Viewing Objects

Peter J Barclay and Jessie B Kennedy

Computer Studies Dept., Napier University
219 Colinton Road, Edinburgh EH14 1DJ

Abstract. This paper examines the incorporation of database views into an object oriented conceptual model. An approach is presented where views are themselves objects, instances of view classes. These view objects provide new ways of interacting with preexisting data; no new objects are required to populate the view. Although this approach requires no new concepts to be added to the object oriented data model, a large category of views may be realised. These views allow (parameterisable) specification of their populations, and may be arranged in hierarchies; the objects they contain may be decomposed or combined, and may have properties added or hidden. The views presented maintain the integrity of the underlying object model, and allow updating where appropriate. A prototype implementation of a data management system supporting such views is described briefly.

1 Introduction

This paper presents an approach to database views consistent with a generic object oriented data model. Section 2 reviews database views, and their incorporation in object oriented approaches. Section 3 overviews the object oriented data model used. Section 4 shows how a wide category of views can be realised without need of new modelling constructs. Section 5 addresses the use and maintenance of views, describing briefly one implementation. Section 6 concludes with a discussion, a comparison with related work, and some directions for future research.

2 Background

2.1 Database Views

ANSI/SPARC have defined a three-level architecture of database description [Jar76] where the topmost 'external' level represents a collection of subschemata[1] appropriate for particular database users. The central 'conceptual' level provides a comprehensive overall description of the enterprise modelled. Since some users may work at this conceptual level directly, it might be considered a special case of a view of the data — the most complete view. The views at the external level are abstractions over this base.

Views help manage the intellectual complexity of interacting with the data, by hiding unnecessary detail and presenting information in the most appropriate format;

[1] More accurately, these are alternative schemata which may be derived from the conceptual schema or some subset of it.

further, they may provide a level of security where only the information allowable to certain users will be present in the views which they use.

A view is sometimes defined as a query, but is perhaps better thought of simply as a database schema and its extension. Since for the user of a view, the view schema provides the most comprehensive overall description of the data as she knows it, the term 'notional (conceptual) schema' will be used here to mean a schema describing the data as if the view it represents were the central conceptual level. The term 'implementing schema' will be used for the schema which shows how the notional view schema is abstracted from the base conceptual schema. This implementing schema corresponds to the queries in the 'view as query' perspective.

2.2 Views in an Object Oriented Context

Views have been well investigated in the relational context (*eg* [Dat87, chapter 8]). However, developments such as object oriented database systems [Dit88], [Oxb88], [ABD+89], [GJ89], [Kho90], [ZM90b], [ZM90a] and database programming languages [Atk78], [Bun84] (including persistent programming languages [ABC+83], [ABC+84], [Coc82], [Coo90]) require development of the concept of a view beyond that found in the relational model.

Programming languages have incorporated various notions of data abstraction [Gut77], [MP88] which have been realised in constructs such as the packages of Ada [Bar82], [alr83]. Data abstraction is central in the class concept of object oriented programming languages [SB85], [Sau89] such as Simula [BDMN79], Smalltalk [GR83] and C++ [Str87], [BG93]. Although some object oriented database systems such as Postgres have been based on extensions of the relational model [pos90], [Sto87], others such as Gemstone [BMO+89] and ONTOS [ont90] have been based closely on such object oriented programming languages. However, the programming language notions of data abstraction are often insufficient for database views since the latter requires the notions of population — a subset of preexisting objects are to participate in the view. Further, databases frequently require multiple coexisting abstractions over the same data.

So far, views have been little supported in object oriented database systems (some exceptions are reviewed in section 6.2). Whereas relational views may hide or create attributes in tables, objected oriented views must hide or create behaviour as well as structure. Further, they should not violate encapsulation, and should interact felicitously with the inheritance graph and the composition graph of the underlying model. Some approaches to object oriented views (*eg* [HZ90]) have involved the creation of new objects to populate the view; this gives rise to various problems in assigning identity [KC86] to these 'imaginary' objects.

3 NOM — The Napier Object Model

This section reviews briefly the object oriented modelling context within which views will be explored.

3.1 A Conceptual Object Oriented Model

NOM (the Napier Object Model) is an object oriented data model based on the modelling approach described by the authors in [BK91]. The basic aim of NOM is to provide a simple, 'vanilla' object data model for the investigation of issues in object oriented modelling. The model is more biased towards expressivity for semantics capture than towards efficient implementation.

NOM has been used for the analysis of novel database application areas [BK92a], and for the investigation of specific modelling issues such as declarative integrity constraints and activeness in object oriented data models [BK92b].

