Collection Views: Dynamically composed views which inherit behaviour

Peter M.D. Gray, Graham J.L. Kemp, Patrick Brunschwig
and Suzanne Embury *

Department of Computing Science, University of Aberdeen, King's College,
Aberdeen, Scotland, AB24 3UE

Abstract. Collection Views provide a means of coercing an object (or set of objects) of one type into an equivalent set of objects of another, more useful type. For example, in some circumstances it may be more convenient to view a shape object as a set of coordinate objects — in order to use a method to display the shape on the screen, for example. Collection views provide the DBMS with information on how to perform this coercion automatically. The DBMS can then adapt sets of values retrieved from some level of an is-part-of hierarchy, so that they are usable by pre-defined method functions defined on collections of parts. This adaptation is performed by composing a series of functions at run-time, rather than requiring the user to anticipate the queries that will be asked and create many stored classes to support them. Collection views thus allow stored data to inherit method behaviour defined in external applications, withour requiring the user to modify that behaviour or to store modified copies of the data. We have extended the P/FDM system to support collection views, and have demonstrated their utility by a bioinformatics example.
Keywords: Views, OODB, Bioinformatics, Inheritance

1 Introduction

In our research with scientific databases we have found the need to adapt external method code to work with sets of stored data values which are not at the chosen level of aggregation in an `is-part-of` hierarchy. In the course of doing this we have found the need for a kind of view over collections of objects at various such levels of aggregation. This *collection view* is distinguished by a particular kind of transitivity (or inheritance) based on the abstract `is-part-of` relationship. At first sight it fits the usual kind of subtype inheritance used in databases, but this turns out not to be so.

Consider, for example, a hierarchy of geographic entities at various levels, associated with spatial regions each of which encloses a collection of non-overlapping regions at the next level. For example, we could have countries related to their provinces, within which are districts within which are towns. On

* Present address: Dept. of Computer Science, Cardiff University, PO Box 916, Cardiff, Wales, UK CF2 3XF

an E-R diagram these correspond to distinct entity classes joined together by a hierarchy of one-many relationships. Suppose we want a view that gives us aggregate data for all the towns in a district, or all the provinces in a country, in order to compute demographic statistics or the economic profile of a region. In each case the relationship is implied by the transitivity of existing relationships. Thus, before applying the aggregate function which expects a set of towns as its argument, we make use of the "part-of" relationships to turn a province or a country into a set of towns. Unlike normal views, where we derive virtual (non-persistent) objects as instances of classes, here we derive relationships to sets of existing objects. We wish to avoid storing these derived relationships persistently, since they are based on transitivity, with many possible combinations that can quickly be derived at runtime.

As a related example, consider the case where we abstract down from a town to a geometric point representing the town centre. Once again, we might wish to form a view representing a derived relationship to a bag of such points.

The same considerations apply to the well known bill-of-materials hierarchy, based on `is-part-of` relationships between components at different named levels of sub-assembly, ranging from aircraft wings down to fasteners. This is, of course, very similar to the way in which large protein molecules are made up of chains of residues which are made up of atoms. We wish to define methods on sets of components that are related to a common higher-level assembly. These methods could compute bulk properties such as average cost or total weight, or else print frequency histograms, etc.

From a conceptual modelling viewpoint, the important idea is that we can define various methods on the derived sets to compute or display aggregate values. For example, a method could compute the centroid of a set of points. By exploiting the transitivity of the part-of relationships, it could be applied to a whole county or country. Alternatively, a weighted centroid could be defined on a set of town, with towns weighted according to population. Thus these methods are defined on abstract *collection classes* which are populated by derived relationships from different starting classes (see Figure 1). Thus the methods define a formal interface to the abstract class (in the OOP sense), specifying behaviour.

We have found this representation to be a great improvement on alternatives. Firstly, we have simplified the ER diagram by separating collection class interfaces clearly off to the side and avoiding many extra subtype arrows. Next, we have used runtime generation of composed functions to reduce the number of view definitions that have to be stored (for example adapted definitions of the centroid function for each level of the view hierarchy). Also, by using composed views to adapt the data at run-time, we have avoided having to have many stored subtypes containing derived data. These would have taken up space and needed specialised updates whenever the base data was changed. We also feel that this insight is a valuable contribution to E-R modelling. It arose first in a real protein database application, which we explain below, where our partners were scientists with a strong background in OOP techniques, but not in schemas!

We have implemented this on the P/FDM object database [11], which has a schema founded on the E-R Model with SubTypes. It uses functions to hide the distinction between derived and stored properties. In consequence, it is good at computing over collections of scientific data stored according to an E-R model. Functions are defined on classes, and thus behave like methods that do not change state. By integrating collection views with the query language we are able to use normal selection predicates to compute particular starting sets from which the collection class instances can be derived. Thus we can take advantage of the database software for processing large collections, and use the optimiser to plan use of indexes as appropriate.

We believe that the ideas set forward below could also be introduced into a well structured object query language such as OQL, or else through classes that pre-process calls to JDBC. However, it is not essential to do this. The real need is for a clear conceptual separation of this kind of view class in an E-R diagram, even when the classes are implemented in an *ad hoc* fashion as interfaces to Java or C++, or else to legacy code through CORBA object wrappers [13]. They are applicable where the data are structured around part-whole relationships, with various levels of repeated subunits, which is not uncommon.

Below, we shall introduce the protein schema from which the examples in this paper are drawn. Next, we compare and contrast the class/type hierarchies seen in OOP and object databases. We then describe the concept of collection views and how they relate to a particular kind of OO Design Pattern [8]. We then see how they are formally described and the rules for their use. Finally, we show how these have been implemented in the P/FDM object database, and we give examples of their use.

