ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Jessie Kennedy and Peter Barclay (Eds)

Interfaces to Databases (IDS-3)

Proceedings of the 3rd International Workshop on Interfaces to
Databases, Napier University, Edinburgh, 8-10 July 1996

Paper:

Exploitation of Object-Oriented and Active Constructs in Database Interface Development

Norman W. Paton, Khoa Doan, Oscar Diaz and Arturo Jaime

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Exploitation of Object-Oriented and Active Constructs in Database Interface Development

Norman W. Paton

Department of Computer Science, University of Manchester
Oxford Road, Manchester, UK
e-mail: norm@cs.man.ac.uk

Khoa Doan

Human-Computer Interaction Laboratory, University of Maryland
College Park, MD 20742, Maryland, USA
email: doan@cs.umd.edu

Oscar Díaz, Arturo Jaime

Departamento de Lenguajes y Sistemas Informaticos
University of the Basque Country
San Sebastián, Spain
e-mail: <jipdigao,jipjaela>@si.ehu.es

Abstract

This paper presents some experiences in the exploitation of a database interface development architecture in which the interface is implemented using the facilities of the database. It is shown how novel interfaces, specifically a multi-paradigm query interface and a debugger for an active rule system, can benefit from and exploit the uniform representation of interface and database system concepts as database objects.

1 Introduction

It is clear that database interface research lags behind certain other aspects of database system development [28]. This is perhaps not surprising, as recent research in database systems has, in general, extended the facilities which a database system is expected to support. This in turn leads to increasingly sophisticated systems, with more facilities which must be accessible through the interface to different categories of user. This trend presents two distinct and complementary challenges to database interface developers: the identification of *what* facilities are required by users of advanced database systems, and the provision of tools which indicate *how* such interfaces can be constructed. This paper addresses both of these issues, although the principal focus is upon the latter, namely the provision of facilities which assist in the process of implementing database interfaces.

There is no single architecture for the implementation of database interfaces; many approaches have been proposed, some of which emphasise a clear separation between different interface components. This paper, however, focuses upon the provision of interface development tools which are fully integrated with the underlying database. This enables a single representation to be used for the modelling of application, system and interface data, which it is argued is of particular value to database interface developers. The approach is illustrated using two example interfaces which have been developed for the ADAM object-oriented database (OODB) [14]. These example interfaces are both novel in terms of the functionalities they support, and thus indicate how the requirements of advanced database facilities can be supported within an integrated interface development environment.

The paper is structured as follows: section 2 presents related work, outlining both the increasingly demanding requirements presented by database systems to their interfaces, and the range of techniques which have been used to implement database interfaces; section 3 gives an introduction to the project of which this work is part, its motivation, the basic interface development tools supported, and some early work on their application; sections 4 and 5 contain substantial examples of the exploitation of uniformity, specifically a multi-paradigm query interface and an active rule system debugger; conclusions are presented in section 6.

2 Related Work

2.1 Extending Requirements

Commercial database systems invariably support a range of interfaces and interface development tools. For example, commercial relational systems support database access and update using SQL, and allow form interfaces to be constructed interactively for well defined repetitive tasks like data entry. Such interfaces are necessary and appropriate for a range of tasks and users, but are not sufficient to support effectively the wide spectrum of database activities associated with a complex site, and are rarely targeted towards emerging database technologies. Focusing upon the latter, it is clear that many new facilities and constructs are being proposed for use with future database systems: the management of complex and composite objects, potentially with versions; support for advanced data types for use in multimedia or spatial applications; the introduction of a temporal dimension to allow monitoring of change; the development of active facilities to support automatic and timely response to events of relevance to the database. Such extensions to database systems require that corresponding revisions be made to database interfaces. A significant challenge for database researchers and developers is to increase the functionality of systems without reducing their accessibility or maintainability, a challenge which will only be met if considerable attention is directed towards database interfaces.

Various researchers have proposed database interfaces which are targeted at specific extensions to database functionality, including composite objects and user-defined relationships [2], versions [15], spatial data [19], persistent data types [7] and programs [13, 6]. It is clear from such work that there is an extensive design space, with many alternative ways of representing different concepts and operations applicable to them. This range of requirements, combined with the variety of possible solutions, suggests a need for effective tools to facilitate rapid construction of database interfaces, so that proposals can be evaluated effectively with users [23].

