Using Active Constructs in User-Interfaces to Object-Oriented Databases

Kenneth J. Mitchell, Jessie B. Kennedy, Peter J. Barclay
Computer Studies Department, Napier University,
Canal Court, 42 Craiglockhart Avenue, Edinburgh EH14 1LT, Scotland, UK
e-mail: <kenny, jessie, pete>@dcs.napier.ac.uk

Abstract

This paper examines the use of active constructs in the definition of user-interfaces to object-oriented databases. A development environment for user-interfaces to databases is presented which features the interactive use of active features of an object oriented data language through visual programming facilities. Examples are provided highlighting the salient issues of this approach, including dynamic query interfaces and complex event management.

1. Introduction

Object-oriented databases have continued to attract interest in application domains which have proved impracticable for traditional database technology. Over the same period, object-oriented user-interface technology has spread to many computer application frameworks.

This paper is concerned with findings in object-oriented databases, active database management systems, user-interface technology and human computer interaction with the aim of furthering the development of integrating technologies for user-interfaces to object-oriented databases.

Highlighting the perceived benefits of integrating programming languages and database systems when discussing object data management, Cattel [7] states:

The most important functionality advantages of object data management systems over relational database management systems are the closer association between programming language and database system... and the more complex data structures for objects...

Similarly, the active database management system manifesto [10] predicts the incorporation of the event driven paradigm into many classes of information system as a mechanism of collaboration between disparate system components:

Events, and responses to them, will therefore become a paradigm (perhaps the basic paradigm) of future information systems.

These two observations suggest user-interfaces to object-oriented databases may benefit from direct association between programming language and database system supported by the incorporation of the event driven paradigm. In previous work, the unification of object-oriented database and user-interface technologies in a well defined framework [19] has been described. Here, the use of active constructs in defining complex behaviours for user-interfaces to object-oriented databases is demonstrated and some fundamental benefits of this unification are shown.

In particular, database integrity modelling features, such as constraints and triggers, often provided by active database management systems, may be applied uniformly to development of user-interfaces to object-oriented databases. We identify ways in which such active features enable the concise specification of advanced user-interface mechanisms. One exemplary case from information visualisation is the well defined specification of dynamic queries and their associated interface controls. We also consider complex event conditions and responses, declarative updates of multiple visual views and responses to external process events.

A user-interface to databases development environment, DRIVE [20] has been implemented which dynamically interprets a conceptual object oriented data language with active constructs. This supports the interactive specification of active features through visual programming facilities. Such active features become effective immediately upon specification.

The following section gives an overview of contributing areas from which we are integrating ideas. A summary of the framework for user-interfaces to databases provides a context for this work. Section 3 highlights the interactive specification of prototype user-interfaces to databases through visual programming facilities and classifies our current system in terms of the active database management system manifesto. Section 4 introduces the main exemplar of this paper, the Transaction Explorer. This concerns an advanced user-interface facilitating the analysis of a database of residential and commercial property sales transactions in Edinburgh. Section 5 describes in detail the specification of dynamic queries using active language constructs. Section 6 highlights further important uses of active features, such as complex event conditions and responses, responses to external process events and declarative updates of multiple visual views. The paper concludes with the benefits of our approach and suggestions for further work.

2. Background

2.1. Object-oriented databases

With the increasingly complex data requirements of modern information systems many people have been drawn to the use of object-oriented databases (OODBs). This is most apparent in application domains which rely on the ability to manage compositional and classification hierarchies of data items, and data with closely associated behaviour. The sustained investment in this approach suggests there is significant benefit to be gained from such technology.

In contrast to the relational data model, data models for OODBs vary widely. Considerable effort has been made to standardise object-oriented data models [8] in the absence of a tractable mathematical definition. OODBs are generally considered to be an integration of object oriented programming language and database concepts. This is typically provided either by extending existing object oriented programming languages with database functionality or extending relational databases systems with object oriented concepts. An evolution towards seamless integration of programming language and data language has been identified [7].

2.2. Active database management systems

The modification of data in databases frequently results in situations where inconsistencies arise or some further action is needed. Dealing with such events may be required to maintain the semantic integrity of the database. In modelling database integrity, constraints may be used to specify conditions which hold throughout the lifetime of the data, and triggers may be defined to specify events and conditions upon which further actions will be taken. In addition, rules may be used to compute data values from other values, providing derived data. These features are often supported in active database management systems (ADBMSs).

ADBMSs have been a “hot topic” in recent years and in common with relational and object-oriented database management systems an identification of their common features exists in a manifesto [10]. A caveat of ADBMSs is the potential for mis-management of triggers. In complex sets of triggering rules, since an action of a trigger may in turn satisfy other triggers’ conditions, determining their termination is an undecidable problem. However, with care and development environment support their power can exploited to define database characteristics which would be impracticable if not impossible with traditional database functionality.

2.3. User-interfaces to databases

The usability of user-interfaces can be a critical factor in the success of database systems. Few would doubt the improvement for certain tasks gained by the use of techniques such as forms, query-by-example [28] and graphical schema editors over textual language interfaces. With advances in human computer interaction and the availability of more powerful graphics work-stations, information visualisation techniques [2] now enable the exploration of data of ever growing volume and dimensionality, by for example dynamic queries [24], pixel-oriented visualisation techniques [14] and advanced layout algorithms [9][11]. Sound principles for user-interfaces to databases (IDSs) are emerging supported by direct manipulation [23], sense-making [22] and Information Visualisation Artefacts (IVAs) [25].

