Integrating View Schemata Using an Extended Object Definition Language*

Mark Roantree[1] Jessie B. Kennedy[2] Peter J. Barclay[2]

[1] School of Computer Applications, Dublin City University, Dublin, Ireland.
{mark.roantree@compapp.dcu.ie}
[2] School of Computing, Napier University, Edinburgh, Scotland.

Abstract. *View mechanisms play an important role in restructuring data for users, while maintaining the integrity and autonomy for the underlying database schema. Although far more complex than their relational counterparts, numerous object-oriented view mechanisms have been specified and implemented over the last decade. These view mechanisms have served different functions: view schemata for object-oriented databases; object views of relational (and other) database systems, and the formation of federated schemata for distributed information systems. In the latter category there is still a significant amount of research required to construct a view language powerful enough to support federated views. Such a language (or set of languages) should support not only object views, but also a wrapper specification language for external information sources, and a set of restructuring and integration operators. Furthermore, with the advent of standard models and technologies such as CORBA for distribution, ODMG for storage, and XML for web publishing, these languages should be based upon, or cooperate with, these standards. In this research, we present a view mechanism which retains the semantic information incorporated in ODMG schemata, provide a set of operators which facilitate the restructuring and integration necessary to merge schemata, and provide wrappers to heterogenous systems such as legacy systems, ODBC databases, and XML data sources.*

1 Introduction

The concept of a federation of databases [17] is one where heterogeneous databases (or information systems) can communicate with each other through an interface provided by a common data model. In our case, the common data model is the ODMG model, the standard model for object-oriented databases since 1993 [6]. The most common architecture for these systems is as follows: data resides in many (generally heterogeneous) information systems or databases; the schema of each Information System (IS) is generally translated to an O-O format, and this new schema is called the *component* schema; view schemata are defined (subsets of the component schema) that are shared with other systems; these view schemata are exported to a global or federated server where they are

* Supported by Forbairt Strategic Research Programme ST/98/014

integrated to form many global or federated schemata. Please refer to [4] for a fuller description of object-based federated database systems.

This paper is structured as follows: in the remainder of this section, we provide a motivation and discuss related work; in §2, ODMG view and wrapper languages are presented; in §3, two schemata are integrated using the view language; in §4, a description of the implementation takes place; and finally in §5, some conclusions are offered.

1.1 Background and Motivation

The OASIS project [11] dealt with the federation of healthcare systems using an ODMG model as canonical model. While the ODMG model provides the semantic power suggested in [16] for federated data models, and offered a standard which suited forward interoperability, it lacked fundamental features required for integrating information systems. The model must (at least) provide a view system, a facility for wrappers to external information systesms, and integration operators which facilitate the construction of federated schemata in (structurally) different ways. Furthermore, all of these features should be provided in the data definition language of the database: in this case ODMG's Object Definition Language. In the case of the view specifications, the view language must be capable of retaining as much semantic information as possible, and thus, a view which results in a single virtual class is not sufficient. Instead, the view should resemble a subschema that retains information regarding inheritance and relationships between classes. Furthermore, this view mechanism must be powerful enough to support the restructuring of the subschema specification for any schema integration step. The contribution of this research is to provide extensions to the ODMG metamodel to support virtual subschemata and their components; extend the Object Definition Language (ODL) to support wrapper and *semantically-rich* view specifications; and provide an architecture and services to support the transfer, integration and display of view schemata.

1.2 Related Work

Our research requires the development of an object-oriented view mechanism which provides operators capable of restructuring the object-oriented schema hierarchy. A number of view mechanisms have emerged for object-oriented databases in the past. Older views mechanisms concentrated on the construction of a single virtual class, a representation which loses vital (for the schema integration process) semantic information. Although later view mechanisms such as [15][7][18] provide a means of defining virtual schemata, they use proprietary object models (which hinder interoperability) and do not include valuable restructuring operators such as those described in [10]. Recent work which offers a view mechanism can be found in [1] where they also adopt the usage of modern standards (XML) to offer an object-based view mechanism.