2 Protein structure domain: motivation for collection views

The idea of collection views presented in this paper was motivated by our participation in a collaborative bioinformatics project. As part of this, the project partners had several meetings to agree a schema for protein structure data. Of particular interest is the three-dimensional structure of proteins for which the basic data are available as atomic coordinates and distributed by PDB [2] on behalf of scientists world-wide. These data are of great importance to a large number of scientists in different disciplines, working on drug design, or understanding a new protein, or on its relationship to genome sequences.

Proteins are molecules which are found in all living organisms and perform a variety of biochemical functions. Each protein contains at least one polypeptide chain, which is made up of hundreds of amino-acid residues. The twenty different kinds of amino-acid residues found in proteins differ in size, charge, and various other properties. The order in which the residues occur in the sequence determines how the protein chain will fold in three dimensions, and it is the resulting three dimensional shape which gives a protein its particular biochemical activity.

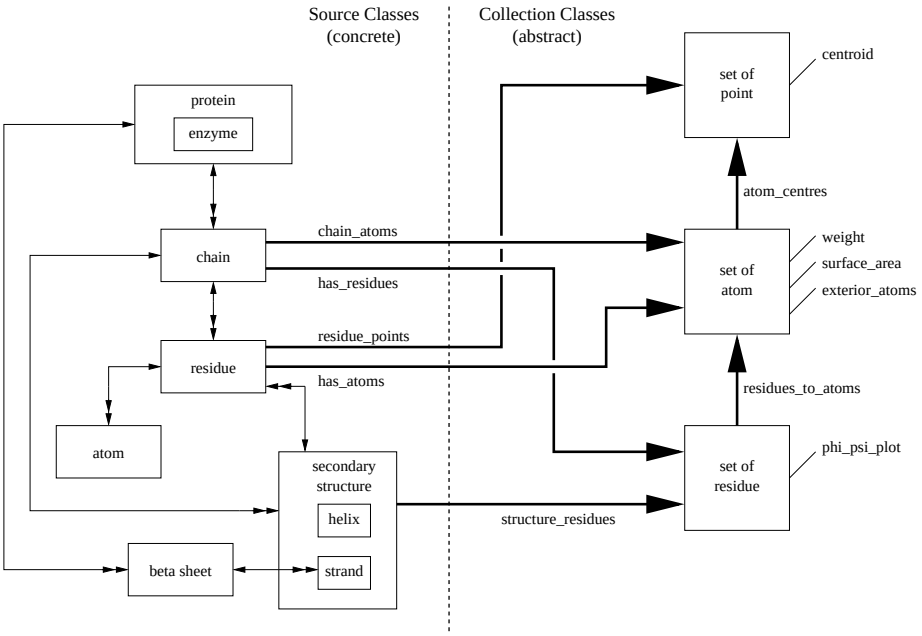

Fig. 1. Extract from the protein structure database. Thick arrows represent collection view adapter functions which populate collection classes, shown as the boxes they point to. Thin arrows represent relationships named in the schema. Nested boxes represent specialised subtypes of the enclosing box.

Protein structure can be viewed at different levels. The *primary structure* is the sequence of amino-acid residues in a protein chain, and is often represented as a string of characters with different letters standing for different residue kinds. At the next level, sections of protein chains fold into regular shapes (*secondary structure elements*) which are stabilised by hydrogen bonds between backbone atoms. Alpha-helices and strands are the most common examples of such structures. Each strand will lie adjacent to one or more others to form a beta-sheet. The *tertiary structure* of a protein is a description of its conformation in terms of the coordinates of its individual atoms. An extract from the schema diagram for our protein structure database is shown on the left side of Figure 1. A more complete description is given in [5, 10].

From the object-oriented viewpoint, one of the most significant aspects is that the same set of objects may be viewed at different levels of abstraction, ignoring differences of detail that are important at lower levels. Thus a helix object may be viewed as an ordered sequence of residues (ignoring 3-D coordinates) which may in turn be viewed as a set of residues (unordered) which can be viewed as a merged set of atoms with some derived property (weight, surface area). However, it is also meaningful to ask for the weight of a chain, or the weight of an individual residue. In designing the schema, we wanted to capture the notion

of kinds of protein fragment to which functions such as weight or surface area could be applied. Therefore, our early schema diagrams included classes like *set-of-residue* and *set-of-atom*, and in order to inherit functions defined on these classes we found ourselves over-using *is-a* inheritance arrows. This confused the schema diagram. It was also inappropriate since it was only *behaviour*, and not *structure*, that we wanted to inherit from classes like *set-of-atom*.

One thing that shows that the view hierarchy is not inheriting structure, is that the means of identifying instances changes as we proceed through the hierarchy, since we are looking at objects (sets of residues, sets of atoms) at different levels of aggregation. By contrast, in an *is-a* hierarchy, the means of identifying instances is the same at all levels and all specialised classes inherit the key definition defined at the top of that hierarchy. For example, a *student* instance is completely identified by virtue of being a person, as also is a more specialised *foreign student*. Thus, while our early schema diagrams showed classes like *chain* and *residue* related by an *is-a* relationship to *set-of-atom*, these classes were not in fact related in that way.

This led us to introduce a special kind of inheritance hierarchy for collection views which would clarify the schema diagram by considering collection view inheritance as if in "another dimension" (see right of Figure 1). This also saves one from a combinatorial explosion of subtype inheritance arrows, as explained earlier.

2.1 Objects in object databases and OOP: classes and class hierarchies

Another way to see the difference between the two kinds of inheritance is to look at how they are used by OO programmers and in OODB. Class declarations in an object-oriented programming language are used by the programmer to describe the structures of temporary objects holding evanescent (non-persistent) variables. They are needed to describe the computations performed using those variables; thus they emphasise procedural *behaviour* which may be inherited. For example, a C++ programmer will often introduce extra *slots* into an object which act as instance variables holding extra state information, to be used by a group of methods (procedures) that use these variables as a kind of shared workspace in their computations.