2.2 Database Interface Architectures

As far as interface development architectures are concerned, there is a sense in which database interfaces are nothing special. The database can be seen as an application for which an interface is required, and conventional approaches to interface development used to institute a solution. From this perspective, database interfaces can be implemented using software such as user interface management systems or widget sets, and examples of the use of such systems with databases abound [29, 3, 18].

There is also an extent to which, however, database interfaces can be held to be different from interfaces to other software systems. Such a distinction can be drawn from the observation that databases and interface management systems both include facilities for the structuring and manipulation of data. In the case of the interface management system, information must be stored which describes (for example) windows, their contents, relationships, properties and operations. As (some or all of) this information could be stored and manipulated using the facilities of the database, an obvious solution is to use the database to store information on its interface, and to use the language(s) of the database to manipulate such data. For this to be practical, the database must support comprehensive mechanisms for the structuring and manipulation of interface data, but a number of approaches have been reported in the literature [21, 17, 4, 20]. The use of an integrated architecture avoids impedance mismatches between the language used to store and manipulate interface data and that associated with the database, as discussed further in [25]. Some systems, as well as using database constructs for modelling interface data also implement interface development facilities within the database [16]. This paper presents experience with an OODB which integrates database and interface data, where the benefit of uniformity is combined with the recognised strengths of the object-oriented paradigm for interface development. The distinctive

features of this paper are the description of how database objects can be used beneath the surface of the interface, and the exploitation of active database facilities in interface development.

3 Context

3.1 Extensible Databases and their Interfaces

The work presented in this paper was initiated with a view to providing an effective interface to the extensible OODB ADAM. In ADAM, support for metaclasses as first class objects enabled its core data model to be extended with a range of facilities, including relationship objects [8], active rules [10] and composite objects with versions [22]. Where the data model can be extended with new constructs, it is necessary to be able to extend the interface to the database with visualisations relevant to the extensions, otherwise users are likely to be unaware of the semantics of the objects to which they are exposed [24]. The approach taken to the solution of this problem is presented in the following subsections, along with a summary of initial results.

3.2 Database Interface Toolkit

The use of metaclasses in ADAM to support extensibility has the important consequence that database objects are used to represent application data, data dictionary information and the system itself. This means that the same language and structuring mechanisms as were used to implement the ADAM system are used to construct applications for ADAM. This has the further consequence that an application programmer can become a system developer, extending the system with new facilities as required, using mechanisms which are familiar from the development of normal ADAM applications. By representing interface data using normal database objects, it becomes possible for graphical interfaces to be developed within this uniform context.

The representation of interface data as normal database objects has been achieved in ADAM by providing a view of an existing widget set using ADAM classes [23]. The effect of this process is to make available to ADAM programmers around 60 classes which describe dialog boxes, menus, pictures, buttons, etc. Instances of these classes are created and manipulated by message-sending, in common with other database objects, but messages can have visible consequences. For example, sending the message *open* to an instance of the class *picture* will lead to a picture window being opened on the screen.

In this paper, code fragments are not presented in ADAM notation (for which see [14, 24]), but rather using a form which requires minimal introduction. All examples presented in the paper represent functionality which is fully implemented. To exemplify the creation of a toolkit object, the class *picture* can be defined as follows:

```
class picture is_a window {
  attributes {
    name: string;
    member_graphics: set of graphics;
    ...
  }
}
```

Thus each *picture* object has a *name* and a set of *graphics* objects (lines, circles, bitmaps, etc) which have been drawn in the picture. When the message *open* is sent to an instance of *picture*, a window is opened on screen in which are drawn the objects from the *member_graphics* attribute. As such, toolkit objects are standard database objects, they can be directly related to application concepts (e.g. a *road* object can have a reference to the *line* objects which represent the road in a *picture*), and can be stored persistently alongside the appropriate application data.

3.3 Preliminary Exploitation

This subsection outlines some preliminary exploitation of the representation of interface data as database objects, emphasising the use of a common data model, the parallel evolution of interface and database constructs, and support for

additional interface functionality. Each of these aspects is illustrated through an example in the rest of this subsection. More detailed presentation of more recent results is given in sections 4 and 5.