2 View Language Syntax

In this section, the syntax of the ODL$_v$ language is presented by illustrating the important production rules and providing a brief overview of operations. Space restrictions prevent a detailed discussion of the language, although this can be found in [14]. The ODMG provides the Object Definition Language (ODL) for defining base schema: we have defined the ODL$_v$ specification language for views, and the ODL$_w$ specification language for wrappers.

The main subschema production (not shown here) contains a number of `SchemaSegment` productions which in turn contains productions to import base and virtual classes, restructure imported classes, and export those classes which comprise the view schema. In the following BNF expressions, assume `identifier` to be a single identifier; `qualifier_dcl` to be an identifier with a qualifier (eg. class.property); and `double_qualifier_dcl` to be an identifier with two qualifiers (eg. viewname.class.property).

In *definition 1*, some idea is provided of the types of operations permitted at the property level: renaming of properties; hiding of properties, and the derivation of new attributes. Note that the first two operations are for attributes *and* relationships, and the third property is for attributes only.

Definition 1: property_dcl:
"**property**"
`( property_rename_dcl)?`
`( property_hide_dcl)?`
`( property_derive_dcl)?`

Each class declarator contains three productions: the `class_operation` production, the `class_rename` production, and the `class_filter` production.

Definition 2: class_dcl:
"**class**"
`( class_operation_dcl)?`
`( class_rename_dcl)?`
`( class_filter_dcl)?`

The `rename` expression can be used to rename classes. The `class_filter_dcl` is used to include an OQL query for the chosen class, and this topic of virtual class extents is discussed fully in §3.2. Each operator inside the `operation` expression falls into one of two categories: restructuring and integration operators.

2.1 Restructuring Operators

The view mechanism contains five restructuring operators: `aggregate`, `expand`, `subclass`, `superclass`, and `flatten`. Of the five operators, the first two operators, `aggregate` and `expand`, deal with association relationships, and the remaining three operators, `subclass`, `superclass` and `flatten`, deal with hierarchical relationships. For space reasons we will discuss three operators: `aggregate`,

`expand` and `subclass`. Of the remaining two, the `superclass` operator is used to create a new superclass from an existing class, and the `flatten` operator is used to *collapse* or *fold* a superclass into one of its subclasses.

Definition 3: class_aggregate_dcl:
`identifier :=` "**aggregate**"
`(identifier (,identifier)* )`
"**from**" `qualifier_dcl`
`(` "**as**" `identifier)?`
`(`"**to**" `identifier)?`

The `aggregate` operator (in *definition 3*) forms a new class from an existing set of attributes inside a single class. These properties are replaced with a relationship inside the original class. The operator $\triangleright$ denotes an `aggregate` operation, and the expression $B(b_0) \triangleright A(a_0, a_1, .. a_i)$ states that a new class A is formed from an existing class B using properties a_0 to a_i, with a_0 and b_0 representing the newly formed relationship properties.

Definition 4: class_expand_dcl:
"**expand**" `double_qualifier_dcl`

The `expand` operator performs the reverse operation to the `aggregate` operator. This operation is achieved by expanding the named relationship property inside one class into the full set of properties. The operator $\triangleleft$ denotes an expand operation, and the expression $B(b_0) \triangleleft A(a_0, a_1, .. a_n)$ states that the class A will disappear, and that properties a_1 to a_n are placed inside class B. As a side-effect of this operation, the relationship properties b_0 and a_0 will be hidden.

Definition 5: class_subclass_dcl:
`identifier :=` "**subclass**" `(identifier (, identifier)* )`
"**from**" `qualifier_dcl`

The `subclass` operator creates a new subclass from an existing class by removing n properties from the existing class and placing them inside the newly defined subclass. A new *ISA* relationship is formed between the new subclass and the original class. The operator indicates a `subclass` operation, and the expression $B \quad A(a_1, .., a_i)$ states that a new class A is formed from properties a_1 to a_i, and the new class A is a subclass of B. As a side effect the properties a_1 to a_i are hidden from the class B.

2.2 Integration Operators

The view mechanism contains five integration operators: `join`, `ojoin`, `superjoin`, `osuperjoin`, and `nulljoin`. For space reasons, we will cover two of them in detail, together with rules for managing conflicts. A full description can be found in [14].

