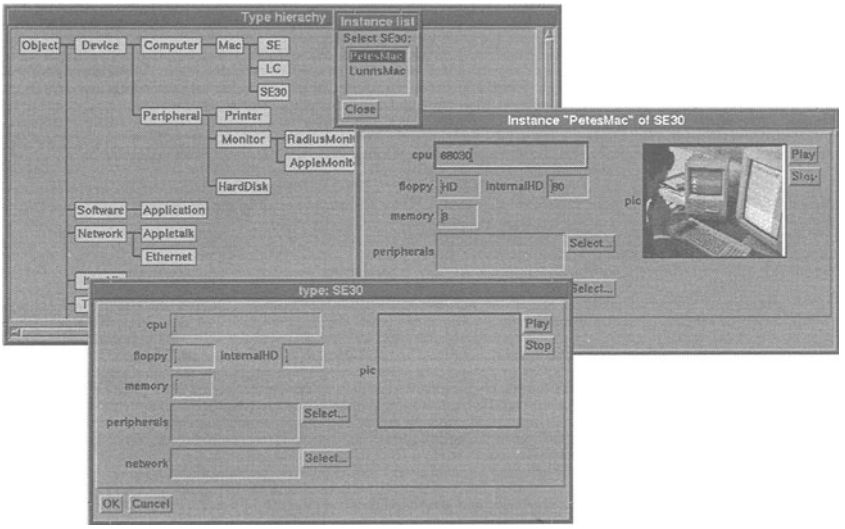

Figure 3 Moggetto in use.

4 USER INTERFACE DEFINITION

As stated above, the Moggetto framework supports the display of database objects according to the user interface definitions held by the database. We now examine how these object user interfaces may be defined.

An approach adopted by several OODBs is to provide access to a windowing system through the OODB's DDL. Each type may define its own user interface or inherit one from a super type. Inheritance is used to "bind" user interfaces to objects in exactly the same way as it is used to bind methods to object invocations (indeed, a user interface may be defined as a "display" method). Many OODBs, such as Ode (Agra90), ObjectStore (Lamb 1991) and O_2 (Deux 1991) provide bindings to general-purpose high-level programming languages, such as C or C++, permitting the direct use of windowing system libraries and toolkits. This approach has the virtue of completeness because it makes the whole range of windowing system services available, from low-level event handling through to pre-defined widget types. However, it is necessarily low-level and expensive in terms of the development effort required to exploit the windowing system.

Other OODBs use database programming languages such as GemStone's OPAL (Bretl 1989), and Kiwi's OOPS+/NAU and LOCO (Laenens 1989) which include user interface primitives as part of the language kernel. These are usually high-level abstractions over windowing system objects and functions, oriented towards the requirements of typical database user interfaces. For example, OOPS+/NAU includes primitives for defining such graphical objects as windows, text fields and buttons which may be bound to objects, views, attributes and methods.

In recent years, graphical user interface development tools have become commonplace, especially graphical user interface (GUI) builders (e.g. (Cardelli 1988)). With a GUI builder, a user interface presentation is defined by directly manipulating graphical instances (or facsimiles) of user interface components. Hence positioning, resizing, setting of components'

attributes, etc. is performed by mouse selection and dragging rather than by hand-coding. This helps support the rapid incremental development of prototypes which effective user interface development demands but which textual bindings to windowing system programming interfaces support poorly. The output from a GUI builder is skeleton code which the programmer can integrate with an underlying application via call-back/event handling routines. In the OODB domain, these may be used to map user input events onto database queries and method invocations.

Moggetto has adopted this approach by integrating GUI builder facilities with the display manager. Instead of skeleton code, however, requests to instantiate database objects (*UIobjects* - see below) which store the definition of the defined display are generated (via the object store interface). The input to an object's display definition comes from the user who wishes to define its display, but also from the database. The user's input is in the form of direct manipulation of the various interactive components of which the display is composed. The database's input is in the form of schema information; information about the object type for which the user interface is being designed. For example, if an object type has a textual attribute A, then the user is permitted to associate a user interface component type such as a text field with it so that an instance of the type will be displayed using the text field to display, and allow interaction with, the value of attribute A.