3.2 NOODL — the Napier Object Oriented Data Definition Language

NOODL (Napier Object Oriented Data Definition Language) is a data definition and manipulation language based on the textual notation used in [BK91]. A brief summary only is given here; further detail will be introduced later through examples. A complete description of NOODL may be found in [Bar93].

A NOODL schema consists of a collection of class definitions; each definition pertains to one particular class of object appearing in the domain modelled. Each class is named, and its ancestors (superclasses) cited. The properties of the class are named and defined, and their sorts given. NOM blurs any distinction between 'attributes' and 'methods'; the term 'property' is intended to cover both. Properties declared without definition represent stored values, those declared with definition represent computed values. The definition of a property may be an arbitrary query.

The names of properties may serve as messages to get and set the corresponding values; such messages are called *gettors* and *settors* respectively, and are distinguished simply by the absence or presence of the new value. For example, the expression `x.name` returns the name of the object `x`, and `x.name("Inyan Hoksi")` sets the value of the name property of `x` to be "Inyan Hoksi".

Operations[2] which may be suffered by instances of the class, and integrity constraints to which they are subject, are also specified in each class definition. A simple example of a NOODL schema is shown in figure 1.

4 Object Oriented Views

4.1 An Approach to Object Oriented Views

The basic technique used here to create any desired view is to create a class of objects, instances of which represent the view itself. The operations of this class provide a site for the various queries defining how the view is derived from the base. No new objects are created to populate the view; the same populations are simply viewed differently, through the new operations. This approach circumvents the problems of assigning identity to imaginary view objects, and facilitates updatability.

[2] Properties are simple characteristics of an object, representing a (notionally) stored value; operations represent the more complex behaviour of an object, are parameterisable and represented by arbitrarily long sequences of query expressions.

In the following sections, three classes of views are treated separately: selection-views, projection-views, and join-views[3]. Any general view may be a combination of these three categories, which are treated separately for clarity of exposition.

A selection-view does not change the 'shape' of the data, but hides the existence of those instances which do not meet the selection criterion ; the selection criterion thus specifies the *population* of the view. A projection-view reshapes instances of individual classes; objects may lose some properties they possess in the base, and (despite the name 'projection') may also gain new properties not specified in the base. A join-view may aggregate together objects which are separate in the base, or disaggregate single base objects into fragments in the view.

Such views can be specified (and implemented) entirely at the conceptual level, using NOODL. For simplicity of exposition, the example base schema shows only properties; operations can be treated similarly. (Operations are however used extensively in the *implementation* of the example views).

4.2 Example

In this section an example of a NOODL schema is presented which will serve as the base for the views developed subsequently (figure 1). This schema corresponds to the 'conceptual level' of the ANSI/SPARC architecture. Exact NOODL syntax is sometimes altered slightly for clarity of exposition.

This schema describes a fragment of an enterprise involving employees and other people, and the departments the former work for. The domain declaration introduces `Location` as an enumerated domain, containing only the values specified. These are taken to be all the locations with which this enterprise is concerned.

The schema shows that a person has a name, of sort Text (an arbitrary collection of alphanumeric and formatting characters), and a date of birth (dob) of sort Date[4]. An integrity constraint is that any living person must have been born after the beginning of 1880 and not later than the current date.

An employee is a sort of person, having the properties of a person above, together with its own direct properties wage, of sort Money, and dept, of sort Department. The line `\ staff` means that staff, defined in the Department class, is the obverse[5] property to dept defined in the Employee class.

Finally, a department is shown to have a name (of sort Text), a location (of the enumerated sort Location), and some staff; the sort of the staff property is a set of Employees, denoted `# Employee`.

4.3 Selection Views

In order to create a selection-view, it is necessary to hide the existence of any objects which do not meet the selection criteria. Such objects must not be found when

[3] Although these names follow relational terminology, there are some differences between these categories and the relational equivalents; these are indicated as they arise.

[4] This is essentially an abstract data type (defined elsewhere) with appropriate operators to support date arithmetic.

[5] By 'obverse' is meant the intuitive inverse of a set-valued property — see [BK91] or [Bar93].

```
domain Location is ("Edinburgh", "Paris", "Athens", "Reykjavik")

class Person
properties
   name : Text ;;
   dob  : Date ;;
constraint
   reasonable_age  is
      "1-Jan-1880" < self.dob and self.dob <= Today.date ;;

class Employee
ISA Person
properties
   wage : Money      ;;
   dept : Department
        \ staff      ;;

class Department
properties
   name     : Text         ;;
   location : Location     ;;
   staff    : # Employee
            \ dept         ;;
```

Fig. 1. Example NOODL Schema (Base)

traversing class extents, must not be returned by queries, and must not be created by database updates.