2.4. A framework for user-interfaces to databases

In [19] we have presented a framework for IDSs and in [15] we show its relevance to information visualisation systems. This is a high-level, contextual framework facilitating discussion of human-data interaction. Based on Abowd and Beale’s model of human-computer interaction [1], it has four main components: the user, the database, the visualisation and the interaction.

To enable the specification of IDSs in terms of this framework, each component has been modelled in an object-oriented modelling language, NOODL. NOODL is a simple high-level language originally designed to restore a conceptual level to object-oriented databases [4]. The model of the framework contains the following main class categorisations: user, data, interface, and visualisation classes. It permits multiple interface objects associated with each data object and separates visualisation objects to enable the reuse of visual interface elements. Users are modelled explicitly with features for user-specific views, access privileges and visual user-embodiment.

For the purposes of this paper we are concerned only with data, interface and visualisation classes. Data classes correspond to all the classes of the database schema. Interface classes provide the control and linkage between data and their visualisations in a similar way to the presentation, abstract and control model (PAC) [21]. An interface class has properties linking a referent (data object), a metaphor (visualisation object) and compositional properties for hierarchies of interface objects. Visualisation classes model the properties and events of user-interface elements. Specifications under this model are automatically generated through language interpretation in DRIVE [20] (Database Representation Independent Virtual Environment).

2.5. User-interfaces to object-oriented databases

OODBMS vendors typically supply schema and object browsers to view the state of the database. Although it is a relatively straightforward task to display tables from relational databases, the nature of object-oriented data models requires the construction of hierarchical browsers with specific presentations predefined for each data type. Updates can usually be made to atomic property values of database objects, such as numbers and text, but more advanced kinds of modifications to the database are rarely supported. Kadyamatimba et al [13] have developed a generic user-interface for OODBs based on the desktop metaphor, which provides facilities for the population and manipulation of data instances including display and maintenance of bi-directional associations. Smalltalk's meta language constructs enable programmers to modify the database schema at run-time and therefore permit the development of tools to interactively evolve a database application's schema. This includes the potential for manipulating the behaviour of database objects.

In DRIVE [20], a meta-model of the NOODL language is used to represent and interpret the constructs of a NOODL IDS specification. This mechanism provides immediate incremental feedback during the interactive modification of schemas, instances, complex and atomic property values, and behaviour.

Development environments exist for many relational database products. Such tools may employ a visual programming environment where modifications to IDS designs can be made interactively. Indeed, environments for advanced information visualisation techniques using relational tables are now becoming a commercial viability [3]. In OODBs such development environments are mainly provided by vendors of object-oriented programming languages as application frameworks such as OWL or OSF. Although such language compilers have advanced a long way towards the facility of visual programming environments, they do not provide immediate feedback on modifications to the design of user interfaces to object oriented databases. DRIVE attempts to remedy this by supporting incremental IDS construction using an interpreted design language.

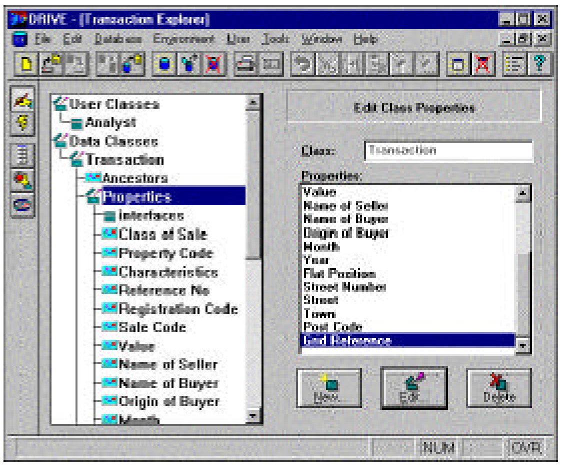


Figure 1. IDS editor within DRIVE

3. Interactive application of active user-interface to database (IDS) design features

3.1. The user-interface to database (IDS) editor

Figure 1 depicts DRIVE in the process of editing the properties of a data class. This combines a hierarchical list view of the IDS with a node specific dialog editor. When the designer selects an element in the IDS list, the view on the right changes to display the controls required to edit that element. Interestingly, the majority of nodes correspond to editors for adding, editing and deleting from ordered sets. This technique allows access to edit the entire contents of the IDS including schema and instance data.

3.2. Context sensitive NOODL definition editor

With the structural data aspects of the IDS specified, the behaviour of the application may be developed in terms of the IDS schema using the definition editor. Figure 2 shows the designer specifying a constraint on a property of a data class. The two lists above the current definition show the range of valid operators and operands from which the user can select. At this point in the predicate's definition, NOODL's *self* keyword has been selected and the lists are showing the possible items which may follow. As this is a constraint on the Transaction class, only the properties and operations of this class are valid here. The message (RHT_PRD_ERR) in the lower right of the image indicates that the current definition is incomplete. Once a constraint is validated it immediately becomes active in the IDS. This scheme is employed for all behaviour specifications in DRIVE, including operations, triggers and derived properties. An important advantage of this editor is that it frees the designer from the syntax of the language used and reduces the interaction to only a few mouse clicks. We believe this approach is preferable to traditional textual code editing; however a usability study to investigate this hypothesis is necessary.