In contrast, a semantic data modelling approach leads to class declarations which describe data values which are of interest long term, emphasising their *structure*, i.e. the attributes and relationships of concepts in the domain which are to be stored. This structure may be inherited. We believe that a semantic data modelling approach leads to subclass-superclass relationships which are better suited to the integration of long term persistent data. The importance of the introduction of collection views is that it allows one to add extra functionality without confusing the subclass hierarchies.

We believe it helps to be aware of the difference between these two kinds of relationships, and we have a built a system in which we can formalise the difference. Thus, in this paper, we describe an investigation of *collection views*,

through which behaviour is inherited, in addition to having *is a subclass of* relationships through which both structure and behaviour are inherited. It turns out that *collection views* are another kind of *user-defined transitive relationship* with its own kind of transitivity, to add to those studied by [17].

3 Views in Database Schema Architecture

The original notion of views in databases comes from the definition of *external schema* or *subschema* in the ANSI-SPARC document of 1975. This described a database architecture with a central conceptual schema defining all data items, their types and constraints and relationships. The conceptual schema was viewed through alternative subschemas suited to different applications.

The ANSI-SPARC idea of a sub-schema was twofold. Firstly it only showed an application a subset of the available entity classes and data item names, which helped to provide privacy and access control. Secondly and more importantly, it served to isolate an application program from minor changes to the conceptual schema as it evolved. It might be that some of the data item names were changed. More significantly, some data items that were stored in an early version of the schema might be derived items in later versions. This might be because of a change in units used to record a value, or because the schema now stored more precise details from which the value could be derived. In this case the job of the subschema was to convert and map from the conceptual schema item values onto values expected by the application. We should now say that it helped to provide a *constant interface protocol*, much as an object presents to the outside world. It is this latter aspect that collection views are providing.

We consider a *collection view* to be a kind of adapter or coercion mechanism that takes any object of a given class or type and derives on demand a collection of objects of an adapted type, so as to fit a desired tool or method. It thus presents a constant interface, as just described, to a method which is possibly legacy code, adapting it to an evolving collection of objects. Each such "view" is basically a derived relationship, without the privacy aspects of a subschema.

Views in OODB

Early proposals for parametrised views were made by Abiteboul and Bonner [1]. A review of these and other kinds of object views is given by Chan and Kerr [3]. A commercial implementation of View expressions in OQL is described by Florescu *et al.* [6]. View expressions are queries which can use the full expressivity of OQL and which can derive subsets of an existing class, or else sets of constructed tuples which may contain object identifiers. They are intended to help in reusing cached results of OQL queries, and do not define their results as members of derived classes with method interfaces. In the MultiView system [14] a view is formed by applying a query operator to a source class (or classes) that restructures the source class's type and/or extent membership in order to form a virtual class with a type and extent derived from its source class(es).

A similar concept is *application views*, introduced by Fowler [7]. Similar types of view are provided in OPM [4] which is in use for genome databases. The essential point is that an application view is based on a one-to-one mapping from a subset of a persistent source class, selected by a predicate, onto a derived class. The selected objects form the extent of a new derived class with new methods. Thus the view perceives a subset of the database objects, for example proteins with no helices. Methods can be selectively chosen ("inherited") from those applicable to the source class, or derived methods added. However, the examples given concentrate on methods applying to individual instances and not to collections as a whole.

The advantage of the one-to-one mapping is that it makes it possible to have methods that appear to update objects in a view but which actually apply the update to the corresponding stored objects. This is not suitable for collection views, since the methods are intended to apply to the results of a one-to-many mapping. Also it is much more straightforward to make updates direct to the source class, or else through a separate application view.

It would be possible to integrate collection views with application views. For example, we could derive a collection view starting from a particular class in an application view. However, this is future work and it would be much harder to represent clearly in an E-R diagram.

Of course, it may be desirable to cache a derived view, where this takes a considerable effort to compute and many clients wish to access it. This then leads to interest in triggered rules to update the cached view following small changes to the underlying database. This is an active area of research [16] but it is largely orthogonal to this paper. Instead, we are concerned with uses of inheritance in views which support OOP interfaces to collections at different levels of aggregation taken from persistent object-oriented data, and we are aware of very little other work on this.

4 Collection Views as Adapters in OOP

Once one has the idea of a view as an adapter which puts a new face on the class and provides it with a desired interface then one can see a corresponding framework used in object-oriented programming. Figure 2 is based on the well known text on Design Patterns [8]. Using standard OM/T notation, it shows how a `TextView` class member is adapted to look like a `Shape` class member. More precisely, it acquires an *interface* like a Shape member. This is done by defining a sub-type `TextShape` which inherits the interface, but also includes an extra slot `text` pointing to the `TextView` member. Method definitions in the sub-type override those in the supertype and compute the desired interface values by calling up methods on the TextView member. Thus the data items (slot values) in the Shape object are filled in by calculation or copying from the TextView object and any other related objects, as defined in the body of the adapter method.