3.3.1 A common data model: the database browser example

The first substantial program to be implemented using the toolkit mentioned above was a database browser which allowed the examination of schemas and their associated instances [24]. The browser was itself defined as an ADAM object, with reference to the various toolkit objects which were used to display information on screen. An example screen from this browser is presented in figure 1, where an instance object from the toolkit is being browsed – in effect the browser is being used to examine its own internal structure.

Figure 1: Layout of database browser showing fragment of schema and a form browsing an instance of the class *tree*.

A particular benefit which stems from the implementation of the browser within the database environment is the ease with which database information is made available to the browser. Whenever the user selects information for display, the callback behaviour of the browser, which is implemented as ADAM code, can directly access database instances or metadata. This both simplifies the implementation of the browser and speeds access to the database from the interface.

3.3.2 Parallel evolution: the tailorable visualisations example

As mentioned in section 3.1, extensible databases essentially require extensible interfaces, otherwise what is shown to the user will not reflect what is stored in the database. To enable experimentation with a range of alternative visualisations of advanced data modelling constructs, the browser described in the previous section was used as a testbed for prototyping a range of visual representations of modelling features [1]. The rapid implementation of these extensions to the browser was facilitated by the ease of access to the new modelling constructs which were also represented as database objects, and by the use of such object-oriented programming mechanisms as specialisation and overriding to tailor the default visualisations supported by the browser [23].

Revised visualisations were essentially implemented as follows:

1. Features common to the default visualisations and their extensions were identified for implementation as methods attached to the classes representing the default visualisations.
2. Subclasses of the classes representing the default visualisations were created for each extended visualisation, in which some behaviour was inherited directly from the default, some was specialised in the subclass, and some was overridden altogether.

Thus the development of enhanced visualisations exploited both the uniform representation of database and interface data, and the object-oriented nature of such data in the ADAM system.

3.3.3 *Extra functionality: the dynamic displays example*

Being objects, interfaces can benefit from all the functionality already available for standard objects such as persistence, concurrency control, integrity and security maintenance, and query languages. For instance, the query language can be used to retrieve all interface objects in which *red circles* appear. In ADAM, database objects can also be associated with active behaviour, i.e. behaviour which is not explicitly invoked but which is automatically executed as a result of a given situation being encountered. The definition of such behaviour involves the description of the *situation* to be monitored and the *action* to be taken when the situation occurs. How interface objects can profit from this feature is illustrated in what follows.

In a graphical interface which is used to display database objects, *dynamic displays* are updated automatically as modifications occur to the database objects being visualised. In this way, the consistency of a database object and its graphical representation(s) is maintained. This requires that changes to database objects be notified to all interfaces which are displaying them. For this mechanism to be efficient, such monitoring should be restricted to the objects being displayed, rather than affecting the whole database. This requires that an up-to-date record is kept of all displayed objects.

The point to notice is that the situations to be monitored involve both the interface on which objects are displayed and the objects which are being displayed. Since interfaces are objects, the same active mechanism can be used for both tasks. Thus whenever the display is updated by the user, the active mechanism can respond automatically to ensure that all displayed objects are monitored. The active behaviour which is used to monitor the displayed objects can then inform the interface of all changes to these objects, so that the display can be updated to reflect the revised state of the database [9]. In this database, rules are used to allow monitoring of the specific objects that are on screen, rather than having events monitoring updates to all objects in the database.

If interfaces were supported separately from the DBMS, interface tracking would have been implemented in an *ad-hoc* manner, increasing the difficulty for the programmer of maintaining the system – it would not have been possible to use a single mechanism to monitor change at the database and at the interface, and each monitoring mechanism would have been required to cross the database/interface boundary whenever a relevant change was detected.

4 Exploitation – Multi-Paradigm Query Interface

This section describes the implementation of a multi-paradigm query interface [12, 11] which provides three different query interfaces paradigms, namely textual (Daplex) [27], form-based and graph-based. Each query interface is a separate component, but the option is provided to translate between any of the above query paradigms. It has been perceived that different users or categories of user may prefer or be best suited to different query interface styles. A multi-paradigm interface provides system support for different categories of user by allowing users to select the interface with which they feel most at home. Therefore, a multi-paradigm interface can be used to enhance the accessibility of an object-oriented database.