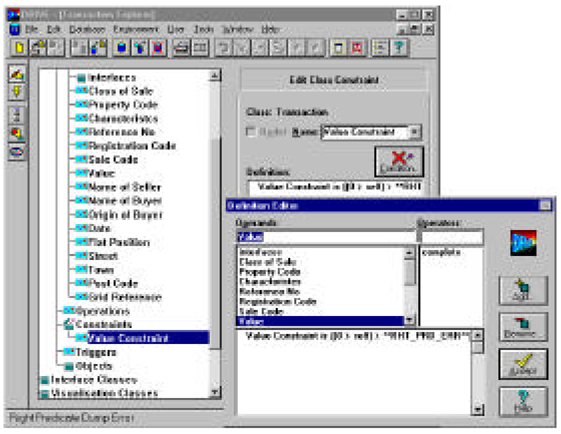


Figure 2. Context sensitive definition editor

3.3. Categorisation of DRIVE as an active database management system

There are essentially two ways to manage constraints and triggers in a database management system. The most common method is to add monitor processes to check the conditions of constraints and triggers upon particular events. This is a defining aspect of ADBMSs. The alternative approach is to check the conditions upon each data modification. This approach is rarely used outside of application code hard-wired for this purpose. Widom and Ceri [26] note several disadvantages of this approach:

1. Adding, changing, or removing a constraint or trigger requires finding and modifying the relevant code in every application
2. The correct constraint-checking or triggering behaviour is guaranteed only when every application implements it correctly
3. Additional application-to-database communication is required after every modification

However, as a consequence of DRIVE's unified IDS model and its integrated NOODL interpreter, these disadvantages can perhaps be countered:

1. The relevant code for checking constraint and trigger conditions is automatically handled by the definition editor. If a property is specified in a trigger condition, this trigger is added to the property's 'integrity pool' (and accordingly removed if necessary). Each time that property is updated, all its relevant integrity pool conditions are checked. Because the definition editor manages integrity pools dynamically, there is no redundant checking of irrelevant conditions.
2. The correct constraint-checking or triggering behaviour is guaranteed by the uniform use of integrity pools across all IDS prototypes
3. Additional application-to-database communication is potentially eliminated through the use of a unified model of application and data

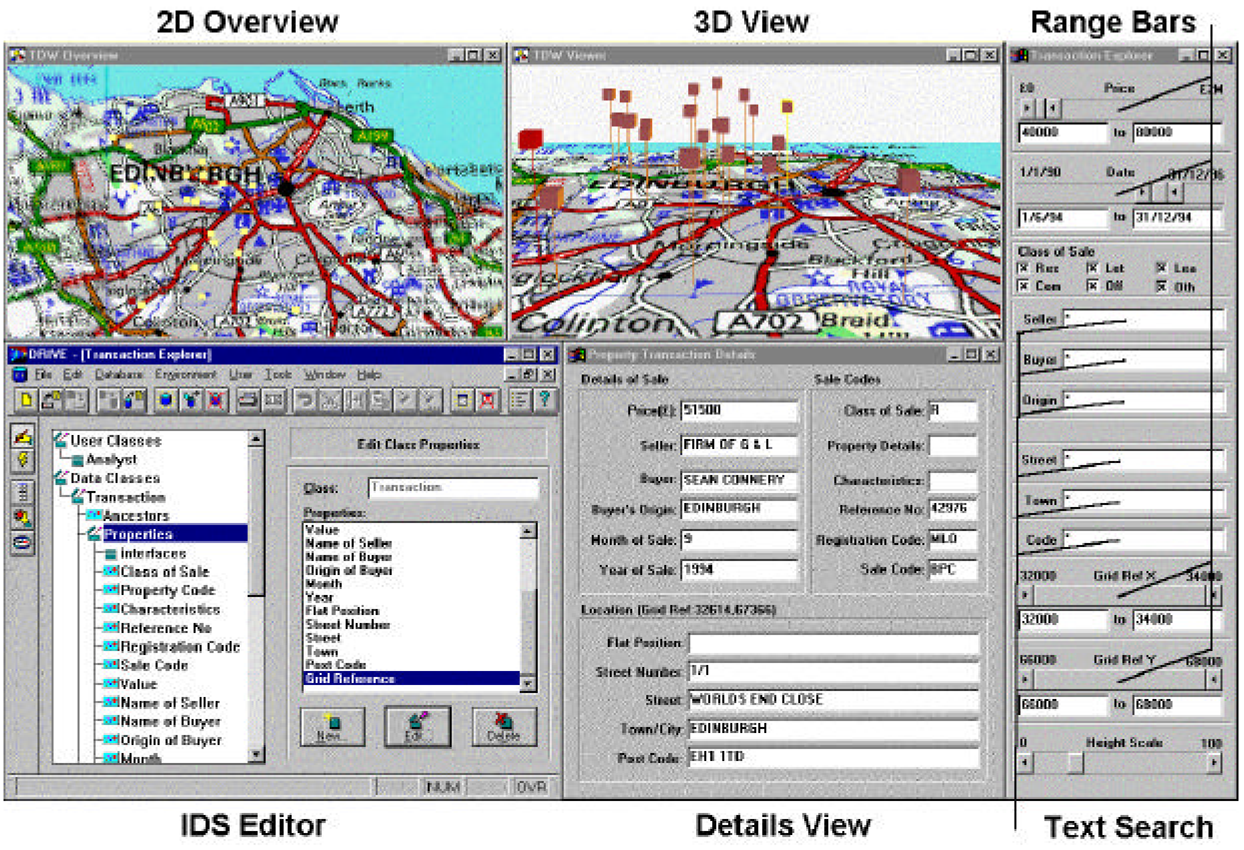
The active database management system manifesto [10] presents a summary of the characteristics found in ADBMSs. Although not an ADBMS as defined in the manifesto, DRIVE exhibits the behaviour of many features identified therein. The triggers defined in DRIVE are composed of declarative conditions and actions using NOODL constructs. Widom and Ceri [26] suggest this approach can be easier to use than the more common Event-Condition-Action rules of ADBMSs, at the expense of a loss of flexibility. However, with the ability to define trigger conditions specified on visualisation class states we believe the decrease in flexibility is limited. This is shown through our examples in section 6.