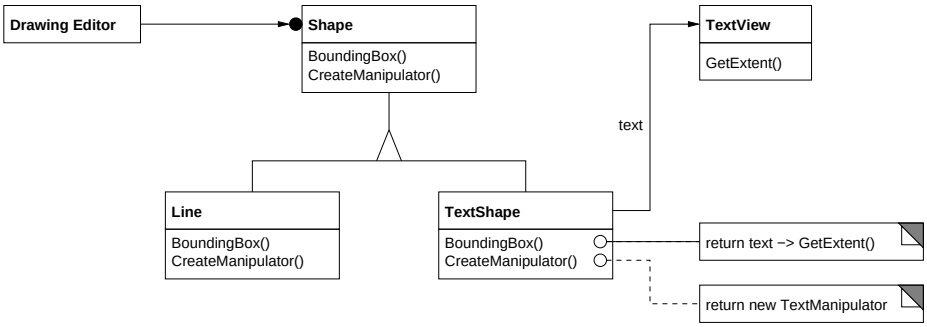

Fig. 2. Adapter pattern. BoundingBox requests, declared in class Shape, are converted to GetExtent requests defined in TextView. Since TextShape adapts TextView to the Shape interface, the drawing editor can reuse the otherwise incompatible TextView class. (Adapted from [8])

The interesting thing about this technique is that it can create the derived or adapted object of class Shape at run-time and the object need not persist after the running of the application. This contrasts with objects of the persistent class TextView, from which it is derived. This class will have all of its instances held in persistent storage (a database). We do not wish to store the derived adapter objects persistently, since they take up space and would need maintaining.

Thus we want to include classes like `TextShape` on the ER diagram, as shown on the right of Figure 1. Just as `TextShape` is not simply a supertype of `TextView`, so `set-of-atom` is not simply a supertype of `residue`. Instead, they are classes used to describe computational behaviour, where the behaviour is encapsulated in method definitions (operations). The important missing links are filled in by *adapter functions* (shown by thick arrows) which complete the connection to data stored in the persistent classes. (These hide the implementation of the adapter, which differs in detail from that in Figure 1). We must not read these arrows as *is-a*, instead we read them as *can be viewed as*. For example, "a chain can-be-viewed-as a set of residues", states that the collection class `set-of-residue` is derivable by adapters from the persistent class `chain`. It can also be derived by different adapters from other classes storing various segments of chain in the form of helices, strands, turns and loops.

Once we have used a collection view to derive an object of the collection class set-of-residue then we can of course apply to it any *method* defined on the class. Thus this class and its methods and adapters enable us to view helices and chains as a set of residues and thus to present a *constant interface* to a display method such as phi-psi-plot (shown in Figure 1).

Likewise, consider the collection class set-of-atom. This is obtainable by an adapter from an individual residue or from a chain. It is also derivable indirectly from helices by viewing them as set-of-residue! Thus we can have a view on a

view, forming an interesting kind of inheritance path (see below). Once again the view presents a consistent interface to application methods that can calculate either the weight or the total surface area of any adapted piece of protein.

5 Implementing collection views in P/FDM

5.1 Semantic data modelling approach

In our work we have taken a semantic data model database as our starting point. In particular, we use the Functional Data Model (FDM) [18], which was proved on the MultiBase project [15] for heterogeneous data integration, because of its ability to capture data semantics independently of storage details. We have implemented a database management system called P/FDM, which is like the original FDM and is used for data integration [12]. We continue to improve and extend it because of its solid theoretical foundation and amazing adaptability [9]. Its data manipulation language (Daplex) strongly influenced early OODBMS such as IRIS. In fact our current version of Daplex is very similar to the ODMG language OQL both in syntax structure, and in how it handles and passes object identifiers as values. It is recursively defined and strongly typed (see *http://www.csd.abdn.ac.uk/~pfdm/*).

Following FDM principles, we declare a class to be a subclass of another only if it truly is a special kind of the same general concept, e.g. an "enzyme" might be declared as a special kind of "protein". Even then, we only declare a new subclass if there are additional attributes or relationships which are meaningful for the subclass (e.g. enzyme classification number) but which are not meaningful for instances of the superclass in general. All attributes, relationships and methods defined on the superclass are also defined on all its subclass(es), including the key attributes which we use to identify object instances.

5.2 Schema Definitions for Collections on Concrete Classes

A collection view can be defined on any of the concrete classes or subclasses described in the database schema, or on another collection view. It takes any object of the given class or view type and derives on demand a collection of objects of an adapted type, so as to fit a desired tool or method. It may return a set of concrete instances declared in the database schema, or else a set of primitive values or tuples of such values (such as `point(x,y,z)`).

We have implemented this scheme on the P/FDM object database by adapting and generalising the system code (written in Prolog) that handles inheritance. Because P/FDM represents relationships as functions, which may be stored or derived, it is easy to represent adapter functions directly as P/FDM functions. In order to do this we identify the adapter functions F by introducing extra schema declarations with the keywords:

```
using F, an Atype can be viewed as a set of Btype
```

Schema Extract

```
declare protein ->> entity                    % Class declaration
declare protein_name(protein) -> string       % Function declaration

declare chain ->> entity
declare chain_id(chain) -> string
declare num_residues(chain) -> integer
declare chain_protein(chain) -> protein
define has_residues(chain) ->> residue        % Adapter function

declare residue ->> entity
declare position(residue) -> integer
declare name(residue) -> string
define has_atoms(residue) ->> atom            % Adapter function

declare structure ->> entity
declare structure_chain(structure) -> chain
declare structure_residues(structure) ->> residue
declare helix ->> structure

declare atom ->> entity

define weight(set of atom) -> float           % Function defined on a set
define surface_area(set of atom) -> float     % Function defined on a set
```

Collection View Declarations

```
using has_residues, a chain can be viewed as a set of residue
using structure_residues, a structure can be viewed as a set of residue
using has_atoms, a residue can be viewed as a set of atom
using atom_centres, a set of atom can be viewed as a set of point
```

Fig. 3. Schema extract and collection view declarations.

Example declarations are given in Figure 3. In one example we are able to use an existing stored relationship called *has_residues* as an adapter function, relating a protein chain to its many residues. In the general case an adapter function would be defined as a stored method, written in Daplex or in Prolog (possibly with callouts to C).