This is achieved by defining operations on the view which represent these filtered extents; these can be thought of as virtual classes, but do not introduce new sorts into the model. Then operations are defined on the view to represent the properties as in the base, but which hide the existence of any unwanted objects. The necessary steps are itemised below.

- Create a class to represent the selection-view itself.
- Define operations on the view class to return the extents of the data classes in the base, filtered by the selection criterion.
- Define operations to represent all properties returning objects not affected by the selection criterion; these are (trivially) defined as in the base.
- Define operations representing those properties returning objects affected by the selection criterion; these evaluate the properties as defined in the base, and then either return the result or a substituted fail value, depending on whether the selection criterion is met.
- Define operations to represent settors for properties not affected by the selection criterion; these are (trivially) defined as in the base.
- Define operations to represent settors for properties affected by the selection criteria; these invoke the settors defined in the base if the criteria are met,

otherwise abort.

For example, consider a selection-view which limits the population of the viewed data to only those departments located in Edinburgh or Reykjavik, and only those persons born after the beginning of 1960. A notional conceptual schema for such a view is shown in figure 2; The NOODL schema which actually implements this view in terms of the base is shown in figures 3 and 4.

```
domain Location is ("Edinburgh", "Reykjavik")

class Person
properties
   name : Text ;;
   dob  : Date ;;
constraint
   valid_age is
      "1-Jan-1960" < self.dob and self.dob <= Today.date ;;


class Employee
ISA Person
properties
   wage : Money      ;;
   dept : Department ;;
        \ staff      ;;

class Department
properties
   name     : Text         ;;
   location : Location     ;;
   staff    : # Employee
            \ dept         ;;
```

Fig. 2. Notional Schema for Selection View

Figure 3 shows the definition of the virtual classes. These are defined using the NOODL where-clause, which returns a set containing those values of the specified set meeting the specified condition. The virtual classes `SV_person`, `SV_employee` and `SV_department` will generally have smaller extents than the corresponding real classes `Person, Employee` and `Department` in the base.

Note that parameterised view extents as described in [AB91] are easily implemented in this approach. For example, if the extent of class Person had been defined in the view by:

```
   SV_person Date d : # Person is
      Person where its.dob > d       ;;
```

then all persons born after the beginning of 1960 would be represented by `SV_person("1-Jan-60")`; similarly, all persons born after, say, the beginning of 1940, would be `SV_person("1-Jan-40")`. This parameterisation produces an infinite number of virtual classes, although of course only finitely many of them will be populated. Figure 3 also shows gettors for the selection-view; where a gettor would return a value not meeting the selection criteria, it is substituted for a fail value.

```
domain Namable is Person or Department

class SV { selection-view }
operations

   { class extents }

   SV_person     : # Person is
      Person     where its.dob > "1-Jan-60"                      ;;
   SV_employee   : # Employee is
      Employee   where its.dob > "1-Jan-60"                      ;;
   SV_department : # Department is
      Department where its.location in ("Edinburgh", "Reykjavik") ;;

   { gettors }

   SV_name Namable pd : Text is pd.name  ;;
   SV_dob  Person p : Date is
      if p.dob > "1-Jan-60" then
         p.dob
      else
         bottom { fail value }           ;;
   SV_wage Employee e : Money is e.wage  ;;
   SV_dept Employee e : Department is
      if e.dept.location in ("Edinburgh", "Reykjavik") then
         e.dept
      else
         bottom { fail value }           ;;
   SV_location Department d : Text is
      if dept.location in ("Edinburgh", "Reykjavik") then
          dept.location
      else
          "error - I don't exist!"       ;;
   SV_staff Department d : # Employee is
      d.staff where its.dob > "1-Jan-60" ;;

{ bottom is NOODL universal fail value }
```

Fig. 3. Implementing Schema for Selection View (Extents and Gettors)

```
{ selection view settors }

SV_set_name Namable pd, Text n is pd.name(n)    ;;
SV_set_dob Person p, Date d is
   if d > "1-Jan-60" then
      p.dob(d)
   else
      error("selection view update violation")  ;;
SV_set_wage Employee e, Money w is e.wage(w)    ;;
SV_set_dept Employee e, Department d is
   if d.location in ("Edinburgh", "Reykjavik") then
      e.dept(d)
   else
      error("selection view update violation")  ;;
 SV_set_location Department d, Location l is
    if l in ("Edinburgh", "Reykjavik") then
       d.location(l)
    else
       error("selection view update violation") ;;
 SV_set_staff Department d, # Employee se is
    d.staff(se where its.dob > "1-Jan-60")      ;;
```

Fig. 4. Implementing Schema for Selection View (Settors)

Figure 4 shows the definition of settors appropriate to the view; it is the responsibility of the view designer to ensure that they maintain value-closure [HZ90]; that is, that they do not create objects which cannot exist in the view.