The architecture of the multi-paradigm query interface is presented in figure 2. In this architecture, the interfaces, the internal form and the database are all described using database objects. The graphical interfaces share the same object-oriented structures to describe queries internally, which are constructed incrementally in response to user operations, and which can be mapped into the internal form of the optimiser, which is used to plan efficient evaluation

strategies. A mapping has also been developed which allows Daplex queries to be described using the object-oriented structures, thereby facilitating the translation of Daplex queries into any of the graphical query paradigms, and supporting the translation of the object-oriented internal form into Daplex.

Figure 2: Architecture of the multi-paradigm query interface.

The implementations of the interfaces are essentially built around the four ADAM classes depicted in figure 3, namely *query_mixin*, *daplex_interface*, *graph_interface* and *form_interface*. This class hierarchy allows the functionality common to all the paradigms to be stored in *query_mixin* and then inherited by the specific interfaces.

Figure 3: Overview of classes used in the implementation of query interfaces.

4.1 Interface Representation

The visual representations required by the three existing paradigms are very different – the Daplex interface requires an editor window into which the textual query can be typed; the form-based interface requires a class browser where all the classes of the database to be queried are displayed; the graph-based interface is based on two picture windows,

one depicting graphically the database schema and the other the query graph. All the interfaces require an operation panel which consists of a number of buttons providing facilities for constructing, editing and executing the query. For example, *Condition* and *Aggregate* buttons in the form and graph-based interfaces are used to enter boolean conditions, and to express the aggregate operations for the query. In all the interfaces, a *Run* button is used to execute the query, and the $\rightarrow$ Daplex, $\rightarrow$ Graph and $\rightarrow$ Form buttons are for translating the current query into other paradigms. Figure 4 is a screen dump of the top-level window of the graph-based interface.

Figure 4: Layout of the graph-based query interface.

Information associated with the display of a particular query interface is stored as properties of the class used to model the corresponding paradigm. For example, in the case of the graph-based interface depicted in figure 4, the class *graph_interface* must reference the base windows which contains the interface, the buttons which provide the operations, the figures which represent the schema, and the figures which represent the query. This is done by defining properties on *graph_interface* which reference the widget objects which are displayed on screen. As the widget objects are normal ADAM objects, their object identifiers can be stored directly in the attributes of *graph_interface*. For example, the following code fragment shows part of the definitions of *query_mixin* and *graph_interface*:

```
class query_mixin {
    attributes {
        internal_form: internal_query_object;
        ...
    }
    ...
}

class graph_interface is_a query_mixin {
    attributes {
```



```

    schema_window: picture;
    query_window: picture;
    operation_panel: dialog;
    run_button: button;
    ...
  }
}

```

4.2 Internal Query Representation

In all the interfaces, queries are represented internally using an object-oriented form defined as an instance of *internal_query_object* accessed through the *internal_form* attribute of *query_mixin*. The *internal_query_object* has four attributes which are the components of a query, namely, *main_query*, *subquery*, *boolean_expr* and *aggregate_expr*. The *main_query* is an object-oriented canonical form that stores both scalar and non-scalar attributes of the objects whose values are being retrieved; the *boolean_expr* is the textual expression of the boolean conditions of the query; the *aggregate_expr* is used to store information on the aggregation operations over the *main_query*; the *subquery* is itself an *internal_query_object*. The structure of the *internal_query_object* is depicted in figure 5.

Figure 5: Structure of the *internal_query_object*.

4.3 Query Translation

Since all the interfaces use a common internal form to represent a query, the translation of the queries between different interfaces simply involves the visualisation of the internal form of the query being translated. This internal representation groups together related concepts, which are in turn normally closely associated in the graphical or textual languages. For example, a node in a graph-based query interface is associated with a menu which lists the attributes of the object and the relationships in which it is involved. Exactly this information is stored in the *obj_store* and *scalar_slot_gen* slots of the *OO_Form* object in the internal representation of a query depicted in figure 5.