5.3 Composition of Adapter Functions

There is a kind of transitivity between adapter functions that allows us to compose them. Many people will look on this as a kind of *view inheritance*, so that a method defined on a more abstract view is applicable to an object belonging to a view lower down in the hierarchy. Thus for example the method centroid

Queries

```
for each c in chain such that
    weight(c) >= 1000 and weight(c) =< 2000
print(protein_name(chain_protein(c)), weight(c));

print(chain_id(any c in chain such that weight(c) > 1000));

for each h in helix such that surface_area(h) > 1000
print(chain_id(structure_chain(h)), surface_area(h));

for each c in chain
print(average(over r in has_residues(c) of weight(r)));
```

Extract from Typescript

```
| ?- daplex.
|: for the c in chain such that protein_code(chain_protein(c)) = "1CRN"
|:   for each r in has_residues(c)
|:   print(name(r), weight(r));

ASN 23.6
ALA 15.4
GLY 12.0     % more result values have been omitted

|: for each c in chain
|: print(chain_id(c), weight(c));

A 1006.2     % more result values have been omitted
```

Fig. 4. Example queries on collection views.

can be applied to a set of points, but it can also be applied to a set of residues, or even to a helix further down the hierarchy of thick arrows.

In order to implement this we use the adapters to perform what in programming language theory is a coercion or *type cast*. For example:

`y := sqrt(2)` is coerced to `y := sqrt(float(2))`

Thus if `h` is a variable holding an instance of type helix then `centroid(h)` really stands for (see Figure 1):

`centroid(atom_centres(residues_to_atoms(structure_residues(h))))`

Here `structure_residues` is an adapter function defined on class secondary-structure-element that delivers an object of class set-of-residue. In this way the various adapter functions are composed together into a derived relationship which will construct a set of objects to which centroid is applicable.

Where only a single coercion is needed, we can determine it from a knowledge of the source and destination classes, since we allow at most one adapter function (or arrow) between any two classes. However, where arrows form a path, there

may be alternative paths, and we explain how to resolve this later. Note that adapter functions are often multi-valued (i.e. deliver a set or bag) and that two such functions can be composed and will deliver a single flattened set as result. This is in accordance with Shipman's original Functional Data Model, because if instead they delivered a set of sets then it would not fit the next adaptor function.

We must stress the difference from normal method inheritance in C++ or Smalltalk, where the method that is inherited simply works on those slots in the object that are defined at its level. There is no question of adapting the values before applying the method. Likewise if one applies a method defined on a secondary structure to an instance of a helix (one of its subtypes) then the method just works on that part of the data structure which is common to all secondary structure objects and not specialised to a helix. This is subtype inheritance, and once again there is no notion of adapting values.

5.4 Collection Views in an Object-Relational Database

Let us consider how the examples in Figure 4 would look in an object-relational syntax, which may be more familiar. A syntax for Object-Relational DBMS with examples is defined in the well known text [19]. Here they introduce user-defined functions with arguments and discuss their inheritance behaviour. For example, they give the function *overpaid(employee_t)*, which can be overloaded to work on the more specialised argument type *student_employee_t*. Thus they give:

```
select e.name
from   emp e
where  overpaid(e);
```

What we are proposing looks similar, with the same syntax of a user-defined function applied to an entity type, but here we interpret it as a dynamically composed view by adaptation, instead of as an inherited function. Using this syntax we could write analogous queries to those in Figure 4:

```
select r.name
from   residue r
where  weight(r) > 20.0
```

Here `weight` does not apply directly to `residue`, which needs adapting via the adapter function `has_atoms` to generate a setof(atom). Similarly we could have:

```
select c.chainid
from   chain c
where  weight(c) > 220.0
```

Here the collection view mechanism would compose `has_residues` with `has_atoms` to make a suitable adaptor. Another example from the figure, this time adapting to surface_area, can be done using dot notation:

```
select h.structure_chain.chainid, h.surface_area
from   helix h
where  h.surface_area >1000
```

The examples with nested loops are awkward in SQL-3, and it is easier to separate the loops using auxiliary functions. Trying these examples makes one realise the great advantages of the referentially transparent FDM syntax which is so easily substitutable.

5.5 Extending Adapter Functions

Where an adaptor function maps a single item A onto a set of items of type B it is always possible to define a corresponding adapter that maps a set of A onto a flattened set of B. For example, when A = `residue` and B = `set of atom`, then the adapter from A to B is `has_atoms`. We now define the adapter `residues_to_atoms` to take a `set of residue` S and deliver the union of the results of `has_atoms` applied to each residue in turn: `union(map(has_atoms,S))`.

This is often what is needed when an operation (such as centroid) is defined over a set of values of one type (B), but is given a set of values of another type (A). Thus, an end-user may provide a specific adaptor for this set-to-set case (usually for efficiency) but where they do not the system will automatically extend the item-to-set adapter, as just described.