4.4 Projection Views

In a projection-view, the shapes of the data objects are altered; they may gain or lose properties.

For example, consider a view where class Person is hidden, as are the wage and dob properties of an employee. To show how a class may also gain properties not defined for it in the base[6] the class Employee gains a property age derived from the hidden base property dob. The location property of class Department is hidden.

Figure 5 shows the notional conceptual schema which would describe this projection-view. The schema actually implementing this view over the base (as defined in figure 1) is given in figure 6. The steps to create a projection-view are itemised below.

- Create a class to represent the projection-view itself.
- Define operations to return the (full) extents of those data classes in the base which appear in the view.
- Define operations to represent gettors for all properties in the base appearing in the view. No gettors are defined for those properties which are to be hidden.

[6] This could be termed an 'accretion-view'

```
class Employee
properties
   name : Text    ;;
   age  : Number  ;;
   dept : Department
        \ staff   ;;

class Department
properties
   name     : Text  ;;
   staff    : # Employee
            \ dept  ;;
```

Fig. 5. Notional Schema for Projection View

(For consistency, any properties which return objects belonging to classes which are hidden in the view should themselves be hidden).
- Define operations representing those gettors for properties defined in the view but not present in the base.
- Define operations to represent settors for properties present in both the base and the view; these are (trivially) defined as in the base.
- Define operations to represent settors for properties present in the view but not the base (where possible).

Where classes or properties are hidden in a view, it is the responsibility of the view designer to ensure that type-closure is maintained [HZ90]; that is, all the sorts mentioned in operation signatures appearing in the view schema must be provided in the view.

4.5 Join Views

A 'join-view' is a view in which separate base objects may be aggregated into single view objects, or single base objects may be disaggregated into separate view objects. Despite the name, relational-style join operations will often be unnecessary since links between objects will be encoded in the schema; that is, finding them requires navigations rather than searches (see [Bar93, chapter3]). The steps necessary to construct a join-view are itemised below:

- Create a class to represent the join-view itself.
- Define operations to return the extents of those data classes which appear in the view. Where objects of several base classes are aggregated, a virtual class derived from the extent of one of these will serve to represent the aggregation. Where base objects are disaggregated, several virtual classes derived from the same base class will be used.
- Define operations to represent gettors for properties of data classes appearing in the view without (dis)aggregation; these are (trivially) defined as in the base.

```
class PV { projection-view }
operations

   { class extents }

   PV_employee   is Employee   ;;
   PV_department is Department ;;

   { gettors }

   PV_name Namable pd    : Text is pd.name                          ;;
   PV_age Employee e     : Number is (Today.date - e.dob) div 365 ;;
   PV_dept Employee e    : Department is e.dept                     ;;
   PV_staff Department d : # Employee is d.staff                    ;;

   { settors }

   PV_set_name Namable pd, Text n pd.name(n)                        ;;
   PV_set_dept Employee e, Department d is e.dept(d)                ;;
   PV Department d, # Employee se is d.staff(se)                    ;;
```

Fig. 6. Implementing Schema for Projection View

- Define operations to represent gettors for properties of objects suffering (dis)aggregation in the view; these gettors will incorporate queries containing the necessary navigational or search expression.
- Define operations to represent settors for properties of data classes appearing in the view without (dis)aggregation; these are (trivially) defined as in the base.
- Define operations to represent settors for properties of objects suffering (dis)aggregation in the view; this will involve inverting the query expressions to update the correct object in the base.

As an example, imagine that some users have a view in which person and department objects are not present; instead, each employee has as a direct property the name, size and location of the department for which she works. The notional schema describing this view is shown in figure 7. The schema actually implementing this view is shown in figure 8.

The properties dept_name, dept_size and location of an employee are implemented by delegation to the appropriate associated instance; this is similar to the technique Neuhold and Schrefl calls 'message-forwarding' [NS88]. Note that it is possible to update the dept_name and location properties, but not the dept_size property.