This structuring of queries means that queries can be translated between paradigms by a traversal of the object-oriented internal form. Such an approach can in turn be implemented by defining methods on each of the nodes within the internal form so that they know how to translate themselves for use in each of the visual interfaces, an organisational technique which has also been used in object-oriented compiler construction [5].

Figure 6: The DEAR architecture.

4.4 Database Access

One of the major problems in database application development is the *impedance mismatch* between the language used for the implementation of the application or interface and the language of the database itself. However, this problem is completely avoided in the multi-paradigm query interface, as the interface, the database and the internal form of a query are all represented as database objects.

In the context of query interfaces, database access is performed at two levels: at the instance level when evaluating a query, and at the metadata level when building a visualisation of a database schema or a representation of some part of a schema (e.g. a form). Both of these categories of access are carried out regularly, and in an interactive context where efficient processing is essential. By integrating the interface with the database, it is possible to perform query construction, answer presentation, visualisation presentation and inter-paradigm translation without the need to perform any cumbersome mappings between different languages or systems. This considerably eases the amount of effort that is required both for programmers building such applications, and for the system which is required to execute the resulting functionality.

5 Exploitation - Active Rule System Debugger

Database systems have traditionally striven to support the structural features of a domain, but considerable effort has recently been put into capturing behavioural characteristics as well. Object-oriented DBMS (OODMBS) and active DBMS illustrate this tendency. Database interfaces should evolve accordingly.

The previous section has focused on how to display structural data. However, displaying the dynamic side poses new challenges. Whereas database structure can be obtained by *querying attributes*, the dynamics of database behaviour requires obtaining relevant information on the fly as the flow of control proceeds. Here, *event trapping* rather than attribute querying is required, where an event stands for a happening of interest. In the context of an interface, this could be any occurrence that is relevant for tracing or explanation purposes.

Event trapping on external systems is a non-trivial task. Interaction with the interface to the external system commonly occurs by way of message-sending mechanisms, i.e. the observed system interacts with the normal interface to the database. By contrast, event detection is non-intrusive i.e., the observed system is unaware that it is being observed; observation, unlike interaction, does not disturb the monitored system. This requires that the graphical user interface (GUI) be able to place relevant event detectors in the observed system to keep track of the dynamics of the system. Such an architecture for an interface debugger is shown in figure 6.

The architecture closely follows the *Model-View-Controller* (MVC) framework [26]. The observed system is the model, the interface or way of presenting the information to a user, is the view, and finally, the debugger controller supports the interactive aspects of the problem. The controller interprets input events according to the current context of the application, and interacts with the model and the view component. It is worth noticing that here, input events can arise from either the user or the observed system. In most MVC implementations, the model keeps a list of its views so that the model can inform its views of any change. Instead, in this implementation, event detectors are placed on the model to inform the controller, which in turn, can undertake appropriate actions on the views.

DEAR, a Debugger for Active Rules, investigates how to display active DBMS dynamics, more concretely, active rule interactions. Active rules are being incorporated into database systems to provide automatic responses when given circumstances are reached, without user intervention. A common description of these rules is through an event which triggers the rule, a condition to be evaluated against the database, and an action to be performed if the condition is satisfied. Unlike passive databases, active systems can react autonomously to situations which require timely responses.

Figure 7: A DEAR trace.

Experience using active rules has shown that, while rules can be exploited beneficially in a range of applications (e.g. constraint maintenance, derived data support), it is not a straightforward task to implement, debug or maintain large rule bases. It is thus important for active rule systems to provide debugging tools. Of course, a paramount component of such a debugging tool is an adequate interface.

Since active rules do not have a fixed order of execution, but are fired once appropriate events occur, the dynamics correspond to the cycle of event detection and rule execution. Figure 7 shows such an event-rule cycle. The representation is a tree where the root is created artificially (the corresponding node is labeled *root*), and its direct descendents are the first events that have been raised. Such events can cause rules to be fired which, in turn, can raise further events when their actions are executed.

DEAR follows the architecture shown in figure 6. As the GUI is part of the OODB, the debugger controllers and debugger interfaces are all represented using the same object-oriented language. Moreover, communication between the *observed system* (i.e. the DBMS) and the debugger controller is achieved by using the active rule mechanism provided by the underlying DBMS. The definitions of one of these rules, a debugger controller and a debugger interface object are shown in figure 8.