6 Checks on a Collection View Definition

We give below a series of checks on a syntactically well-formed view definition that enforce the rules and conditions discussed earlier.

```
using <adapter-name>, a <etype> can be viewed as a set of <rtype>
using <adapter-name>, a set of <etype> can be viewed as a set of <rtype>
```

- The result type `<rtype>` applies to each instance of the set formed by the collection view. Here *etype* must be a defined entity class name, or a named subtype of such a class, and *rtype* must be likewise, or else a tuple type (such as `point`) or scalar type. For any given *etype* there can be at most one collection view mapping onto a particular *rtype*. However, there may be other collection views, as long as they map onto different *rtype*s.
- The set expression forming the body of the definition of the adapter must compute a set of tuples or scalars or pre-existing object identifiers of type *rtype*, determined from the instance(s) of type *etype*. It must do this without updating any stored values.
- The form of view definition that starts from `set of <etype>` is used to define an adapter to coerce a set of values. It is needed for adapters between two collection views, as in the right of Figure 1. However, note that this set of values may just be the results of a Daplex expression or query delivering results of the right type. It does not have to be the output of applying another adapter function. For example:

```
for each c in chain
print(surface_area(r in has_residues(c) such that name(r)="TYR"));
```

- All of the collection view mappings on the database can be drawn as a graph G whose nodes are the entity types and whose arcs are directed from the etype node to the rtype node. Considered as a whole, the graph must be acyclic, and any newly defined arc that would make it cyclic must be disallowed (unless other arcs that make it so are first removed). Where the graph is not a pure tree, then any alternative paths between two nodes should be flagged, with a warning to the user to ensure these are equivalent.
- Where an adapter arrow joins collection view V1 to V2, then if it is composed with an adapter joining V0 to V1, we must ensure that the derived relationship between V0 and V2 is semantically meaningful. This will happen automatically if the individual adapters are based on is-part-of relationships in the physical world. If not, and we have not yet seen an example of this, then it could be overridden by another adaptor directly joining V0 to V2, with the desired meaning. Alternatively, it would be better to remove the adapter between V1 and V2 if it is not useful.

7 Formal Description

The introduction of collection views into P/FDM requires a modification of the standard method evaluation procedure. It is easiest to present the modified semantics of method evaluation by first presenting the standard semantics, followed by the new cases which must be added for collection views.

Without collection views, method evaluation in P/FDM is a simple two stage process:

- Bind to the correct method definition, following subtype inheritance links where appropriate.
- Evaluate that method definition.

With collection views, this process is extended by modifying the first step and adding an additional second step:

- Bind to the correct method definition, following subtype inheritance links and collection view links where appropriate.
- Coerce the arguments supplied to the method so that they are of the required type.
- Evaluate the method definition using the coerced arguments.

The third step is unaffected by the introduction of collection views.

We will now give a formal description of the semantics of method binding and coercion of arguments under collection views. The following definitions will be used as the basis for this presentation.

– A relation *MS* containing details of the methods known to the database. The tuples contain the method name, the first argument type, the remaining argument types, the result type and the unique identifier for the method definition:

$$MS \equiv \{< m, t, ts, rt, d >\}$$

– A relation *SLs* describing the subtype links known to the database. The tuples contain the subtype and the supertype respectively:

$$SLs \equiv \{< t_i, t_j >\}$$

– A transitive relation *subtype* over *SLs*:

$$(\forall t_i, t_j)\ subtype(t_i, t_j) \Leftrightarrow$$
$$(< t_i, t_j > \in SLs \vee ((\exists t_k) < t_i, t_k > \in SLs \wedge subtype(t_k, t_j)))$$

– A relation *CLs* describing the collection view links known to the database. The tuples contain the two types linked by the view and the definition identifier for the function which links them:

$$CLs \equiv \{< t_i, t_j, d >\}$$

This relation is extended as explained in section 5.5, so that for any t_i which denotes a simple entity type there will also be a corresponding tuple $< set\_of(t_i), t_j, d' >$ adapting the set of that type.

– For each function d_i, a relation D_i which represents the extent of that function.

7.1 Method Binding

In P/FDM, method binding occurs based on the type of the first argument supplied to the method call. The method definition used will either be that which is defined directly on the first argument type (where it exists) or that which is defined on the "lowest" supertype of that type. Formally, the relationship between a method f with first argument of type t and a method definition d is given by the expression:

$$(\forall f, t, d)\ binds(f, t, d) \Leftrightarrow$$
$$(< f, t, \_, \_, d > \in MS \vee (< f, t, \_, \_, d > \notin MS \wedge inherited(f, t, d)))$$

where *inherited(f, t, d)* is defined as:

$$(\forall f, t, d)\ inherited(f, t, d) \Leftrightarrow ((\exists t_k)\ subtype(t, t_k) \wedge < f, t_k, \_, \_, d > \in MS \wedge$$
$$\neg((\exists t_l)\ subtype(t, t_l) \wedge subtype(t_l, t_k) \wedge < f, t_l, \_, \_, \_ > \in MS))$$

These two definitions describe a common form of method inheritance for object databases. Note that we have included the negations of some of the conditions for bindings in later branches of the disjunction to indicate the relative precedences between the binding options. We never bind to an inherited function when a directly defined function is available.

The possibility of binding to method definitions through collection view links adds a third branch to the disjunction:

$$(\forall f,t,d)\ binds(f,t,d) \Leftrightarrow (< f,t,\_,\_,d > \in MS \vee$$
$$(< f,t,\_,\_,d > \notin MS \wedge inherited(f,t,d)) \vee$$
$$(< f,t,\_,\_,d > \notin MS \wedge \neg inherited(f,t,d) \wedge$$
$$((\exists t_k,p) < f,t_k,\_,\_,d > \in MS \wedge best\_coercion\_path(t,t_k,p)))$$

In other words, a function f with definition d can be applied to a value of type t if the "best" available coercion path exists from t to the type that the function f is defined on (t_k). We define the notion of a *coercion path* from type t_1 to type t_{n+1} as a sequence of collection view function definitions which can be composed to transform values from one type to another:

$$(\forall t_1,t_{n+1},d_1,\ldots,d_n)\ cpath(t_1,t_{n+1},< d_1,\ldots,d_n >) \Leftrightarrow$$
$$((\exists t_2,\ldots,t_n)\ clink(t_1,t_2,d_1) \wedge clink(t_2,t_3,d_2) \wedge \ldots \wedge clink(t_n,t_{n+1},d_n))$$

where *clink(t_i, t_j, d)* is defined to allow inheritance of collection view functions as follows:

$$(\forall t_i,t_j,d)\ clink(t_i,t_j,d) \Leftrightarrow (< t_i,t_j,d > \in CLs \vee$$
$$((\exists t_k)\ subtype(t_i,t_k) \wedge < t_k,t_j,d > \in CLs)^1$$

The process of binding to a function through a coercion path is complicated by the fact that there may be several possible paths leading from the starting type to a type on which a function f is defined. Where this is the case, the question arises as to which path should be followed, since this determines the result of the binding process.