Since in an object oriented model information will often be contained in navigation paths which in the relational model would require a join, it may be expected that a wider class of views will be updatable. However, updatability still relies on being able to invert the derivation function, so for example 'statistical summary' views will not in general be updatable.

```
class Employee
properties
   name      : Text   ;;
   dob       : Date   ;;
   wage      : Money  ;;
   dept_name : Text   ;;
   dept_size : Number ;;
   location  : Text   ;;
```

Fig. 7. Notional Schema for Join View

```
class JV { join-view }
operations

   { class extents }

   PV_employee is Employee                          ;;

   { gettors }

   PV_name      Employee e : Text   is e.name                   ;;
   PV_dob       Employee e : Date   is e.dob                    ;;
   PV_wage      Employee e : money  is e.wage                   ;;
   PV_dept_name Employee e : Text   is e.dept.name              ;;
   PV_dept_size Employee e : Number is e.dept.staff.cardinality ;;
   PV_location  Employee e : Text   is e.dept.location          ;;

   { settors }

   PV_set_name      Employee e, Text n   is e.name(n)                ;;
   PV_set_dob       Employee e, Date d   is e.dob(d)                 ;;
   PV_set_wage      Employee e, Money m  is e.wage(m)                ;;
   PV_set_dept_name Employee e, Text n   is e.dept.name(n)           ;;
   { department size not updatable since it is a "statistical" function }
   PV_set_location  Employee e, Text l   is e.dept.location(l)       ;;
```

Fig. 8. Implementing Schema for Join View

The semantics represented are that changing the department name of an employee represents a change in name of the department; all others employees of the department will 'see' the change, since the one object to which they all delegate the get-name message is the only object actually modified. If updating the department name of an employee were to mean transferring the employee to a different department, then the gettor could still be defined appropriately. The view designer must establish the intended semantics of such an update.

In general, it is the responsibility of the view designer to ensure that update operations provide equivalence-preservation [HZ90]; that is, that they provide the correct changes in the base to provide the intended update in the view.

If objects in the base are to be disaggregated in the view (*ie* really it is an 'unjoin-view'), this may be accomplished in a similar manner. If a class Lorry has properties representing information both about the motor and the trailer of the lorry, two new classes Motor and Trailer may be defined if these are to be disaggregated in the view. In fact, both classes have the same extent, the same as the extent of Lorry, but properties relating to the motor or the trailer specifically are defined only on the appropriate virtual class in the view. In this way one class in the base is split into two virtual classes in the view.

5 Using and Maintaining Views

5.1 Interacting with Data Through a View

Another way of thinking of a view is as an interface between some data and some programs; even the interactive manipulation of viewed data by a user requires some program (a browser or query engine) to access the data, and so may be considered in the same way.

To use a view, an instance of the view class must be created; let it be called `myview`. Then the extent of class Person under this view is represented by the NOODL expression `myview.MV_person()`, and the name of object `x` under the view by `myview.MV_name(x)`. Instead of sending the message to the data object as in the base (`x.name`), the message is sent to the view itself, with the data object as parameter. The tag `MV_` shows in which view these messages are defined. Of course, if desired, any program accessing these view operations may rename them, eliminating the view identifier tag so that in the local name space of the program these names correspond to those in the base; for clarity the tagged forms will be used here. (Similarly, the operations could be locally redefined to be applied to the data objects themselves rather than to the view).

Calling the place where a program looks for the persistent data to which it binds a *binding space*, then to use a view a single instance of the view-class is created and placed in this binding space. The operations on persistent objects under this view are then available as the operations, parameterised by these objects, on the view object itself.

5.2 View Evolution

Views should provide logical data independence; this enforces a separation of an individual's view of the data from the community view. The alteration of one view of the data should not impact on any other view of the data (unless of course the latter is a higher level view based on the changed view). Since views are represented as objects defined within the enterprise schema, alteration of a view may be considered as a schema evolution [BCG+87]. In general, where preexistent instances of persistent classes do not require to be modified, evolution may be supported relatively easily;

where instances do require modification, techniques such as conversion, screening [BMO+89], lazy evolution [Owo84] or partial evolution [Bar93] are required. However, since views as presented here do not require the creation of any new database objects other than the ones representing the views themselves, in any implementation the modifications required by a view evolution should be only of the former, more straightforward kind.

5.3 GNOME — a Generic Napier Object Model Environment

GNOME (Generic Napier Object Model Environment) is a software system to support the construction of data intensive applications in the persistent programming language Napier88 [DCBM89], [MBCD89]. GNOME is based on the software construction approach described in [Bar93]. GNOME contains a schema compiler which allows the automatic generation of the Napier88 structures necessary to represent data described in NOODL, providing an active interface to the data [Day88] and enforcing integrity constraints. Although Napier88 is not an object oriented programming language, GNOME supports the development of applications in Napier88 based on an object oriented data design. Data modelling [Mul92], [Gol92] and *ad hoc* querying tools are under construction, and facilities of management of data evolution are planned.

GNOME created the necessary infrastructure to represent an enterprise described in NOODL. Collections representing class extents and procedures, the application[7] of which represent sending messages to objects, are made available to applications which interact with the data. The actual physical data structures are hidden behind this interface, providing first order information hiding [CW85]. These access procedures are placed in a Napier88 environment [Dea88], which constitutes the binding space between the persistent data and the applications. The automatically created binding space contains a complete realisation of the conceptual level description of the data.