Definition 6: class_join_dcl:

"**join**" `qualifier_dcl`
"**from**" `qualifier_dcl, qualifier_dcl`
"**on**" `identifier`
`(link_dcl)*`
`(with_dcl)*`

The operator ⊗ indicates a `join` operation, and the expression `C`←`B(`b_0`)` ⊗ `A(`a_0`)` states that a new class C is formed from classes A and B, using the joining predicate $a_0{=}b_0$. The `join` operator merges two classes by placing all properties from the original classes inside a single virtual class. If the two classes have more than one attribute in common, then the view mechanism assumes that these will have equal values (i.e. they are compared in a pairwise fashion). If not, it will be either necessary (in the view specification) to rename one attribute, elect one value in preference to the other, or redefine one attribute so that both values are equal. Conflict resolution is discussed shortly.

Definition 7: class_superjoin_dcl:

"**superjoin**" `qualifier_dcl`
"**from**" `qualifier_dcl, qualifier_dcl`
"**on**" `identifier`
`(link_dcl)*`
`(with_dcl)*`

The `superjoin` operator is used to connect two classes through the common overlap of a subset of their attributes. The `superjoin` is a composite operator: it is a combination of the `join` and `superclass` primitive operators. The result is three classes: the new superclass containing common attributes, one new subclass containing the remaining attributes of the first source class, and a second new subclass containing the remaining attributes of the second source class. The operator is used to indicate a `superjoin` operation, and the expression `C`←`B(`b_1`,..,`b_i`)  A(`a_1`,..,`a_i`)` states that a new class C is formed from classes A and B using i properties (b_1 to b_i) from B and (a_1 to a_i) from A. As a side-effect, both existing classes A and B, will lose these properties, although both will inherit them from the new superclass C. An outerjoin operation (`osuperjoin`) is also available.

Definition 8: link_dcl and *with_dcl:*

"**link**" `identifier := identifier`
"**with**" `(` `with_rename_rule` | `with_subclass_rule` | `with_prefer_rule` | `with_redefine_rule` `) ;`

In *definition 8* the `link` and `with` declarators are shown. The `link` declarator is used to bind two attributes with different names so that they can be used as a joining predicate. The `with` declarator is used where two attributes may be used in the joining predicate, but for reasons of semantics or structural differences, have different local values.

- The `rename` rule can be used to rename one of the attributes, and thus, remove it form the joining expression.
- The `subclass` rule is used only with the `superjoin` operation, and is used again, to eliminate those attributes from the joining expression. This time, it is achieved by placing both attributes in opposite subclasses (after a `superjoin`).
- The `prefer` rule is used only for secondary joining attributes. For example, if *PatientID* is the joining property, but both classes also contain an *address* property in common, this would form part of the join (or superjoin) operation. By using the `prefer` rule to favour one attribute over the other, the `join` operation will unite both attributes, and where a conflict of addresses takes place, one *address* value is preferred over the other.
- The `redefine` rule is used to redefine a property where both schemata are known to employ different numerical formats. For example, if one database uses *Fahrenheit* and the other *Celsius*, a numerical expression can be applied to values of one schema before the `join` operation.

2.3 The ODL$_W$ Language

For completeness, a brief overview of the wrapper language is provided now.

Definition 9: The ODL$_W$ wrapper statement
"**wrapper**" `identifier (RW)?`
{ `(class_dcl)+` }

A single `wrapper` statement is used to map one ODMG schema to a non-ODMG IS. Each statement map comprise multiple `class` declarations which represent the mapping of an ODMG class to some entity in the IS. A wrapper is marked as read-write (RW) only if it supports some form of locking, thus allowing update transactions.

Definition 10: class_dcl:
"**class**"{
`name_dcl`
(`attribute_dcl`)?
(`relationship_dcl`)?}

A `class_dcl` production comprises a name declaration and any number of attribute and relationship declarations. The local entity may map to more than one ODMG class where inheritance is not used in the local IS. The remaining productions provide mapping names for entities and their properties and are described in depth in [14].

3 View Language Usage

In this section a sample integration process demonstrates the usage of the view language. Assume that two local ISs have undergone the local schema transformation stage, and now exist as ODMG schemata. Our implementation assumes that data always resides in the local system, and that the ODMG schemata act as wrappers, defined using the ODL$_w$ language. When defining views, it is necessary to understand the issue of generating extents for virtual classes, and this is based on *pivotal* classes and the fact that an object preserving semantics is employed.