In support of rule management and evolution, DRIVE facilitates the creation, modification, and deletion of triggers at any time during the lifetime of the prototype IDS. The rules, specified as part of the IDS definition, may be browsed at any point. Additionally, a toggle may be used to activate or deactivate any individual constraint or trigger.

In terms of the manifesto, the execution model in DRIVE supports event detection, condition evaluation and execution action. It supports multi-grained binding modes for classes, collections, objects, property values and queries. However, it offers only a sequential causal dependent coupling mode (the triggered transaction begins only after the triggering transaction commits), no event consumption other than the use of Boolean conditions specified on IDS states and no persistent event history other than a debugging log file.

Finally, as described above, DRIVE supports a design environment with browser, designer, debugger, maintenance and trace facilities. However, no rule base tuner or analyser for advanced rule management currently exists.

We begin our presentation on the use of active constructs for user-interfaces to object-oriented databases by introducing the Transaction Explorer's dynamic query interface.



User Classes
Analyst
Data Classes
Transaction
Ancestors
Properties
Interfaces
Class of Sale
Property Code
Characteristics
Reference No
Registration Code
Sale Code
Value
Name of Seller
Name of Buyer
Origin of Buyer
Month

Class: Transaction

Properties:
Value
Name of Seller
Name of Buyer
Origin of Buyer
Month
Year
Flat Position
Street Number
Street
Town
Post Code
Grid Reference

Details of Sale

Price(£):	51500
Seller:	FIRM OF G & L
Buyer:	SEAN CONNERY
Buyer's Origin:	EDINBURGH
Month of Sale:	9
Year of Sale:	1994

Sale Codes

Class of Sale:	R
Property Details:	
Characteristics:	
Reference No:	42976
Registration Code:	MLO
Sale Code:	RPC

Location (Grid Ref:32614,67366)

Flat Position:	
Street Number:	1/1
Street:	WORLDS END CLOSE
Town/City:	EDINBURGH
Post Code:	EH1 1TD

£0	Price	£2M
40000	to	80000
1/1/90	Date	31/12/96
1/6/94	to	31/12/94

Class of Sale
[✓ Res] [✓ Let] [✓ Lea]
[✓ Com] [✓ Oil] [✓ Oth]

Seller	*
Buyer	*
Origin	*
Street	*
Town	*
Code	*

32000	Grid Ref X	34000
32000	to	34000
66000	Grid Ref Y	68000
66000	to	68000
0	Height Scale	100

Figure 3. The Transaction Explorer prototype constructed with DRIVE

4. Transaction Explorer

In 1992 dynamic query interfaces (DQIs) were introduced as a new method of information exploration which provides a means of refining database queries with continuous visual feedback on user interactions. The Dynamic HomeFinder [27] for exploring real-estate records showed that querying with this technique is more efficient and effective than other query interfaces.

Figure 4 shows the Transaction Explorer prototype, a tool for analysing a property sale transactions database, constructed with DRIVE. The 2D view shows a HomeFinder style overview with squares showing the geographical locations of properties satisfying the current query criteria. Each square is coloured according to the class of sale of the property transaction, e.g. yellow for residential sale, red for commercial lease. The 3D view is essentially the same as the overview, with the exception of the use of height to represent the relative cash value of each property transaction. Clicking with the mouse on any property transaction in either of these views results in the associated details being displayed in the details view. Each graphical view may zoomed independently using DRIVE's consistent navigation controls [20]. The right hand panel contains the interactive widgets for specifying dynamic queries. Range bar controls [27] define price, date and geographical ranges for property transactions appearing in the graphical views. Specific ranges may be entered in the min and max text boxes beneath each range bar. Text search boxes permit the specification of simple wild card tokens to match with the textual property values of the transaction database. Although feedback on text searches is not continuous, (it is debatable whether it should be) each view is updated upon committing a regular expression. For continuous visual feedback on textual data, alphasliders [3] may be employed. The other controls concern filtering property transactions according to the class of sale and adjusting the height scale factor of the 3D view.