A number of different options exist for selecting the "best" coercion path from a type. For example, it is possible to concoct complicated schemes which favour directly defined collection view functions over inherited ones. However, all such schemes contain an element of arbitrary choice and none are wholly satisfactory. We have chosen to follow a compromise approach in which the shortest path to an appropriate function definition is chosen. Collection view definitions which result in more than one "shortest" path should be flagged at compile-time, to warn the user that they need to be equivalent. For our purposes, therefore, we define the "best" coercion path from a type as follows:

$$(\forall t_i,t_j,p)\ best\_coercion\_path(t_i,t_j,p) \Leftrightarrow (cpath(t_i,t_j,p) \wedge$$
$$\neg((\exists p_k)\ cpath(t_i,t_j,p_k) \wedge \#p_k < \#p))$$

Here, the length of a sequence p is denoted by the expression $\#p$.

7.2 Coercion of Arguments

Once we have identified that a method can be "inherited" through collection view links, it is necessary to transform the arguments that have been given in

[1] We believe it is possible to extend this definition to include the case where the collection link is followed by a sub-type link instead of preceded by it but have had no demand for it.

the method call using the collection view functions so that they match the types required by the function.

Essentially, this stage in the process is one of identifying and evaluating a composed function which will perform the necessary coercion for each argument. Since it is possible that more than one, or even all, of the arguments to the method call may require some coercion, we must be ready to find a coercion path for each of them. By this stage, we have already identified the function definition to which the method call binds, and we can use this information to extract the required argument types from the metadata. That is, the relationship between a method call m, applied to argument types $< at_1, \ldots, at_n >$, and the coercion paths $< p_1, \ldots, p_n >$ necessary to transform those arguments for method evaluation is given by the following expression:

$$(\forall m, at_1, \ldots, at_n, p_1, \ldots, p_n)\ coercion\_paths(m, < at_1, \ldots, at_n >, < p_1, \ldots, p_n >) \Leftrightarrow$$
$$((\exists d, rt_1, \ldots, rt_n)\ binds(m, at_1, d) \wedge\ < m, rt_1, < rt_2, \ldots, rt_n >, \_, d > \in\ MS \wedge$$
$$best\_coercion\_path(at_1, rt_1, p_1)\ \wedge\ \ldots\ \wedge\ best\_coercion\_path(at_n, rt_n, p_n))$$

In other words, the coercion paths are given by finding the identifier (d) of the method to which the call to m binds, and then extracting the expected argument types ($rt_1, \ldots, rt_n$) for this method from the metadata. A coercion path must then be found (as defined by *best_coercion_path*) from each at_i to each rt_i, $1 \leq i \leq n$.
Each coercion path must now be applied to its respective argument value, to transform it into a value of the type required by the method that has been called.

The application of a coercion path $< d_1, d_2, d_3 >$ to a value x denotes the composed function:

$$d_3(d_2(d_1(x)))$$

Note that the first adapter function d_1 will always deliver a set of items, and that subsequent adapters will adapt sets of items to sets of items, as explained in Section 5.5. Here d_1 must be compatible with the type of the expression x, which may be a single entity or a set.

7.3 Implementation Strategies for Collection Views

We have defined the semantics of a function call involving one or more collection view links. In fact, there are two complementary ways in which this semantics can be implemented, corresponding to the two different levels of access to a P/FDM database. In P/FDM, queries may be written directly in Prolog, by including calls to database primitives to evaluate methods, create data, etc. This kind of access requires run-time handling of collection views. Alternatively, queries may be written in the Daplex data manipulation language and then compiled into the equivalent Prolog programs for execution. This kind of access allows the possibility of compile-time handling of collection views.

Run-Time Handling To implement the collection view mechanism at the Prolog level, we have modified the behaviour of the database primitives handling method evaluation. These primitives attempt to determine whether a collection view link can be used to bind the method call to its definition at run-time. The coercion paths required to transform the argument values are then identified and *interpreted* as a series of instructions for transforming the arguments.

Compile-Time Handling At the Daplex level, we handle collection views at compile-time by replacing method calls which require coercion with the calls that will perform that coercion at run-time. Effectively, we are *compiling* the coercion path, rather than interpreting it as we have described above. This approach requires a set of rewrite rules for queries which will insert the necessary additional function calls.

8 Persistence

It is important to realise that collection views are transitory and computed on demand in order to provide a set of objects with an appropriate interface to one or more utility methods (such as phi-psi-plot). They may look like persistent classes (which also have methods) on a schema diagram, but they do not persist. The only changes that can persist are those arising from allowable updates on the underlying objects of the view. It is possible to enlarge the definition of the view by adding extra functions to compute other derived properties; this is a form of schema evolution. However, it is not possible for a view class to have *stored* properties, because it is just an abstract class defining an interface. Furthermore, if the view class derives `set of <etype>` then the result must belong to the powerset of the stored values of `<etype>` in the current database state. Thus, if you compute a collection of of residue objects forming a set-of-residue, it cannot contain any new residue objects.