A debugging session proceeds as follows:

1. The user enters a new session by creating an instance of the *debugger_controller* class. During creation, appropriate event detectors are set on the DBMS,
2. During DBMS operation, relevant events are *unintrusively* trapped and signaled to the debugger controller through active rules. Since this is a debugger for active rules, relevant events correspond to the *detection* of events, the *evaluation* of rule's conditions and the *execution* of rules actions¹.

¹ It is worth noticing that active rules used for DBMS-debugger controller communication are excluded from monitoring in order to avoid recursion.

```

oid := new instance of debugger_active_rule {
  event: tuple of <after,firing,generic_active_rule>
  condition: % if active rule instance is relevant
  action:    % notify debugger of occurrence of firing of an active rule
}

```

(a) An active rule object

```

class rule_debugger is_a debugger_controller {
  attributes {
    operation_mode: string;
    spied_data: set of object;
    interface_oid: debugger_interface;
    ...
  }
  methods {
    start(void);
    end(void);
    establishing_the_root(void);
    next_step(string, boolean);
    back_step(string, boolean);
    display_message(string);
    ...
  }
}

```

(b) A debugger controller class.

```

class debugger_interface is_a interface {
  attributes {
    debugger_oid: rule_debugger;
    browser1: browser;
    button_next: button;
    button_query: button;
    tree: tree
    ...
  }
  methods {
    new_tree(node,void);
    create_node(node,string);
  }
}

```

(c) A debugger interface class.

Figure 8: Definitions of objects participating in DEAR.

3. The debugger controller considers the event (i.e. looks at event parameters and the current debugging state) and if required, sends appropriate messages to the interface object where the dynamic aspects of the DBMS are being displayed.

Since all components are objects, unintrusive communication is easily achieved through the event mechanism available in the active DBMS.

From the interface point of view, our implementation of an active rule debugging tool has to address the following challenges:

- *Easy adjustment.* The interface should be easily customised to take into account the distinct applications supported by rules. Although some common features are shared regardless of the rule set being debugged, the application which is being supported by each rule set can influence the final design. For instance, the most effective visualisation for tracing integrity maintenance rules may be different from that for derived data support.

Using an object-oriented model, the interface can meet the easy-adjustment requirement through inheritance. An active rule debugging interface class can be reused or specialised to account for special requirements posed by rules supporting distinct applications.

- *Easy structure comparison.* Debugging can be an intricate task. Tracers just display certain parameters of the system's dynamics, but the developer still has to undertake the cumbersome task of comparing these parameters with the expected behaviour. This situation can be enhanced by what we call *pattern-based* tracing, whereby developers provide a set of patterns of expected behaviour so that the debugger can focus on deviations of the actual trace from the expected trace. Hence, developers are relieved from the duty of detecting deviations, and can focus on ascertaining the causes.

Expected behaviour patterns should be described in the same 'language' as the tracer. DEAR traces are tree-based, so patterns are defined as trees. DEAR can then detect deviations by comparing the tree being generated by the current execution with the expected trees provided by the developer. GUIs are generally weak when it comes to supporting abstract operators on complex structures, whereas OODBs commonly support operators such as *deep-equal* and *shallow-equal*. Being objects, interfaces can easily be compared using these operators that otherwise would have to be built within the GUI. We benefit here from the tradition of using OODBs to cope with complex structures found in CAD/CAM environments.

- *Easy evolution.* Active DBMS are a new and rapidly growing area where the variability of proposed systems reflects how far the field is from being settled. In this context, desirable features for a GUI include those which support easy evolution (e.g. declarativeness, modularity).

Being objects, interfaces are described through attributes and methods. Attributive description enhances declarativeness, while the encapsulation and specialisation features account for modularity and easy evolution.

- *The two-fold perspective requirement.* Behaviour debugging can be addressed from two perspectives. The behavioural perspective looks at the behaviour as it proceeds, and it is captured through the triggering tree shown in figure 7. Another perspective is to look at *the effect* of firing rules rather than the reasons that rules were fired. The effect perspective shows how the context evolves as rules are fired, where the context can be described by the database state, the rule conflict set and the events consumed.