Pivotal Classes. A view comprises one or more *schema segments*, where a segment is a subset of the base (or some underlying virtual) schema. When a view segment contains multiple classes, one class must be nominated as the *pivotal* class, which subsequently determines the extent for all classes in the view segment in which it is contained. A pivotal class may use a *filter* to reduce the extent of objects for each virtual class in the view segment. If the subschema contains multiple schema segments, each segment must have its own pivotal class and extent. If no extent is specified, then the extent of the base pivotal class(es) is used.

Object Preserving Semantics. No new identifiers are ever constructed as a result of a new virtual object. Where a virtual class is derived from a single base class, the base class identifier is used, and the onus is on the view mechanism to provide access to the virtual class (and not the base class which has the same oid). Every attribute and relationship property in the view schema is connected to a base class equivalent, thus facilitating the updating of view properties. There is one notable exception. Since an attribute may be a collection of literals, it is unclear how individual elements in such a collection can be updated since they contain no identifiers. This weakness in the ODMG model was highlighted in [19], and must be rectified in order to facilitate updates.

3.1 Schema Integration with ODL$_v$

This research is based on the integration of healthcare systems and examples are used to define views on a Patient Administration Systems (PAS) and a HIV System. Note that for global (or federated) views, there cannot be base classes, as the architecture (described in §4) assumes that the Federated Kernel has no database schema, and is used only for importing schemata from component systems, and the definition of federated schemata. In *example 1*, a local view is defined to retrieve the set of Patients having bloodtype *A*, and the Consultants to which they have links.

Example 1. Defining a simple class filter:

```
database PAS
subschema PasLocal5 {
    SchemaSegment{
        ImportBase Patient, Consultant;
```

```
        Restructure {
            use Patient
            class {
                rename Patient as aPatient
                filter aPatient where BloodType ='A' }
        };
    Export PasLocal5.aPatient, PasLocal5.Consultant; };
};
```

The target database is named; this view (named *PasLocal5*) contains only one schema segment, which itself contains two classes of which the *Patient* class is the pivotal class.

In *example 2*, an example of an `aggregate` operation is shown. Some view expressions (database name and schema segment) have been omitted for space reasons.

Example 2. Defining an aggregate class:

```
subschema AddressData {
    ImportBase Person, Patient;
    Restructure {
        use Patient
        class {
            Residence := aggregate (Street,City,PostCode)
            from Patient as resides to PatLink }
    };
Export AddressData.Person, AddressData.Patient, AddressData.Residence; };
```

In *example 3* a join operation is used to merge two views: one from the PAS database and the second from the HIV database. The joining class is the *Patient* class and in this case it is assumed that *PatientID* (PAS) and *MRN* are identical (HIV); the *Quantity* field (used to indicate medication level) must be identical to join objects from both view extents, but *Quantity* is recorded in ounces in the PAS databases, and in grams in the HIV database. Finally, only Patients with bloodtype *O* are required in this view.

In *figure 1* both the local (imported) schemata, and the federated schema are illustrated, and in *example 3* the schema definition is presented.

Example 3. Defining the federated schema in figure 1:

```
subschema Fed {
    ImportVirtual Pas.Patient, Pas.Person, Pas.Consultant, Pas.Drug, Hiv.Patient,
Hiv.Person, Hiv.Episodes;
    Restructure {
        use Pas.Patient, Hiv.Patient
        class {
            join Patient from Pas.Patient, Hiv.Patient on PatientId
            link PatientId := MRN;
            with Pas.Patient.Quantity = Quantity / 0.03527;
```