The novelty of our approach is that the graphical components assembled to form the displays embody their own editing semantics and knowledge about their relationship to database object types. The user interface components used by the display manager are MOG objects (Colebourne 1993).

4.1 MOG

MOG is an *augmented widget set.* By this we mean that MOG objects add editing behaviour to the end-user behaviour embodied by normal widgets. This allows the construction of user interfaces which can be tailored or reconfigured by the user at run-time by suspending execution, editing MOG objects' presentation attributes and resuming execution with the new user interface configuration. Even where participative design (Norman 1986) methodologies are employed, end-user modifiability is a fundamental user requirement and MOG-based user interfaces support this directly (Fischer 1990):

"*End user modifiability is not a luxury, but a necessity in cases where systems do not fit a particular task, a particular style of working or a personal sense of aesthetics.*".

The MOG objects' editing mode which permits suspended-time editing is called "user config" mode. As an example of its use, a button MOG object can react to events in two ways. It will react to an end-user's mouse button press event by inverting its graphical image to provide semantic feedback and by invoking the appropriate call-back. In addition, by switching to user config mode the user can interactively resize it, change its label, change its font, reposition it, etc. (figure 4). This is done by reinterpreting user events. Normally, a button widget recognises mouse button press events and, when one is detected, simply calls the designated call-back routine (this typically forms the interface with the underlying application). In user config mode, however, this call-back routine is ignored. Instead, one of a number of different event handling routines is invoked depending on which event is detected and where on the button they occur. These event handlers implement the editing functions and are encapsulated by the MOG object and are independent of any underlying application (figure 5). When the user has finished editing the MOG object and returns it to normal end-user mode, the application can resume execution with the new user interface configuration. In addition, the changes are recorded in the application's configuration file so that they are remembered for subsequent invocations.

Of course, at run-time, the attributes which can be edited are restricted to those which do not affect the functionality of the underlying application and which only locally affect the existing user interface components' display representations.

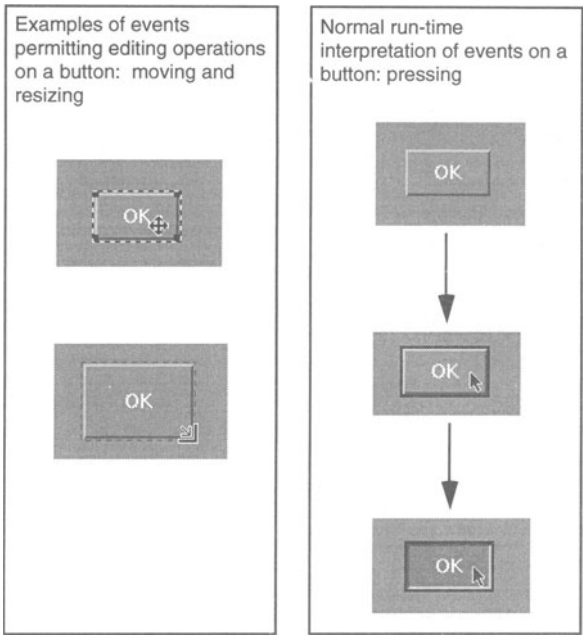

Figure 4. Examples of mouse button press events.

A MOG object is defined as a C++ object which encapsulates one or more widgets from existing widget sets such as Motif or OpenLook. The reuse of existing widget sets saves implementation effort and permits look-and-feel compatability with other applications. An encapsulated widget contains information about its resources, call-backs, recognised events, etc. and MOG augments this with operations which allow this static data to be accessed and modified. These are implemented as event handlers which forward events to the encapsulated widget when in use by an end-user but intercept and interpret as editing operations when in use by a designer (or end-user engaged in tailoring). Hence, MOG objects behave as normal widgets at run-time or as graphical editor components according to the context in which they are used. This context is determined by their mode.