9 Conclusions and Discussion

Different people involved in designing a schema can have different ideas on the role and importance of inheritance, and this can impede progress in agreeing a common schema. By separating out the different ways in which inheritance is used, we can focus on the structural relationships among the data and think about what subtype-supertype relationships are needed for long-term persistence.

In the case of collection views there may be more than one path leading to the same desired target class (for example forming sets of atoms from helices). One is faced with the usual problem as to choice of path. One can trap the path at definition time and warn of ambiguity or disallow it if it is cyclic. We have chosen to put an ordering on the paths based on length, by using a breadth-first search. We currently prefer subtype "steps" in the path to adapter "steps". All this has been concisely and systematically implemented in Prolog. We also

require that paths between the same pairs of types should yield equivalent sets as results (but we have no automatic way to check this). More work is needed, but these problems are commonly encountered and do not invalidate the basic concept of collection views.

This description obviously raises some questions about possible improvements. In particular, is it possible to *lazily evaluate* the adapter functions so that they produce the members of a set (of atoms or residues etc.) one by one on demand instead of all at once, to save on storage allocation. The usual difficulty arises where an intermediate or final bag is generated from which it is necessary to remove duplicates, thus holding up the process. However, one often knows beforehand (from the definition of the adapters) that this cannot happen, and it would be nice to make use of this knowledge.

We have seen the value of being able to define views on object databases that return collections of objects. These collections can then be processed by an object-oriented program (possibly wrapping legacy code). The view is valuable in providing a consistent interface to this application code. It provides automatic casts which are commonly used and appreciated in programming languages, for example when working with mixed integers, reals and fractions. By defining collection views through inheritance of adapter functions in the schema we avoid a combinatorial explosion of view classes. This will also allow the schema to evolve while continuing to provide a consistent interface to the application, by modifications to the adapter functions. This is the proper role of a schema in supporting views.

Acknowledgements

We are grateful to our partners in the EC-funded BRIDGE project [10] for originally bringing this problem to our attention in discussions on a standard protein database schema from which sets of protein fragments could be derived. We are also grateful to the EU ERASMUS scheme for funding Patrick Brunschwig who was able to implement the design while visiting from Univ. of Zurich.

References

1. S. Abiteboul and A. Bonner. Objects and Views. In J. Clifford and R. King, editors, *SIGMOD 91 Conference*, pages 238–247, Denver, Colorado, May 1991. ACM Press.
2. F.C. Bernstein, T.F. Koetzle, G.J.B. Williams, E.F. Mayer, M.D. Bruce, J.R. Rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi. The Protein Data Bank: a Computer-Based Archival File for Macromolecular Structures. *J. Mol. Biol.*, 112:535–542, 1977.
3. D.K.C. Chan and D.A. Kerr. Improving one's views of object-oriented databases. Technical report, Glasgow University, Dept. of Computing Science, 1994.
4. I.A. Chen and V.M. Markowitz. *An Overview of the Object-Protocol Model (OPM) and OPM Data Management Tools.* Information Systems, Pergamon Press, 1995.

5. S. M. Embury and P. M. D. Gray. The Declarative Expression of Semantic Integrity in a Database of Protein Structure. In A. Illaramendi and O. Díaz, editors, *Data Management Systems: Proceedings of the Basque International Workshop on Information Technology (BIWIT 95)*, pages 216–224, San Sebastían, Spain, July 1995. IEEE Computer Society Press.
6. D. Florescu, L. Raschid, and P. Valduriez. Answering queries using oql view expressions. In Mumick and Gupta [16].
7. M. Fowler. Application Views: another technique in the analysis and design armoury. *Journal of Object-Oriented Programming*, 12, 1993.
8. E. Gamma, R. Helm, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.
9. P.M.D. Gray, S.M. Embury, K.Y. Hui, and G.J.L. Kemp. The Evolving Role of Constraints in the Functional Data Model. *J. Intelligent Information Systems*, 12:113–137, 1999.
10. P.M.D. Gray, G.J.L. Kemp, C.J. Rawlings, N.P. Brown, C. Sander, J.M. Thornton, C.M. Orengo, S.J. Wodak, and J. Richelle. Macromolecular structure information and databases. *Trends in Biochemical Sciences*, 21:251–256, 1996.
11. P.M.D. Gray, K.G. Kulkarni, and N.W. Paton. *Object-Oriented Databases: a Semantic Data Model Approach*. Prentice Hall Series in Computer Science. Prentice Hall International Ltd., 1992.
12. G.J.L. Kemp, J. Dupont, and P.M.D. Gray. Using the Functional Data Model to Integrate Distributed Biological Data Sources. In P. Svensson and J.C. French, editors, *Proceedings Eighth International Conference on Scientific and Statistical Database Management*, pages 176–185. IEEE Computer Society Press, 1996.
13. G.J.L. Kemp, C.J. Robertson, and P.M.D. Gray. Efficient access to biological databases using CORBA. *CCP11 Newsletter (http://www.hgmp.mrc.ac.uk/CCP11/newsletter/vol3_1/kemp/)*, 3, 1999.
14. H. A. Kuno and E. A. Rundensteiner. The MultiView OODB View System: Design and Implementation. *TAPOS*, 2:202–225, 1996.
15. T. Landers and R. L. Rosenberg. An Overview of MULTIBASE. In H.-J. Schneider, editor, *Distributed Data Bases*. North-Holland Publishing Company, 1982.
16. I. S. Mumick and A. Gupta, editors. *Proc. Workshop on Materialised Views: Techniques and Applications(Montreal)*. ACM SIGMOD, 1996.
17. A. Sathi, M.S. Fox, and M. Greenberg. Representation of Activity Knowledge for Project Management. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7:531–552, 1985.
18. D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.
19. M. Stonebraker and D. Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., 1996.