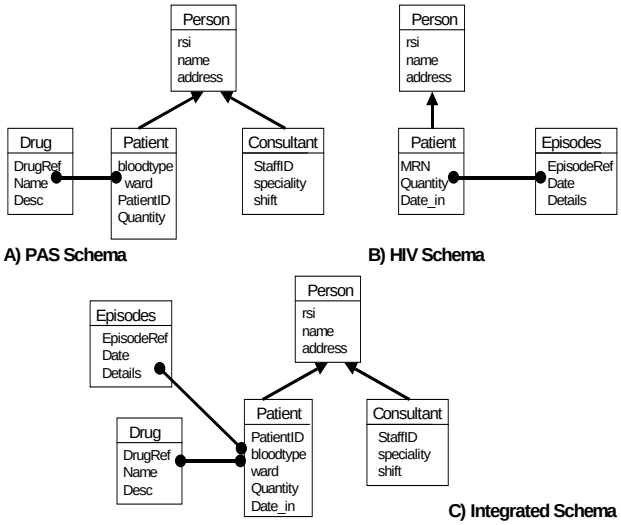

Fig. 1. *Integration of PAS and HIV Schemata.*

filter Patient where bloodtype = 'O'; }
};
Export Fed.Person, Fed.Patient, Fed.Consultant, Fed.Drug, Fed.Episodes; };

An exhaustive collection of sample views containing all of the operators, defined at both local and global levels are provided in [14]. In the following section, a discussion on the generation of virtual class extents is presented.

3.2 Generating Virtual Class Extents

To generate the extents for virtual classes (using an object-preserving semantics), it is necessary to take the extent of the base class, and apply some predicate to generate the virtual extent. Where a class is a specialised class, an extent is generated for the specialised class, and all of its superclasses. Note that for the *Patient* instance in *figure 1*, its *Person* abstract instance will have the same 'unique' identifier: it is the same object, but in different roles.

Before discussing how *virtual* class extents are generated, it is necessary to understand how a similar method would generate *base* class extents. It is necessary at this point to introduce the concept of *shallow* and *deep* extents: a shallow extent (E_s) is the set of all objects constructed specifically for that class; and a deep extent (E_d) is the set of all objects and subclass objects constructed for a given class. The full (deep) extent of a class C with i subclasses S is given below.

General Expression for Base Class Extent:

$$E_d(C) = E_s(C) \cup E_d(S_1) \cup E_d(S_2) \cup ... \cup E_d(S_i)$$

This generalised expression states that the deep extent E_d for the base class C can be expressed as the shallow extent E_s for C plus the union of the deep extents (ΣE_d) of all of subclasses S of C. Furthermore, each deep extent E_d can be reduced to a set of shallow extents E_s through recursive procedures. Thus, in our perception, a class may have multiple extents, and each shallow extent can be generated by applying simple "select *" queries against each individual class.

In general, the generation of extents for virtual classes is similar to that of base class extents, except that a filter function `f()` must be applied to the class which affects the class extent[1]. Every class in the segment is directly or indirectly connected to the pivotal class and has their extent affected by the filter.

General Expression for Virtual Class Extent:

$$E_d(\ f(VC)\ ) = E_s(\ f(VC)\ ) \cup E_d(\ f(VS_1)\ ) \cup E_d(\ f(VS_2)\ ) \cup \ldots \cup E_d(\ f(VS_i)\ )$$

However, unlike base class extents, the extent generating queries are applied to the classes to which these virtual classes are mapped. These may be base or virtual classes, and the mappings may be *one-to-one* or *one-to-many*, depending on the operation used to define the current virtual class. Thus, the shallow extent for virtual class *VC* is *the union of the extents of the classes to which it is mapped*, with each extent subsequently passed through the filter function `f()`.

The `join` operation in *example 3* results in a federated view comprising five classes, where two classes are join classes. The view mechanism executes a `join` operation on the *Patient* class (explicit in the view definition) and a subsequent (implicit) `join` operation on the superclass *Person* since both arguments (*Patient* classes) in the join operation have the same superclass. Where join classes have superclasses with the same name, they are eliminated from the view schema unless both are identical, or one is preferred over the other [14]. The deep extents for all five view classes are generated as follows:

$$E_d(f(\text{Fed.Drug})) = E_s(\ f(\text{Fed.Drug})\ )$$
$$E_d(f(\text{Fed.Episodes})) = E_s(\ f(\text{Fed.Episodes})\ )$$
$$E_d(f(\text{Fed.Consultant})) = E_s(\ f(\text{Fed.Consultant})\ )$$
$$E_d(f(\text{Fed.Patient})) = E_s(\ f(\text{Fed.Patient})\ )$$
$$E_d(f(\text{Fed.Person})) = E_s(\ f(\text{Fed.Person})\ ) \cup E_d(\ f(\text{Fed.Patient})\ ) \cup E_d(\ f(\text{Fed.Consultant})\ )$$