In addition to performing "surface" changes whose affects are localised to the static, presentation aspects of the display, a deeper level of interactive editing can be performed on MOG objects. These include redefining a MOG object's call-back routine or creating a new MOG object and adding it to the set of MOG objects enclosed within a "container" MOG object (e.g. a subwindow such as a form). Such operations cannot be performed as suspended-time editing operations because recompilation is required and the protocol of communication between the user interface and the underlying application may be radically altered. However, the fact that MOG objects embody the necessary semantics to allow deep editing operations allows them to be easily composed to form the basic components offered to the user of a GUI builder. The mode which permits these deep editing operations is called "program" mode.

Figure 5 illustrates the basic components of a MOG object; the encapsulated widget(s), the functionality which handles the editing modes, and an event switch which redirects events to the appropriate internal event handler depending on which mode the MOG object is in.

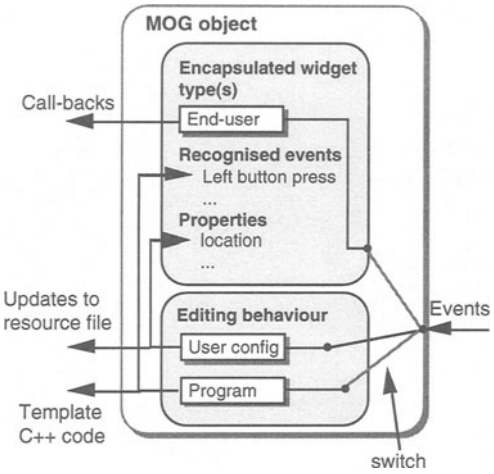

Figure 5 A MOG object.

As described above, MOG is not a GUI builder but a set of inherently tailorable user interface components. This contrasts with conventional GUI builders such as Gilt (Myers 1991), X-designer (George 1991), TeleUse (TeleSoft 1990), the NeXT interface builder and ViewCentre which are tools through which users manipulate user interface components. By embodying editing behaviour in the objects themselves any MOG-based application acquires an end-user tailoring capability without the need for a separate tool. Of course, GUI builders can be built from MOG objects (MOG can be thought of as a GUI builder-builder). A MOG-based GUI builder is simply a program where the focus is on interacting with MOG objects in edit mode(s) and which provides a means to create MOG objects of the desired types and a few other secondary functions. In effect, all other interaction is handled by the actual MOG objects of which the user interface is (being) composed. When the designer is finished, the user interface is saved by copying itself as a set of template C++ files, with the necessary MOG libraries, ready for compilation and linking with the application. Again, this is done not by the GUI builder but by the newly designed user interface itself because the top-level MOG object includes a "generate code" method.

A MOG-based GUI builder forms part of the display manager to enable Oggetto types' user interface definition. How MOG is integrated with Oggetto through the display manager is the subject of the next section.

4.2 Database object display definition with MOG

The MOG objects used by Moggetto have been adapted slightly to suit the OODB application:

- When an Oggetto object's display is first defined, instead of outputting source code which can be compiled to produce an executable user interface, it outputs requests to the database to instantiate instances of Oggetto objects which model the user interface definition (UIobjects). At run time these are read from the database and are interpreted by the display manager which remaps them onto instances of MOG objects to construct the actual object's user interface.

- When an Oggetto object's user interface is subsequently modified, instead of changes being recorded in a configuration file, the changes are recorded in the database by modifying the UIobjects.

Hence, the fact that the display is modelled in the database and interpreted at run-time by the Moggetto framework makes design and modification of Oggetto database objects' user interfaces easy to manage. In a normal MOG application, the application developer still needs to be aware of the need to write call-back routines and the details of the data structures generated by MOG. In Moggetto, this is hidden from the user because a function in the object store interface serves as the default MOG object call-back. This maps the originating user event onto the appropriate database query expression (figure 6).