Together, these features satisfy the demands of the visual information seeking mantra of 'overview, zoom and filter, and details-on-demand' [23]. The class definition below shows a NOODL specification of the original relational property transaction database imported into DRIVE.

Note the use of an enumerated SaleType and Date (using Julian dates) and Vector2D base types. Each of these are particularly useful in mapping to visual representations with DRIVE. In addition, the value constraint shown partially specified in an earlier example ensures a sensible range of price values for property transactions in the database.

5. User-interface design with active features

```
class Transaction                    { Data Class }
properties
    Class of Sale          : SaleType ;
    Property Code        : Text ;
    Characteristics      : Text ;
    Reference No           : Number ;
    Registration Code  : Text ;
    Sale Code              : Text ;
    Value                     : Number ;
    Name of Seller        : Text ;
    Name of Buyer         : Text ;
    Origin of Buyer        : Text ;
    Date                         : Date ;
    Flat Position           : Text ;
    Street Number        : Text ;
    Street                     : Text ;
    Town                          : Text ;
    Post Code              : Text ;
    Grid Ref                  : Vector2D
constraint
    Value Constraint is 0 < self.Value < 2000000
```

5.1. Dynamic query updates as declarative active responses

Given a NOODL real estate transaction as defined above, we can define queries on property values of objects of the Transaction data class (e.g. Value), using parameters specified by property values of visualisation objects (e.g. minRange and maxRange properties of the RangeBar class in Figure 4). This would determine the visibility of the graphical representation of objects in the interface (e.g. the presence or absence of a specific property transaction in the visualisation).

Figure 4 gives a NOODL specification of a range bar visualisation class. This shows the various properties of RangeBar as defined by DRIVE's custom environment library for dynamic query controls. The constraints defined on these properties further support the definition of RangeBar semantics. Next we show one method of accomplishing dynamic queries using active constructs to specify the interaction between the data classes and visualisation classes.

The following schema extract highlights the classes of the Explorer prototype which are concerned with dynamic queries based on property price. This specifies dynamic queries for property transaction values defined in NOODL using the frame-work for IDSs.

```
class RangeBar { Visualisation Class }
properties
  name              : Text ;
  position          : Vector2d ;
  size              : Vector2d ;
  minLimit          : Number ;
  maxLimit          : Number ;
  minRange          : Number ;
  maxRange          : Number ;
  minText           : Text ;
  maxText           : Text
constraints
  self.minLimit    <      self.maxLimit ;
  self.minRange  <      self.maxRange ;
  self.minRange  >=    self.minLimit ;
  self.maxRange <=    self.maxLimit
```

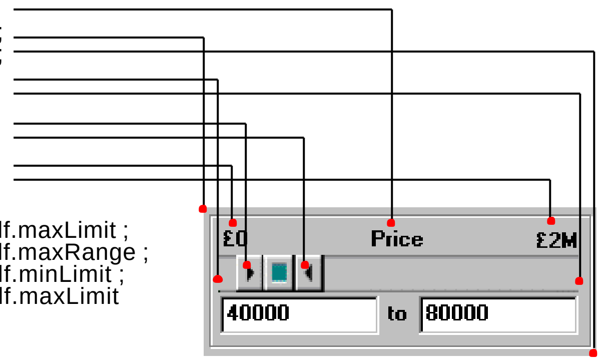


Figure 4. Example range bar visualisation class with properties identified

```
class PriceRangeBar              { Visualisation Class }
isa RangeBar
properties
    interface : PriceRangeInterface ref metaphor ;
    override minText : Text is "£" + self.minLimit ;
    override maxText : Text is
        "£" + self.maxLimit / 1000000 + "M"

class TransactionShape         { Visualisation Class }
isa Shape
properties
    interface : TransactionInterface ref metaphor ;
    override position : Vector3D is
self.position.x(self.interface.referent.GridRef.x),
self.position.y(self.interface.referent.Value/self.height) ,
self.position.z(self.interface.referent.GridRef.y) ;
    override visible : Bool is
self.interface.te.pricerange.metaphor.minRange <
self.interface.referent.Value <
self.interface.te.pricerange.metaphor.maxRange ;
    height : Real
operations
    select ;
    move
constraint
    0 < self.height < 100

class PriceRangeInterface              { Interface Class }
properties
    metaphor   : PriceRangeBar ref interface ;
    te              : TransactionExplorer ref pricerange

class TransactionInterface           { Interfaces Class }
properties
    referent : Transaction ref interface ; {link to Data Class}
    composite     : TransactionExplorer ref shapes ;
    metaphor : TransactionShape ref interface
trigger
    self.metaphor.select =>
        self.composite.detail.referent(self.referent)

class TransactionExplorer              { Interface Class }
properties
    user               : Analyst ref accessor ;
    pricerange        : PriceRangeInterface ref te ;
    shapes            : #TransactionInterface  ref  composite
```

```
;
    detail              : DetailInterface ref composite
```

The PriceRangeBar class reuses the functionality of the RangeBar class and defines constants for the min and max text labels. It also defines a link to the PriceRangeInterface class (in accordance with the principles of the IDS framework) which connects the control's visual properties with the TransactionExplorer interface class.