The shallow extents for view classes are taken as the extents for those classes to which they are mapped. Thus for the *Person* and *Patient* view classes this is a binary mapping, and both will have multiple extents. Thus, the shallow extent for the *Patient* class will be redefined as:

$$E_s(\ f(\text{Fed.Patient})\ ) = E_s(\ f(\text{Pas.Patient}) \cup f(\text{Hiv.Patient})\ )$$

[1] In base classes a filter function `f()` is also applied, but this will always be 'select *'.

Now we apply the concept of any class having multiple extents to virtual classes, and use it to simplify the generation of extents. Once the extents have been generated, the view representation can be used to hide or restructure properties to adhere to the view definition.

4 Architecture and Implementation

In this section we provide implementations details of the view architecture and services. The federated architecture used the ODMG model as a canonical model with the result that the component schema, its local views, the federated schema, and its global views, are all defined using the extended ODMG model and specification languages. The schema repository was extended to accommodate view metadata [12], and processors for both view and wrapper specifications were implemented using ANTLR [2], to define the productions, C++ to code the semantic actions for each production, and Versant [20] as the ODMG database implementation.

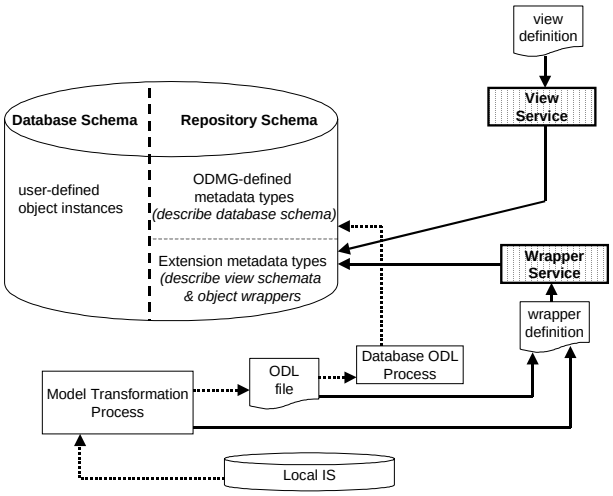

Fig. 2. *Service Architecture for local Information Systems.*

The process for integrating a new system requires a number of steps.

- An ODMG database is constructed with a schema representing the data to be shared with the federation. This hand-crafted step uses external research (such as [3][8]) to generate the component ODMG schema.
- A wrapper definition is specified using the ODL_W language which maps entities and properties from the ODMG schema to the schema of the participating IS. The wrapper specification is passed through the *Wrapper Service*

(see *figure 2*) which writes meta-objects to the (extended) schema repository of the ODMG database. The Object Manager can now extract data from the local IS when ODMG queries are generated.
- A different Object Manager is required for each data model. In our implementation, XML and ODBC object managers were constructed to build a federation of relational databases and XML data sources.
- Views are defined using the ODL$_v$ language, and passed through the *View Service* (see *figure 2*) which generates meta-objects as described in [14].
- An extraction process transfers views from participating systems (through their ODMG schema) to a federated kernel. This extraction process can employ one of four protocols: ODMG vendor-specific, ODMG generic, XML and CORBA protocols [13].
- At the federated kernel the ODL$_v$ language is again used to define views, although in this case, these specifications will use views imported from separate ISs.
- A *View Display Service* has been implemented to display views from local or global schemata.

Note that none of the three final steps are illustrated in *figure 2*. Outside the local operations shown in the illustration, is a *Federated Kernel*, which begins as an 'empty' ODMG database, used to import all of the locally defined views.