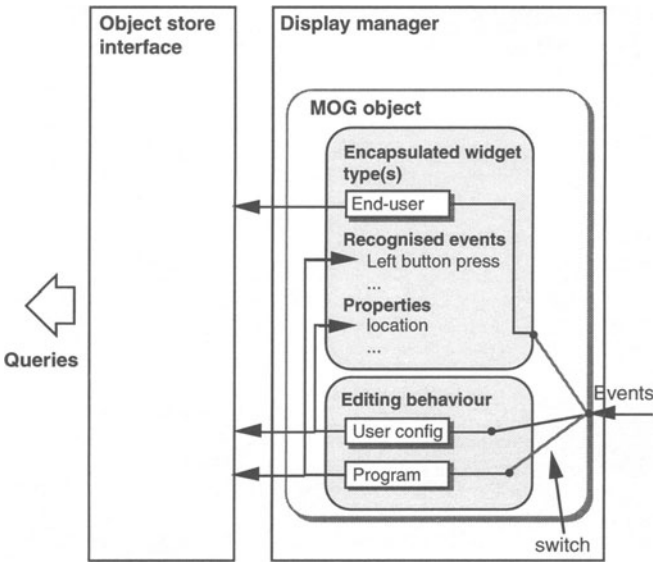

Figure 6 A MOG object integrated with the Moggetto framework.

As described above, reactive MOG objects must be associated, via UIobjects, with corresponding properties in database objects so that user events can be mapped onto the appropriate queries by the Object store interface. Failure to enforce this constraint requires that the behaviour of "unmapped" MOG objects have to be defined manually and linked with the Object store interface, breaking Moggetto's homogeneity. For example, if a database object type, who's user interface definition is encapsulated by its associated UIobject set, evolves then evolution of the user interface to track changes to the underlying object type is easily managed. This is not so where part of the user interface functionality is defined and stored outside of the Moggetto system.

5 OBJECT TYPE DEFINITION

The display manager's integral GUI builder has another function apart from the definition of displays for pre-existing object types. Its role can be reversed so that it is used as a type

definition tool. Here, a new object type is added to the database schema by manipulating MOG objects to construct a graphical template from which a new object type is generated. Typically, the user will derive a new object type as a sub-type of an existing one and will be presented with the user interface defined for this supertype. To do this, the user can simply add new MOG objects, for example fields, buttons, sub-windows, etc., and, for each, specify a corresponding property of the new Oggetto object type.

Figure 7 illustrates the definition of new object type with Moggetto; the new type "LaserPrinter" is being defined as a subtype of Printer.

Here, the attributes "manufacturer" and "network" (a multi-valued attribute) have been inherited from Printer. The user has created a new text field MOG object, labelled "DPI" and is positioning it in the window which is used to represent the user interface for instances of LaserPrinter. The next step is to specify that this MOG object represents the visualisation of a new property of the LaserPrinter Oggetto class. The user is prompted to specify its name (same as the MOG object's label by default - "DPI" in this case) whether it represents an attribute or a method; if an attribute then what type, singleton or multi-valued, etc.. Properties' visibility are also specifiable. Users are permitted to define new object types where only a subset of the properties are displayed. In such projections, the properties are still defined by directly manipulating MOG objects and specifying an associated property in the new Oggetto object type, but the user specifies that the property will remain invisible when instances of the type are displayed.