The TransactionExplorer class is a composite interface class which contains all the necessary components for co-ordinating real estate transaction data and their visual representations. In particular, the set of TransactionInterface objects are linked through the shapes property, and pricerange and detail are classified as component properties in terms of the IDS framework. The user property defines an accessibility for the user of the IDS. In the framework, if an accessor is defined in connection with a composite interface class, then that user has access to all components and sub-components of that class (unless otherwise constrained).

The TransactionInterface class has the Transaction data class as a referent and the TransactionShape visualisation class as a metaphor. The metaphor provides a visual representation for each transaction in the database. It inherits the features of the generic shape class for representing shapes in a 3D visualisation and is provided as part of DRIVE's 3D widget environment library (TDW) [6]. For this visualisation, the position of each shape is overridden to define its locations in terms of the properties of its associated property transaction values. This active derived property specifies that x, y and z co-ordinates of the Vector3D type are mapped to the grid reference and scaled price properties of the Transaction class.

As stated earlier, the active constructs of NOODL such as constraints and triggers are managed by the use of dynamic integrity pools. Derived property values are also handled by this mechanism.

The central point of this visualisation for dynamic queries pivots on the visibility property of the TransactionShape class. In common with the position property, this is defined as an overridden, derived value. Here, the min and max ranges of the PriceRangeBar are used in a Boolean comparison with the value of the property transaction, whose result is assigned to the visibility of the 3D shape.

The active features implemented in DRIVE ensure that this visibility condition is maintained if/when the range bar is dragged, the min and max ranges are dragged, textually specified or modified in the database and the value of the associated property transaction is altered in the database. Such updates trigger corresponding updates to the user's display.

With the flexibility of expression provided by the NOODL language, alternative methods for the specification of dynamic queries can be found. The definition detailed here merely provides an example of a possible use of active language constructs in user-interface design.

Ioannidis [12] presents an alternative perspective on dynamic queries in terms of SQL. In this work he identifies:

- multiple interacting dynamic queries. Being based on the relational model, the multiple interacting dynamic query views are specified using user-defined foreign keys between employees and department tables. Although implementations can be tailored to handle this, it is subject to the well-documented problems of using user-defined keys which do not arise through the use unique object identifiers managed by object-oriented database systems.
- general dynamic views. These are defined by arbitrary queries with parameters that are interactively modified with immediately displayed results. Such queries may involve the use of mathematical functions based on database properties.
- hypothetical updates. This is the ability to modify many data items hypothetically through dynamic query controls. Thus allowing refinement of a database modification query, which may then be committed.

All such features of DQIs are expressible using NOODL. In addition, he also identifies potentially efficient mechanisms for their implementation. Such optimisations are crucial to the success of DQIs for large datasets. Currently DRIVE does not address optimisations which are specific to dynamic queries, but their incorporation is not precluded.

Before moving on to further examples, it is noted that given constraints defined on data (e.g. the Value Constraint of the Transaction class) it may be possible automatically to define corresponding user-interface constraints based on the intrinsic properties of the data. For example, the Value Constraint may be used to derive the min and max limit values of the PriceRangeBar. The automatic resolution of such derived constraints would be difficult to police. In practice this example may be resolved by defining minValue and maxValue constants as properties of the Transaction class and defining data and visualisation class constraints in terms of these constants. In a wider sense, visual variable constraints may be concept-ualised as an extension of the existing identification of the appropriateness of particular visual variables [22] (visual representations) for its associated data.

6. Further examples

6.1. Complex triggering conditions

As stated earlier, conditions may be defined with Boolean predicates on database states (and derived database states). Staying with the Transaction Explorer DQI, multiple range bar controls can be used to define conjunctive queries on the data. For example, to combine the use of price and date range bars, the visibility condition of the TransactionShape could be specified as,

```
override visible : Bool is
   (self.interface.te.pricerange.metaphor.minRange <
      self.interface.referent.Value <
      self.interface.te.pricerange.metaphor.maxRange)
   and
   (self.interface.te.daterange.metaphor.minRange <
      self.interface.referent.Date <
      self.interface.te.daterange.metaphor.maxRange)
```

If a disjunctive query is required, e.g. to analyse two disjoint price regions simultaneously, the NOODL specification could be,

```
override visible : Bool is
   (self.interface.te.rangeA.metaphor.minRange <
      self.interface.referent.Value <
      self.interface.te.rangeA.metaphor.maxRange)
   or
   (self.interface.te.rangeB.metaphor.minRange <
      self.interface.referent.Value <
      self.interface.te.rangeB.metaphor.maxRange)
```

Negation operators could also be applied, e.g. to visualise those property transactions that do not lie in a certain price range. Clearly, the complexity of triggering conditions is limited only by the expressiveness of the language used and the computability and tractability of the implied implementation.

6.2. Declarative view updates

An important feature of DRIVE's implementation of the IDS framework is the declarative definition of multiple co-ordinated views.

All visualisation classes within DRIVE make use of an environment manager. This is a dynamic application interface which supports the run-time registration of visualisation classes, including their active properties, and event operations. Active properties are those which are directly updated or represented in the user-interface. Event operations are simple user-interface events. It is through these properties and operations that environment libraries provide a custom built user-interface interpretation of visualisation classes. This approach is based on a dynamically configurable version of the concept of *intended graphic interpretation* in G-LOTOS [5] and later in NIOME [17]. This is the mechanism used in TDW [6], dynamic query and standard widget environment libraries.