This presentation of object oriented views has focussed on their incorporation into a conceptual model. However, since no new constructs are required for the realisation of the object oriented views described in section 4, GNOME immediately provides an implementation for these views without need for modification; similarly, the use of such views adds flexibility and usability to GNOME. The semantic checking which GNOME provides for NOODL schemata ensures that these views are well-formed. It is planned to experiment with some substantial case studies of view-implementation.

6 Conclusion

6.1 Discussion

An approach has been described where views themselves are represented as objects. There is not a general class 'View'; rather, a class is defined for each view required, showing how the view is derived from the base schema. An instance of this view

[7] Despatching over class to execute the correct local code is handled by the GNOME infrastructure.

class then is an object which can mediate access to the database, presenting the underlying data as required by the view.

Views can be used in the construction of other, higher level views. One interesting possibility arising from the representation of views as objects is the construction of view hierarchies. This would be particularly useful where a number of views are in use, corresponding to varying levels of detail, or varying levels of security.

The main disadvantage of this approach is the need to introduce a class for each view which has but a single instance, which may seem contrived from a conceptual viewpoint. On the other hand, such one-instance classes can provide other useful functions such as providing structure for enterprise models [Bar93, chapter 4].

6.2 Related Work

Connor *et al* have presented a technique for using the existential types [MP88] of Napier88 to construct strongly typed multiple coexisting abstractions over the same persistent data [CDMB90]. This provides a mechanism similar to database views, although it focuses only on what are here termed projection-views. Since there is no notion of inheritance in Napier88, these views are not object oriented. Connor *et al*'s work develops a particular way of using Napier88, in contrast with the work reported here which uses Napier88 as the implementation vehicle for a particular semantic model.

Neuhold and Schrefl have described a system based on message forwarding [NS88] where a knowledge base management system attempts to deduce and construct a personalised view of data based on a user's attempts to query it. This work focuses on techniques to realise such a personalised view, rather than on basic issues of view definition.

Shilling and Sweeney have described extensions to the conventional object oriented paradigm which support the construction of views in a software development environment [SS89]. Multiple copies of instance variables are available to support versioning, and a system implemented in C++ resolves references to these by methods. Unlike in the work reported here, a 'view instance' is taken to signify a particular activation of a view with a corresponding set of values of the instance variables.

Views in an object oriented database are outlined by Mariani in [Mar92]. These views have a relational flavour; a mechanism is shown where views can be represented by classes defined using selective inheritance.

Mamou and Medeiros describe 'hyperviews' [MM91]. A software system constructs a view, given the schema which defines it and a query which establishes its population; a graphical interface to the view can then be constructed. Mamou and Medeiros's focuses on the presentation and manipulation of data through views, rather than on view definition.

Abiteboul and Bonner present an approach to views which centres on specifying the populations of the view [AB91]; this work focuses heavily on type-inference, relieving the user of the need to specify the position of the view classes within the class hierarchy (view classes are virtual classes integrated into the conceptual level description, rather than a self-contained alternative description). Since the approach used involves creating new objects, fixing the identity of these is problematical.

Heiler and Zdonik present the only work aimed at realising views without the need for new constructs in the data model. Data abstraction through views is examined [HZ88], the important criteria of value-closure, type-closure and equivalence-preservation are introduced, and the use of views to support the federation of heterogeneous databases is discussed [HZ90].

Richardson and Schwartz introduce 'aspects', which provide a convincing solution to allowing objects to have multiple, independent rôles within a strongly typed model [RS91]. Aspects provide new interfaces to existing objects, and hence could be adapted as a view mechanism. However, providing a new view of a schema would involve creating a new aspect instance for each object populating that schema.

None of the above have emphasised investigation of the extent to which the potential to support views is inherent in a representative 'vanilla' object model. The work reported in [AB91] and [HZ90] is the most similar to that reported in this paper. However, the approach of both of these papers allows the creation of new objects to populate views, raising problems of fixing identity. Neither has represented views themselves as objects, which not only integrates views well into the basic object model but also allows the construction of hierarchies of views (see section 6.1). None of the other work has identified and treated the three issues of populating a view, restructuring objects in a view, and aggregating or disaggregating objects in a view respectively through the (extended) concepts of a selection-view, projection-view and join-view.

6.3 Further Work

Some case studies of view-implementation in real-world applications remain to be investigated.