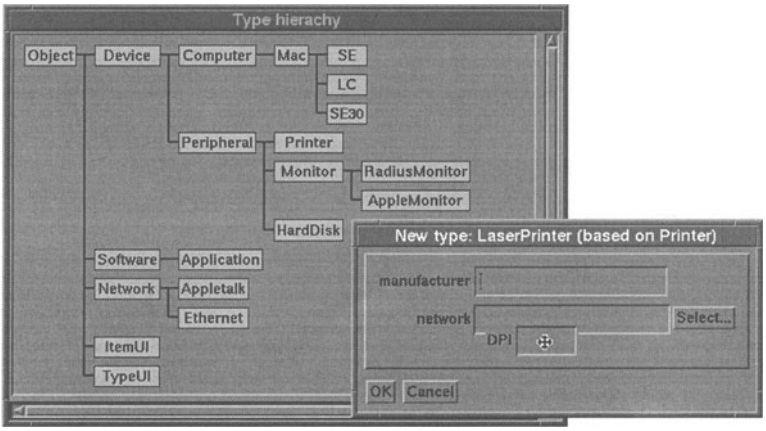

Figure 7 Definition of a new object type.

Once the new type has been defined and added to the database schema, its default display is the same as that through which the type was defined. As with any other objects' user interface, this may be subsequently redefined without affecting the underlying object type definition. This tailoring is effectively performed in MOG user config mode and where the previously described functions are performed on templates of object types' user interfaces, this shallow level of editing can be performed on actual instances of object types at run time. A change of this nature on an instance A of type X, once committed, may be made persistent and propagated to all other instances of X (if already displayed their appearance will change dynamically). Otherwise, the new user interface configuration will be local to A and forgotten when A's display is closed.

6 MODELLING THE USER INTERFACE IN THE DATABASE

The Oggetto objects' displays are modelled in the database as part of the schema. Each MOG object type has a corresponding *UIobject* type where UIobjects form a sub-hierarchy within the Moggetto type lattice explicitly defined for modelling user interface components. Every UIobject subtype corresponds to a MOG object known to the display manager and encapsulates attributes which are used to hold values of the corresponding MOG objects' visual attributes (screen co-ordinates, colour, etc.). Consider the object type Printer which has two attributes; manufacturer (a string) and network (a set of references to instances of the Oggetto type Network). These are modelled as instances of the UIobject subtypes string_field and set_of_object_field respectively (figure 8), each associated with their corresponding attribute and each corresponding to a MOG object type embodied in the display manager.

When a request is made to display an instance of Printer, the associations between Printer's attributes and its UIobjects are resolved and the UIobjects' values are collated by the object librarian which is created to manage the display of the Printer object. This display data is then passed to the display manager which dynamically creates the object display using the appropriate MOG objects which are initialised using the data from the UIobjects. For example, a text field MOG object is created to represent the attribute manufacturer with location, size, etc. as defined by the state of the string_field UIobject in figure 8.

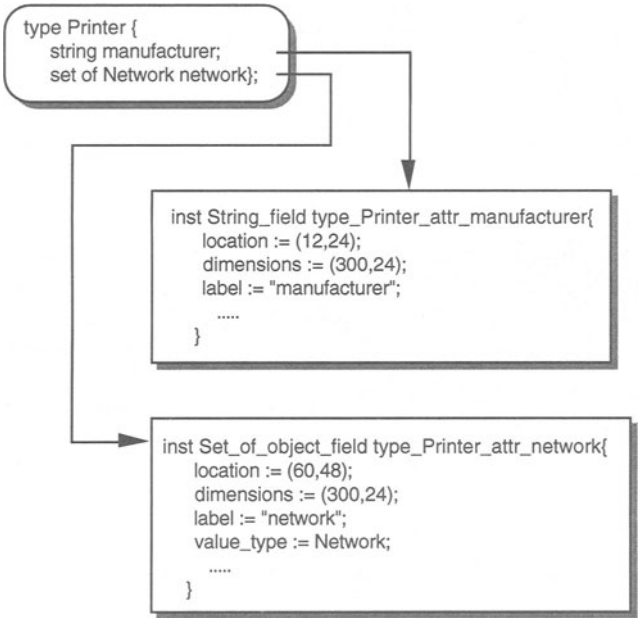

Figure 8 User interface components modelled in the database.