With the visualisation, interface and data classes of the IDS framework, a further declarative semantic can be used to support co-ordination of multiple views. Importantly, this describes the use of an inherent semantic of the NOODL data language and not a hard coding of specific behaviours. If the referent (data object) of an interface object is updated, then the derived properties of all of the interface object's metaphors (visualisation objects) are also modified in relation to the properties of the newly assigned data object. This active update is possible, because the referent property appears in all visualisation properties derived from properties of data objects. This means that all derived properties using the referent property are added to the referent's integrity pool, which is checked upon each modification. All properties and operations in a NOODL schema may be subject to this semantic and therefore must be maintained by an associated integrity pool.

This declarative construct is exemplified in the Transaction Explorer schema extract for the co-ordination of the graphical views with the detail view. Whenever a shape is selected, a trigger defines that the detail view's contents are updated with that shape's data object, e.g.

```
self.metaphor.select =>
   self.composite.detail.referent(self.referent)
```

Specifying a trigger condition on a visualisation class' event operation defines that the trigger action will be executed upon receiving notification of the event through the environment manager.

6.3 External process triggers

A final example is described which suggests the possibilities for using IDSs which can respond re-actively to events for external processes. In the above example the mechanism for responding to user-interface events was introduced. Here, the same technique is applied to logging the user-interface behaviour of any application running under MS Windows 95/NT™ operating systems.

Many windowing operating environments rely on the concept of message passing, e.g. as a mouse moves across the screen, messages are continuously sent to the window currently under the mouse notifying it of the current mouse position. The window can choose to respond to this as fits the desired application behaviour. For example, such a message sent to a range bar window when preceded by a mouse button down message would be interpreted as a command to drag the range bar's slider to the new position.

In DRIVE this event is handled by the range bar widget from dynamic query environment library, which deals with the widget's graphical update and then communicates through the environment manager to update the active properties of the RangeBar visualisation class. A method can be used to trap all messages sent during the operation of a windows session. In this example, these messages are stored and then passed on to their original destination using a custom windows event class, as defined below,

```
class WindowsEvent { Visualisation Class}
properties
    window       : Number ; { active property }
    message  : Number ; { active property }
    hiparam      : Number ; { active property }
    loparam      : Number ; { active property }
    time     : Time       { active property }
operation
    event                 { event operation }
```

To store sequences of these events in a database for later analysis, a simple trigger must be defined to create a data event object and append it to an event set upon notification of an event,

```
    self.metaphor.event =>
        self.referent(new (DataEvent)) ,
        self.referent.window(self.metaphor.window) ,
        self.referent.message(self.metaphor.message) ,
        self.referent.hiparam(self.metaphor.hiparam) ,
        self.referent.loparam(self.metaphor.loparam) ,
        self.referent.time(self.metaphor.time) ,
        self.composite.referent.log.add(self.referent)
```

Once stored, an event set can be visualised in the normal manner using DRIVE's existing visualisation classes or by creating custom ones. Figure 5 shows such a visualisation using a TimeRangeBar widget to manipulate a custom defined TimeLens visualisation.

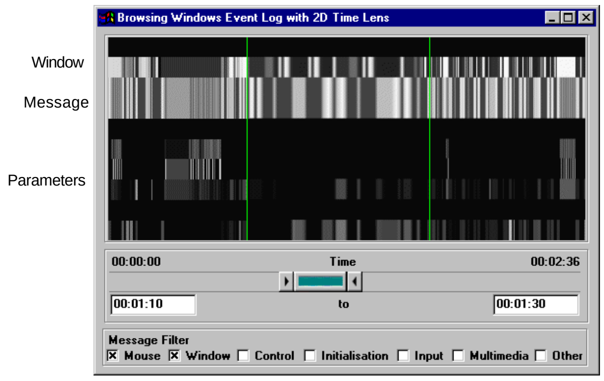

Window
Message
Parameters
Browsing Windows Event Log with 2D Time Lens
00:00:00
Time
00:02:36
00:01:10
to
00:01:30
Message Filter
[✓ Mouse] [✓ Window] Control Initialisation Input Multimedia Other

Figure 1. Windows event log with 2D time lens

This visualisation displays the numerical values as luminance bars for each event's parameters. All the recorded events are ordered in a line across the screen. A lens is used to magnify a selected area of this display, which is controlled by the TimeRangeBar.

Increasing the size of the range bar increases the number of events displayed within the magnified region, and so reduces the magnification. Decreasing this range effectively zooms in on the details of a gradually smaller number of events. Also, check boxes are used to allow the user to filter out common message types.

The 2D TimeLens visualisation of windows messaging behaviour may be used to analyse the usability of any windows interface at a very fine grain. Long bands of continuous colour indicate either hesitation by the user or delays in computer application processing. Obtaining the details of events in these regions can yield further insights into the application's behaviour.

7. Conclusion & further work

In this paper we have shown the use of active constructs in user-interfaces to object-oriented databases. Active constructs allow us clearly and concisely to define advanced user-interface mechanisms, which may be automatically interpreted to realise functional prototypes. With designs constructed in an environment supporting such constructs in a well defined framework, a foundation exists for the principled specification of user-interfaces to object-oriented databases whose components are integrated but not monolithic, reusable but not ad hoc and aptly prescribed but not unnecessarily constrained.