By careful construction of settors it has often been possible to create updatable views; a detailed investigation of updatability in object oriented views is merited (including object creation within views, which for brevity is not discussed in this paper). Such an investigation might identify useful standard approaches to updatability, guaranteeing equivalence-preservation, and relieving the designer of some of the effort of crafting the settors.

An interesting and unexplored area is the automatic derivation of views, in the sense of generating the implementing schema from the notional view schema and the base schema. It is hoped that a GNOME tool will be developed from exploration of this idea.

7 Summary

A proposal for the representation of views in an object context has been presented; the approach followed should be possible in any objected oriented model or system which allows the definition of parameterisable operations on user-defined classes. Since no constructs are added to the model used specifically for the support of views, this work demonstrates that the potentiality for views of useful sophistication is inherent in many object oriented models and systems.

Examples have been given of how view populations may be specified, how objects gain and lose properties in the view as compared to in the base, of how base objects

may be aggregated or disaggregated in the view, and of how appropriate update operations may often be defined. Multiple views may be defined over the same base, and views may be arranged into hierarchies.

No new objects are created in view populations, obviating problems of assigning identity. Views of objects interact safely with the generalisation and aggregation structures of the model, and with explicit integrity constraints. A system supporting such views is briefly described.

References

[AB91] S Abiteboul and A Bonner. Objects and Views. In *proc ACM SIGMOD conference (SIGMOD Record)*, pages 238 – 247, June 1991.

[ABC+83] MP Atkinson, PJ Bailey, KJ Chisholm, WP Cockshott, and R Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4), 1983.

[ABC+84] MP Atkinson, P Bailey, WP Cockshott, et al. Progress with Persistent Programming. In Stocker, editor, *Databases - Role and Structure*. Cambridge University Press, 1984.

[ABD+89] M Atkinson, F Bancilhon, D DeWitt, K Dittrich, D Maier, and S Zdonik. The Object Oriented Database System Manifesto: (a Political Pamphlet). In *proc DOOD*, Kyoto, Dec 1989.

[alr83] *A Reference Manual for the Ada Programming Language*. US Government (ANSI/MIL-STD 1815 A), 1983.

[Atk78] Malcolm P Atkinson. Programming Languages and Databases. In *proc VLDB 4*, pages 408 – 419, Berlin, Sep 1978.

[Bar82] John GP Barnes. *Programming in Ada*. Addison Wesley, 1982.

[Bar93] Peter J Barclay. *Object Oriented Modelling of Complex Data with Automatic Generation of a Persistent Representation*. PhD thesis, Napier University, Edinburgh, 1993.

[BCG+87] J Banerjee, H-T Chou, JF Garza, W Kim, D Woelk, and N Ballou. Data Model Issues in Object Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3 – 26, Jan 1987.

[BDMN79] GM Birtwistle, O-J Dahl, B Myhrhaug, and K Nygaard. *Simula Begin*. Van Nostrand Reinhold, New York, 1979.

[BG93] Kenneth Barclay and Brian Gordon. *Developing Object Oriented Software in C++*. Prentice Hall, 1993.

[BK91] Peter J Barclay and Jessie B Kennedy. Regaining the Conceptual Level in Object Oriented Data Modelling. In *proc BNCOD-9*, Wolverhampton, Jun 1991. Butterworths.

[BK92a] Peter J Barclay and Jessie B Kennedy. Modelling Ecological Data. In *proc 6th International Working Conference on Scientific and Statistical Database Management*, Ascona, Switzerland, Jun 1992. Eidgenossische Technische Hochschule, Zurich.

[BK92b] Peter J Barclay and Jessie B Kennedy. Semantic Integrity for Persistent Objects. *Information and Software Technology*, 34(8):533 – 541, August 1992.

[BMO+89] R Bretl, D Maier, A Otis, J Penney, B Schuchardt, and J Stein. The Gemstone Data Management System. In W Kim and FH Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, 1989.

[Bun84] P Buneman. Can We Reconcile Programming Languages and Databases? In Stocker, editor, *Databases - Role and Structure*. Cambridge University Press, 1984.

[CDMB90] Richard Connor, Alan Dearle, Ron Morrison, and Fred Brown. Existentially Quantified Types as a Database Viewing Mechanism. Technical report, University of St Andrews, 1990.

[Coc82] W Paul Cockshott. *Orthogonal Persistence*. PhD thesis, University of Edinburgh, 1982.

[Coo90] Richard Cooper. *On The Utilisation of Persistent Programming Environments*. PhD thesis, University of Glasgow, 1990.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4), Dec 1985.

[Dat87] CJ Date. *An Introduction to Database Systems*. Addison-Wesley, 1987.

[Day88] Umeshwar Dayal. Active Database Management Systems. In *proc 3rd International Conference on Data and Knowledge Bases*, pages 150 – 169, Jerusalem, Jun 1988.