Subsequently, as described above, the display definition of Printer can be redefined by direct manipulation of the MOG objects of which it is composed. If, for example, the manufacturer field was enlarged and the user decided to use this new size as the default, the dimensions attribute of the string_field UIobject would be reset to the new value.

7 CONCLUSIONS

Moggetto is best viewed as a run-time framework for the direct manipulation of OODB objects on two levels; end-user interaction with the objects themselves, and object type and user interface definition by manipulation of embedded user interface components (MOG objects). It is aimed at facilitating the rapid prototyping of databases in a manner which allows the user to experiment with types and object displays without recourse to hand-coding in a data definition language. As such it has proven useful for databases such as that modelling of our department computer network.

As with King's DBface, we have adopted a variant on the Seeheim run-time architecture. While inappropriate for many user interface systems where direct manipulation of entities in a fast, reactive application is required, the architecture is very well suited to database user interface management.

MOG objects are embedded in database objects to compose their user interfaces as mappings between database object properties (attributes, methods) and instances of UIobjects. At run-time, the framework consults the UIobjects associated with a database object and dynamically constructs the object's display as a composition of the corresponding MOG objects. A user event on the display is either forwarded to the underlying database object as a query, or intercepted by the receiving MOG object and used to reconfigure the display by interpreting it as, for example, a request to relocate an attribute field.

Building database objects' user interfaces as composites of MOG objects and modelling them in the database obviates the need to store object user interfaces outside the database as code libraries. This provides all the necessary functionality to allow users to directly manipulate individual database objects. While object displays and new object types may be defined without recourse to a textual DDL, the functionality of more complex database applications would necessitate hand coding. Some problems associated with this were described at the end of section 4. These problems may be eased if the underlying OODB supports a semantically complete DDL, perhaps through bindings to other languages and programming environments, etc.. However, our intention with the development of Moggetto was to make the approach widely applicable by making minimal assumptions about these factors. Hence, we can confidently claim that the Moggetto approach is applicable to any existing OODB, requiring only that user interface components can be modelled as a separate branch of the type/class lattice.

At present, the framework incorporates simple browsing mechanisms by allowing users to interact with a graphical representation of the database schema and to traverse relationships where objects are composite (attributes of complex types). In addition we have implemented a simple object-oriented analogue of QBE and are developing ideas to graphically define more complex joins, etc. through the direct manipulation of participating object properties.

8 ACKNOWLEDGEMENTS

The authors are indebted to the UK DTI's Information Engineering Directorate and the SERC who funded the work. Additional thanks are due to the Concurrent Computer Corporation who provided useful suggestions during development and to this paper's reviewers for their comments.

9 REFERENCES

Agrawal, R., Gehani, N., Srinivisan, J. (1990) OdeView: The Graphical Interface to Ode, in *Proc. of the ACM SIGMOD International Conference on Management of Data* (eds. H. Gracia-Molina and H. Jagadish), acm Press.

Almarode, J. (1991) Issues in the Design and Implementation of a Schema Designer for an OODBMS, in *Proc. ECOOP '91*, Geneva.

Andrews, T., Harris, C. (1987) Combining Language and Database Advances in an Object-Oriented Development Environment, in *Proc OOPSLA 87.*

Atkinson, M., Morrison, R. (1988) Types, bindings and parameters in a persistent environment, in *Data Types and Persistence* (eds. M. Atkinson and R. Morrison), Springer-Verlag, Berlin.

Batini, C., Catarci, T., Costabile, M., Levialdi, S. (1992) Visual Query Systems: A Taxonomy, in*Visual Database Systems II* (eds. E. Knuth and L. Wegner), North-Holland.

Bretl, R., Maier, D., Otis, et al. (1989) The GemStone Data Management System, in *Object-Oriented Concepts, Databases, and Applications* (eds. W. Kim and F. Luchorskt), ACM Press.

Cardelli, L. (1988) Building User Interfaces by Direct Manipulation, in *Proc. ACM SIGGRAPH Symposium on User Interface Software*, Banff.