5 Conclusions and Future Research

Our research is focused on the creation of federations of healthcare systems. Traditionally, these systems have existed as large legacy systems often build using unlikely combinations of technologies (one such system was a COBOL application running on an old Unix platform). Recently, ODBC and XML interfaces have been built as wrappers to many of these systems, illustrating the fact that users now demand easy-to-use access methods, built using standard technologies. Our federated architecture uses a combination of different standards (ODMG and CORBA) to provide the federated middleware to facilitate the integration of these, still heterogenous, information sources. These systems required complex view mechanisms in order to facilitate the needs of global (or federated) schema construction. Our motivation to build our own view mechanism (view languages and services) was based on the fact that older systems employed proprietary models and software tools in federated schema construction. As a result, we specified a view layer for ODMG databases which allows both the creation of ODMG wrappers to multiple Information System types, and the construction of both local and global (semantically-rich) views. To demonstrate the view mechanism's usability, we implemented a prototype which contains all of the functionality required to construct federated schemata. The only limitation is that the Object Manager which resides between the ODMG Component Schema and the local IS can interact with ODBC databases and XML data stores only. Further Object Manager implementations are required for alternative ISs.

Current research is twofold: the incorporation of behaviour in ODMG views, and 'safe' update views. The former uses a combination of ODMG database technology, the ODL_v view language, and distributed object technology to make class methods available at the Federated Kernel. The latter looks at the wrapper definitions, and then at combinations of view definitions to determine which federated views can allow updates safely.

References

1. Abiteboul S. On Views and XML. *SIGMOD Record* 28:4, pp. 30-38, ACM Press, 1999.
2. ANTLR Reference Manual. *http://www.antlr.org/doc/* 1999.
3. Batini C., Lenzerini M. and Navathe S. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18:4, December 1986.
4. Bukhres O. and Elmagarmid A. (eds.), *Object-Oriented Multidatabase Systems*, Prentice Hall, 1996.
5. Cattel R. et. al. (eds.) (2000). *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann.
6. Cattell R. and Barry D. (eds), *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.
7. Dos Santos C., Abiteboul S. amd Delobel C. Virtual schemas and bases. *Advances in Database Technology* (EDBT94), pp. 81-94, Springer, 1994.
8. Fahrner C. and Vossen G. Transforming Relational Database schemata into Object-Oriented schemata According to ODMG-93. *Proceedings of 4th International Conference on Dedictive and Object-Oriented Databases (DOOD 95)*, pp 429-446, LNCS 1013, 1995.
9. Jordan D. *C++ Object Databases: Programming with the ODMG Standard*. Addison Wesley, 1998.
10. Motro A. Superviews: Visual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13:7, 1987.
11. Roantree M., Murphy J. and Hasselbring W. The OASIS Multidatabase Prototype. *ACM Sigmod Record*, 28:1, March 1999.
12. Roantree M., Kennedy J., and Barclay P. Using a Metadata Software Layer in Information Systems Integration. *Proceedings of 13th Conference on Advanced Information Systems Engineering (CAiSE 2001)*, pp. 299-314, LNCS 2068, June 2001.
13. Roantree M., Kennedy J., and Barclay P. Interoperable Services for Federations of Database Systems. To appear in *5th East-European Conference on Advances in Databases and Information Systems* (ADBIS 2001), Vilnius, September 2001.
14. Roantree M. Constructing View Schemata Using an Extended Object Definition Language. *PhD Thesis*. Napier University, November 2000.
15. Rundensteiner E. Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases. *Proceedings on the 18th International Conference on Very Large Databases (VLDB'92)*, pp 187-198, 1992.
16. Saltor F., Castellanos M. and Garcia-Solaco M. Suitability of Data models as Canonical Models for Federated Databases. *ACM SIGMOD Record,* 20:4, 1991.
17. Sheth A and Larson J. Federated Database Systems for Managing Distributed, heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22:3, pp 183-236, ACM Press, 1990.

18. Scholl M., Schek H. and Tresch M. Object Algebra and Views for Multi-Objectbases. In *Distributed Object Management*, Özsu, Dayel & Valdiurez (eds), pp. 353-374, Morgan Kaufmann, 1994.
19. Subieta K. Object-Oriented Standards: Can ODMG OQL be extended to a Programming Language? *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications*, pp. 546-555, Japan, 1996.
20. Versant Corporation. *Versant C++ Reference Manual 5.2*, April 1999.