[DCBM89] Alan Dearle, Richard Connor, Fred Brown, and Ron Morrison. Napier88 - A Database Programming Language? In *proc DBPL 2*, 1989.

[Dea88] Alan Dearle. Environments: A Flexible Binding Mechanism to Support System Evolution. *22nd International Conference on Systems Sciences*, 1988.

[Dit88] KR Dittrich. Advances in Object Oriented Database Systems. *Lecture Notes in Computer Science*, 334, 1988.

[GJ89] MA Garvey and Michael S Jackson. Introduction to Object Oriented Databases. *Information and Software Technology*, 31(10), Dec 1989.

[Gol92] Craig Goldie. An Object Oriented Schema Compiler. Technical report, Napier University, Edinburgh, 1992.

[GR83] A Goldberg and D Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, May 1983.

[Gut77] John Guttag. Abstract Data Types and the Development of Data Structures. *CACM*, 20(6), Jun 1977.

[HZ88] Sandra Heiler and Stanley Zdonik. Views, Data Abstraction and Inheritance in the FUGUE Data Model. In *proc 2nd Workshop on Object Oriented Database Systems*, pages 225 – 241. Springer Verlag, 1988.

[HZ90] S Heiler and S Zdonik. Object Views: Extending the Vision. In *proc 6th International Conference on Data Engineering*, pages 86 – 93. IEEE Computer Society Press, 1990.

[Jar76] DA Jardine. *The ANSI/SPARC DBMS Model*. North-Holland Pub. Co., 1976.

[KC86] S Khosafian and GC Copeland. Object Identity. In Norman Meyrowitz, editor, *proc OOPSLA*, pages 406 – 416, Portland, Oregon, September 1986.

[Kho90] S Khoshafian. Insight into Object Oriented Databases. *Information and Software Technology*, 32(4):274 – 289, 1990.

[Mar92] John A Mariani. Realising Relational-Style Operators and Views in the Oggetto Object Oriented Database. Technical report, Lancaster University, Lancaster, 1992.

[MBCD89] R Morrison, F Brown, R Connor, and A Dearle. The Napier88 Reference Manual. Technical report, Universities of Glasgow and St Andrews, Jul 1989.

[MM91] J-C Mamou and CB Medeiros. Interactive Manipulation of Object Oriented Views. In *proc 7th International Conference on Data Engineering*, pages 60 – 69. IEEE Computer Society Press, 1991.

[MP88] John C Mitchell and Gordon D Plotkin. Abstract Types Have Existential Type. *ACM TOPLAS*, 10(3):470 – 502, Jul 1988.

[Mul92] Anthony Mullen. An Object Oriented Modelling Tool. Technical report, Napier University, Edinburgh, 1992.

[NS88] EJ Neuhold and M Schrefl. Dynamic Derivation of Personalised Views. In *proc 14th International Conference on Very Large Data Bases*, Long Beach, California, 1988.

[ont90] ONTOS SQL User's Guide. *(ONTOS documentation)*, 12 Dec 1990.

[Owo84] GO Owoso. *Data Description and Manipulation in Persistent Programming Languages*. PhD thesis, University of Edinburgh, 1984.

[Oxb88] EA Oxborrow. Object Oriented Database Systems: What are they and what is their Future? *Database Technology*, Jun 1988.

[pos90] *Postgres Reference Manual (version 2.0)*. University of California, 1990.

[RS91] Joel Richardson and Peter Schwartz. Aspects: Extending Objects to Support Multiple, Independent Roles. In *proc annual SIGMOD conference*, pages 298 – 307. ACM Press, 1991.

[Sau89] John H Saunders. A Survey of Object Oriented Programming Languages. *Journal of Object Oriented Programming*, Mar/Apr 1989.

[SB85] M Stefik and DG Bobrow. Object Oriented Programming: Themes and Variations. *the AI Magazine*, 1985.

[SS89] John J Shilling and Peter F Sweeney. Three Steps to Views: Extending the Object Oriented Paradigm. In Norman Meyrowitz, editor, *proc OOPSLA*, pages 353 – 361, October 1989.

[Sto87] Michael R Stonebraker. Extending a Relational Database System with Procedures. In *ACM TODS*, Sep 1987.

[Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[ZM90a] S Zdonik and D Maier. Fundamentals of Object Oriented Databases. In SB Zdonik, editor, *Readings in Object Oriented Database Systems*, San Mateo, Ca, 1990. Morgan Kaufmann.

[ZM90b] SB Zdonik and D Maier, editors. *Readings in Object Oriented Database Systems*. Morgan Kaufmann, San Mateo, Ca, 1990.