Colebourne, A., Sawyer, P., Sommerville, I. (1993) MOG user interface builder: a mechanism for integrating application and user interface, *Interacting with Computers*, **5** (3).

Collet, C., Brunel, E. (1992) Definition and Manipulation of Forms with FO2, in*Visual Database Systems II* (eds. E. Knuth and L. Wegner) North-Holland.

Deux, O. et al. (1991) The O_2 System.,*Communications of the ACM*, **34** (10).

Fischer, G., Girgensohn, A. (1990) End-User Modifiability in Design Environments, in *Proc CHI'90*, ACM Press.

George, A. (1991) X-Designer OSF/MotifGUI Builder, in *Proc European X User group Conference*, Cambridge, UK.

King, R., Noval, M. (1993) Designing Database Interfaces with DBface, *acm Transactions on Information Systems*, **11** (2).

Laenens, E., Staes, F., Vermeir, D. (1989) Browsing á la carte in Object-Oriented Databases. *The Computer Journal*, **32** (4).

Lamb, C., Landis, G., Orenstein, J., Weinreb, D. (1991) The Objectstore Database System, *Communications of the ACM*, **34** (10).

Leong, Mun-Kew, Sam, S, Narasimbalu, D. (1989) Towards a Visual Language for an Object-Oriented Database System, in,*Visual Database Systems* (ed. T.L. Kunii), North-Holland.

Mariani, J. (1992) Oggetto: An Object Oriented Database Layered on a Triple Store, *The Computer Journal*, **35** (2).

Myers, B. (1991) Separating application code from toolkits: eliminating the spaghetti of call-backs, in *Proc UIST '91.*

Norman, D. (1986) Cognitive Engineering, in*User Centred System Design* (eds. D. Norman and S Draper) Lawrence Erlbaum Associates.

Özsoyoglu, G., Wang, H. (1993) Example-Based Graphical Database Query Languages, *IEEE Computer,* **26** (5).

Pfaff, G. (1983)*User Interface Management Systems.* Springer-Verlag, Berlin.

Sawyer, P., Sommerville, I. (1988) Direct Manipulation of an Object Store, *Software Engineering Journal*, **3** (6).

Sawyer, P., Sommerville, I. (1995) MGA: rule-based specification of active object-oriented database applications, *Information and Software Technology*, (to appear).

TeleSoft (1990) *TeleUse User's Manual,* TeleSoft Inc., San Diego, Ca. USA.

Yen, M. , Scammel, R. (1993) A Human Factors Expermental Comparison of SQL and QBE, *IEEE Transactions on Software Engineering*, **91** (4).

Zloof, M.M. (1975) Query-by-Example, in: *Proc. AFIPS National Computer Conference.*

10 BIOGRAPHY

Pete Sawyer received a BSc and PhD from Lancaster University. After a period working at the Royal Signals and Radar Establishment in Malvern, he returned to Lancaster University and is currently a Lecturer in Computer Science. His research interests include HCI, Object-Oriented Database Systems and Requirements Engineering.

Andy Colebourne received a BSc from Lancaster University. Since graduating he has worked as a Research Associate in the Computing Department at Lancaster University, initially on an SERC/DTI funded research project and currently on the DTI Virtuosi project. His research interests include HCI and Virtual Reality.

John Mariani received a BSc and PhD from Strathclyde University. He is currently a Senior Lecturer in the Computing Department at Lancaster University. His research interests include Database Systems, Virtual Reality and CSCW. Dr Mariani is a chartered engineer and a member of the IEE.

Ian Sommerville received a BSc from Strathclyde University and an MSc and PhD from St. Andrews University. He is currently Professor of Software Engineering in the Computing Department at Lancaster University. His research interests include HCI, Requirements Engineering, Cooperative System Design and Safety-Critical Software. Prof. Sommerville is a chartered engineer and a member of the ACM, the British Computer Society and the IEEE Computer Society.