Here, the point to note is that both perspectives correspond to the same reality. That means for the interface that both the *behavioural* and *effect-based* visualisations are components of a session interface that coordinates the monitoring process. Thus, closing the session interface leads to closing both the behaviour and effect windows, and in general changes in the session interface are propagated to its components where the different perspectives are being shown.

Such situations can be modelled in a straightforward manner if the GUI model supports the aggregation construct (commonly found in object-oriented database models). Otherwise, the aggregation semantics will be buried into the interface *call-backs* rather than being clearly separated from the interface semantics itself.

6 Conclusions

Modern database systems are evolving to support an increasing range of facilities for both structuring and manipulating information in data intensive applications. These facilities introduce requirements for additional interface functionality, but also provide an increasingly supportive environment for the implementation of database interfaces. This paper has demonstrated how object-oriented and active constructs in database systems can be exploited in the development of advanced database interfaces. These facilities have been used not only to represent the layer of interaction objects which are presented to the users directly, but also to organise the internal structure of the interfaces and to coordinate interactions between these structures. The resulting approach reduces complexity for developers, as no cumbersome mappings are required between different languages or systems, and thus eases modification and extension of database interfaces, as required by iterative development techniques.

Acknowledgements: This work was carried out when the first two authors were at Heriot-Watt University, where Khoa Doan was supported by the UK Engineering and Physical Sciences Research Council. We are also pleased to acknowledge the role of the EU Human Capital and Mobility network ACT-NET in supporting interaction in the area of active databases involving Oscar Diaz, Arturo Jaime and Norman Paton. The software presented here sits on top of an interface toolkit implemented by Ghassan al-Qaimari.

References

- [1] G. Al-Qaimari and N. W. Paton. Design and Evaluation of Visualisations for Advanced Data Modelling Constructs. In C. Christment, editor, *Basque International Workshop on IT (BIWIT)*, pages 169–182. Cepadues Press, 1994.
- [2] G. Al-Qaimari, N. W. Paton, and A.C. Kilgour. Visualising Advanced Data Modelling Constructs. *Information and Software Technology*, 36(10):597–606, 1994.
- [3] J. Almarode and T.L. Anderson. GemStone Visual Interface Designer: A Tool for Object-Oriented Database Design. In *Object-Oriented Databases: Analysis, Design and Construction (DS-4)*, pages 73–94. North-Holland, 1991. W. Meersman et al (Eds).
- [4] T. Lougeia Anderson, E. F. Ecklund, and D. Maier. PROTEUS: The DBMS User Interface as an Object. In *On Object-Oriented DB Systems*, pages 139–156. Springer-Verlag, 1991. K. R. Dittrich and U. Dayal (Eds).
- [5] M.L. Barja, N.W. Paton, and M.H. Williams. Semantics Based Implementation of a Deductive Object-Oriented Database Programming Language. *J. Programming Languages*, 2(2):93–108, 1994.
- [6] R. Conner, Q. Cutts, G. Kirby, V. Moore, and R. Morrison. Unifying Interaction with Persistent Data and Program. In P. Sawyer, editor, *Prod. 2nd Int. Wshp. on Interfaces to Database Systems*, pages 197–212. Springer-Verlag, 1994.
- [7] A. Dearle, Q. Cutts, and G. Kirby. Browsing, Grazing and Nibbling Persistent Data Structure. In *Proc. 3rd Int. Workshop on Persistent Object Systems*, pages 56–69. Springer-Verlag, 1989.
- [8] O. Diaz. The operational semantics of user-defined relationships in object-oriented database systems. *Data and Knowledge Engineering*, 16:223–240, 1995.
- [9] O. Diaz, A. Jaime, N.W. Paton, and G. al Qaimari. Supporting Dynamic Displays Using Active Rules. *SIGMOD Record*, 23(1):21–26, 1994.
- [10] O. Diaz, N. Paton, and P.M.D. Gray. Rule management in object oriented databases: a uniform approach. In G.M. Lohman, A. Sernadas, and R. Camps, editors, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, pages 317–326. Morgan Kaufmann, 1991.
- [11] D. K. Doan, N. W. Paton, and A. C. Kilgour. Design and User Testing of a Multi-paradigm Interface to an Object-Oriented Database. *ACM SIGMOD Record*, 24(3):12–17, 1995.
- [12] D. K. Doan, N. W. Paton, A. C. Kilgour, and G. Al-Qaimari. A Multi-Paradigm Query Interface To An Object-Oriented Database. *Interacting With Computers*, 7(1):25–47, 1995.
- [13] M. Gemis, J. Paredaens, and I. Thyssens. A Visual Database Management Interface Based on GOOD. In *The 1st International Workshop on Interfaces to Database Systems (IDS92)*, Glasgow, pages 25–31. Springer-Verlag, 1993. R. Cooper (Ed).
- [14] P.M.D. Gray, K.G. Kulkarni, and N.W. Paton. *Object-Oriented Databases: A Semantic Data Model Approach*. Prentice-Hall, 1992. ISBN 0-13-620203-3.
- [15] B. Gulla. A Browser for a Versioned Entity-Relational Database. In R. Cooper, editor, *The 1st International Workshop On Interfaces to Database Systems (IDS92)*, pages 136–152. Springer-Verlag, 1993.