A potential criticism of an interpreted architecture for the realisation active constructs is that it is too slow. Acknowledging this, we recognise the need for further investigation of optimisation techniques for the rapid interpretation of IDS specifications. With respect to the interpretation of visualisation classes the most advanced graphics hardware and software may be utilised through the environment manager. A similar technique using a data manager may provide a means of incorporating accelerated interpretations of IDS definitions for specific aspects of the language, particularly with respect to data classes.

In specifying derived properties in visualisation classes based on properties of data classes there typically exists at least three references, e.g.

```
self.interface.referent.Value
```

This may be avoided by the use of behaviour classes [16], where visualisation, interface and data classes are defined as participants in a collaboration. With a collaboration, defining the above property specifier may be shortened to *Value*. Thus the visibility condition from the property transaction example could be specified simply as:

```
visible : Bool is minRange < Value < maxRange
```

Modifying the NOODL meta model to incorporate the use of behaviour classes may prove interesting.

References

[1] G. Abowd & R. Beale (1991) Users, systems and interfaces: A unifying framework for interaction, *HCI'91:People and Computers*, 73-87.

[2] *ACM SIGMOD Record Special Issue on Information Visualisation*, T. Catarci & I. Cruz (eds.). ACM-SIGMOD Record, 24(4), Dec. 1996.

[3] C. Alhberg (1996) Spotfire: an information exploration environment, *in [2].*

[4] P. Barclay & J. Kennedy (1991) Regaining the conceptual level in object oriented data modelling, *9th British National Conference on Databases*, 269-305, Butterworths.

[5] T. Bolognesi & D. Latella (1989) Techniques for the formal definition of the G-LOTOS syntax, *IEEE Workshop on Visual Languages*, 43-49.

[6] J. Boyle & K. Mitchell (1996) Embedding three dimensional graphics inside a user interface development framework, *Technical Report.* Robert-Gordon University, Aberdeen.

[7] R. Cattel (1994) Object Data Management, *Addison-*Wesley.

[8] R. Cattel (1994) The Object Database Standard, *Morgan-Kaufmann.*

[9] M. Chalmers (1993) Using a landscape metaphor to represent a corpus of documents, *European Conf. on Spatial Information Theory*, Spring-Verlag.

[10] K. Dittrich, S. Gatziu & A. Geppert (1995) The active database management system manifesto, ACT-NET Consortium, *ACM SIGMOD Record.*

[11] M. Hemmje, C. Kunkel & A. Willet (1994) Lyberworld - A visualisation user interface supporting full text retrieval, *ACM SIGIR.*

[12] Y. Ioannidis (1996) Dynamic information visualisation, *in [2].*

[13] A. Kadyamatimba, J. Mariani & P. Sawyer (1996) Desktop objects: directly manipulating data and meta data, *3rd International Workshop on Interfaces to Databases,* Springer-Verlag Electronic WIC.

[14] D. Keim (1996) Pixel-oriented database visualisation, *in [2].*

[15] J. Kennedy, K. Mitchell & P. Barclay (1996) A framework for information visualisation, *in [2].*

[16] B. Marshall, J. Kennedy & P. Barclay (1996) 'B_classes: A Construct and Method for Modelling Co-operative Object Behaviour', Technical Report, Napier University, Edinburgh.

[17] K. Mitchell (1994) *Schema visualisation.* MSc Thesis. Napier University, Edinburgh.

[18] K. Mitchell, J. Kennedy & P. Barclay (1995) Using a conceptual data language to describe a database and its interface, *13th British National Conference on Databases,* 101-119, Springer-Verlag.

[19] K. Mitchell, J. Kennedy & P. Barclay (1996) A framework for user-interfaces to databases, *in Procs of Workshop on Advanced Visual Interfaces,* ACM press.

[20] K. Mitchell & J. Kennedy (1996) DRIVE: An environment for the organised construction of user-interfaces to databases, *3rd International Workshop on Interfaces to Databases,* Springer-Verlag Electronic WIC.

[21] L. Nigay, P. Mulhem & J. Coutaz (1996) Software architecture modelling for information retrieval systems, *FADIVA'96.*

[22] P. Pirolli & S. Card (1995) Information foraging in information access environments, *ACM SIGCHI'95.*

[23] B. Shneiderman (1983) Direct manipulation: a step beyond programming languages, *IEEE Computer*, 16, 57-59.

[24] E. Tanin, R. Beigel & B. Shneiderman (1996) Incremental data structures and algorithms for dynamic query interfaces, *in [2].*

[25] L. Tweedie, R. Spence, H. Dawkes & H Su (1996) Externalising abstract mathematical models, *ACM SIGCHI'96.*

[26] J. Widom & S. Ceri (1996) *Active database systems*, Morgan-Kaufmann.

[27] C. Williamson & B. Shneiderman (1992) The dynamic HomeFinder: evaluating dynamic queries in a real estate information exploration system, *ACM SIGIR'92*, 339-346.

[28] T. Zloof (1975) Query by Example, *Proceedings of the National Computer Conference*, 431-437.