- [16] R. King and M. Novak. Designing Database Interfaces with DBface. *ACM Trans. Information Systems*, 11:105–132, 1993.
- [17] E. Laenens, F. States, and D. Vermeir. Browsing a la carte in Object-Oriented Databases. *The Computer Journal*, 32(4):333–340, 1989.
- [18] R. Marin, M. Taboada, R.P. Barreiro, J. Mira, and A. Delgado. Rapid Prototyping of Medical Graphic Interfaces. In *Proc DEXA*, pages 161–166, 1992.
- [19] B. Meyer. Beyond Icons: Towards New Metaphors for Visual Query Languages for Spatial Information Systems. In R. Cooper, editor, *Proc. 1st International Workshop On Interfaces to Database Systems (IDS92)*, Glasgow, pages 113–135. Springer-Verlag, 1993.
- [20] K.J. Mitchell, J.B. Kennedy, and P.J. Barclay. Using a Conceptual Language to Describe a Database and its Interface. In *Proc. 13th BNCOD*, pages 101–119. Springer-Verlag, 1995.
- [21] R. Morrison, A. Dearle, A. L. Brown, and M. P. Atkinson. An Integrated Graphics Programming Environment. *Computer Graphics Forum* 5(2), pages 147–157, 1986.
- [22] N. Paton, O. Diaz, and M.L. Barja. Combining active rules and metaclasses for enhanced extensibility in object-oriented systems. *Data and Knowledge Engineering*, 10:45–63, 1993.
- [23] N. W. Paton, G. Al-Qaimari, and D.K. Doan. On Interface Objects In Object-Oriented Database. In D. Bowers, editor, *Proc. 12th British National Conference on Databases (BNCOD)*, pages 153–169. Springer-Verlag, 1994.
- [24] N. W. Paton, G. Al-Qaimari, and A. C. Kilgour. An Extensible Interface To An Extensible Object-Oriented Database System. In R. Cooper, editor, *The 1st International Workshop On Interfaces to Database Systems*, pages 265–281. Springer-Verlag, 1993.
- [25] N. W. Paton, R. Cooper, D. England, G. Al-Qaimari, and A. C. Kilgour. Integrated Architecture For Database Interface Development. *IEE Proceedings – Computers and Digital Techniques*, 141(2):73–78, 1994.
- [26] J. Rumbaugh. Modeling models and viewing views: A look at the model-view-controller framework. *Journal of Object-oriented programming*, 7(2):14–20, 1994.
- [27] D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981. Also in [30], pages 95–111.
- [28] M. Stonebraker, R. Agrawal, U. Dayal, E. Neuhold, and A. Rueter. DBMS Research At A Crossroads: The Vienna Update. In *Proc. of the 19th VLDB*, pages 688–692, Dublin, Ireland, 1993. R. Agrawal et al (Eds).
- [29] C. T. Wu. Benefits of Object-Oriented Programming in Implementing Visual Database Interface. *JOOP*, pages 8–16, March/April 1990.
- [30] S.B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, San Mateo, CA, 1990. ISSN 1046-1698, ISBN 1-55860-000-0.