A Flexible Object-Oriented Database Model and Implementation for Capacity-Augmenting Views*

Y. G. Ra, H. A. Kuno, and E. A. Rundensteiner
Dept. of Elect. Engineering and Computer Science
Software Systems Research Laboratory
The University of Michigan, 1301 Beal Avenue
Ann Arbor, MI 48109-2122
e-mail: ygra@eecs.umich.edu, kuno@umich.edu, rundenst@eecs.umich.edu
fax: (313) 763-1503
phone: (313) 936-2971

April, 1994

Abstract

In this paper, we identify key features required from OODB systems in order to provide support for advanced object-oriented tools that facilitate customized tool integration and transparent changes to database schemata. These features include multiple classification, derived classes, view schemata, dynamic reclassification, and flexible restructuring. Unfortunately, such features are currently not supported by commercial OODBMSs. In fact, in this paper we demonstrate that the object model assumptions underlying most OODB systems, namely, contiguous object layout, fixed typing and upwards inheritance, conflict with the identified requirements. We thus propose a flexible object-oriented modeling approach based on the object-slicing paradigm that overcomes these limitations. We describe a prototype of this object model that we have build on top of the commercial system GemStone to demonstrate the feasibility of our approach. This *MultiView* prototype realizes all features required for supporting capacity-augmenting views. We also compare and discuss the performance results of *MultiView* versus GemStone on the OO7 Benchmark. The system now serves as a platform suitable for implementing advanced OODB tools, such as schema evolution tools.

Keywords: Object-Slicing Paradigm, Multiple Classification, Dynamic Reclassification, Flexible Restructuring, Migration Paths to Technological Advances, Capacity-Augmenting Views, Object-Oriented Databases.

*This work was supported in part by the NSF RIA grant #IRI-9309076 and the University of Michigan Faculty Award Program. Harumi Kuno is also grateful for support from the NASA Graduate Student Researchers Program.

1 Introduction

As views in relational databases successfully provide programmers with the capability to restructure a schema so that it meets the needs of specific applications, a number of researchers have proposed view systems in the context of object-oriented databases (OODBs) [1, 12, 17, 28, 25, 26]. However, most of them fail to preserve the advantages of relational views in OODBs. Specifically, the proposed view systems typically cannot create views such that users perceive them as real database schemas. For example, most systems create only individual virtual classes rather than a complete schema graphs for customized views [12]. Even those researchers who have adopted the concept of *virtual schemata*, that is, customized view class hierarchies over the real schema, do not meet this goal. Their proposed *virtual schemata* behave differently from the real schema, especially regarding the updatability of the view classes and the inheritance of methods and attributes [1, 32]. The objectives of this paper are to identify what features must be provided by the underlying object model in order to support object-oriented view schemata that look and feel like (basic) object-oriented schemata and to demonstrate a general methodology for achieving them.

Along with view mechanisms, schema evolution is also an important issue in OODB research, both because data models are less stable than expected [18] and also because typical OODB application domains such CAD/CAM and multimedia information systems are not well understood and require frequent schema changes. In an earlier paper [24], we show that *capacity-augmenting* view systems (views that augment the information content of a database by adding stored data in addition to deriving data as a function of already existing data [33]) can be used to achieve transparent schema evolution. We also demonstrate that a schema evolution system built using a view approach bears many advantages over a version-based approach [3, 20] – it guarantees that uninvolved views will not be affected by schema change and it allows instance objects to be shared by old and new versions of a schema [24]. This clearly demonstrates the need for developing capacity-augmenting view mechanisms.

The goal of this paper thus is to design and build a powerful object-oriented (OO) data model and view mechanism with the following desirable properties: (1) A view class should look-and-feel like a real database class; (2) A *virtual schema* should behave like a real schema, including use of the same inheritance mechanisms for both base and virtual classes; (3) The underlying class hierarchy model should be flexible enough to integrate the customized virtual classes; (4) A view should be *capacity-augmenting* so that it can be utilized for implementing the advanced schema evolution capabilities.

To realize these properties, we identify the object model features required for such general view support to include a powerful class hierarchy incorporating virtual classes, capacity-augmentation, multiple classification to allow an object to be an instance of multiple classes, classification algorithms to integrate *virtual classes* with real classes, and method/attribute promotion for uniform inheritance. Unfortunately, such features are currently not supported by commercial OODBMSs [5, 10, 15, 21]. In this paper we demonstrate that the object representation assumptions underlying most OODB systems, namely, contiguous object layout, fixed typing and upwards inheritance, conflict with the identified requirements.

We then propose a novel object-oriented modeling approach based on the object-slicing paradigm that overcomes these limitations. The proposed object model is targeted towards supporting *multiple classification*, *capacity-augmenting views*, and the *dynamic restructuring* of objects and classes. The *object-slicing* approach is a technique to store the data of an object as a flexible object hierarchy structure rather than a contiguous object layout. Using this approach as a basic implementation paradigm, we have built our own inheritance mechanism for flexibility. To better support object-oriented views in our model, we support a comprehensive object algebra including *select*, *hide*, *union*, *intersect*, *refine* and *difference* that can be used to define virtual classes. We also show how these *virtual classes* are created and integrated in the context of our object-slicing model. A key characteristic here is that an object that is a member of both a base and a virtual class can now share data storage at the attribute level, even if materialized. In addition, our architecture is being used to implement a powerful object-oriented view management system (*MultiView* [26]) and for implementing a transparent schema evolution system (*TSE* [24]) to demonstrate the support our model offers for such systems.

In Section 2, we identify the object model requirements for supporting powerful view systems. Section 3 explains the implementation of our object model, and Section 4 the implementation of features that are used for constructing *virtual schemata*. Section 5 discusses how updates are handled in our implementation, and speculates on the overhead cost of our implementation by comparing the performances of *MultiView* versus GemStone using some of the OO7 Benchmark tests. Section 6 compares our work with related research, and Section 7 concludes this paper.

2 Requirements for Supporting Object-Oriented Views

In this section, we describe capabilities an underlying OODB must provide in order to support powerful capacity-augmenting views. Figure 1 is a graphical display of the required OODB features and their interrelationships.

2.1 Class customization

One major goal for the support of object-oriented view schemata is that the system's *class-restructuring capabilities* be powerful and flexible, as represented by the left-hand side of Figure 1. At a minimum, this goal requires that a view definition language must be provided, such as a comprehensive set of object algebra operators, to create customized versions of existing classes. Traditional views, however, cannot handle all cases of data restructuring; for example, they cannot extend virtual classes with stored data. However, such advanced capability is required if schema versioning based on object-oriented views is to be supported [4, 24]. Therefore, we extend the traditional class-restructuring capabilities of views to include the creation of *capacity-augmenting virtual classes* that augment the information content of classes on which the virtual classes are based on.

The introduction of query operators for the construction of virtual classes requires that additional features be supported by the underlying data model, as depicted in the left hand half of Figure 1. For example, *select* queries permit users to create virtual classes whose *extents* consists of a subset of existing classes' extents. We therefore must be able to maintain membership predicate information as part of the class definition of a virtual class. For views, (1) we must be able to specify queries on collections of object instances, and (2) the extents of virtual classes depend on the extents of other classes, so it is natural to *associate extents and classes* in the context of views. This facilitates the easy retrieval of a class's extent membership. We thus associate the concepts of both type and extent with a class [1].

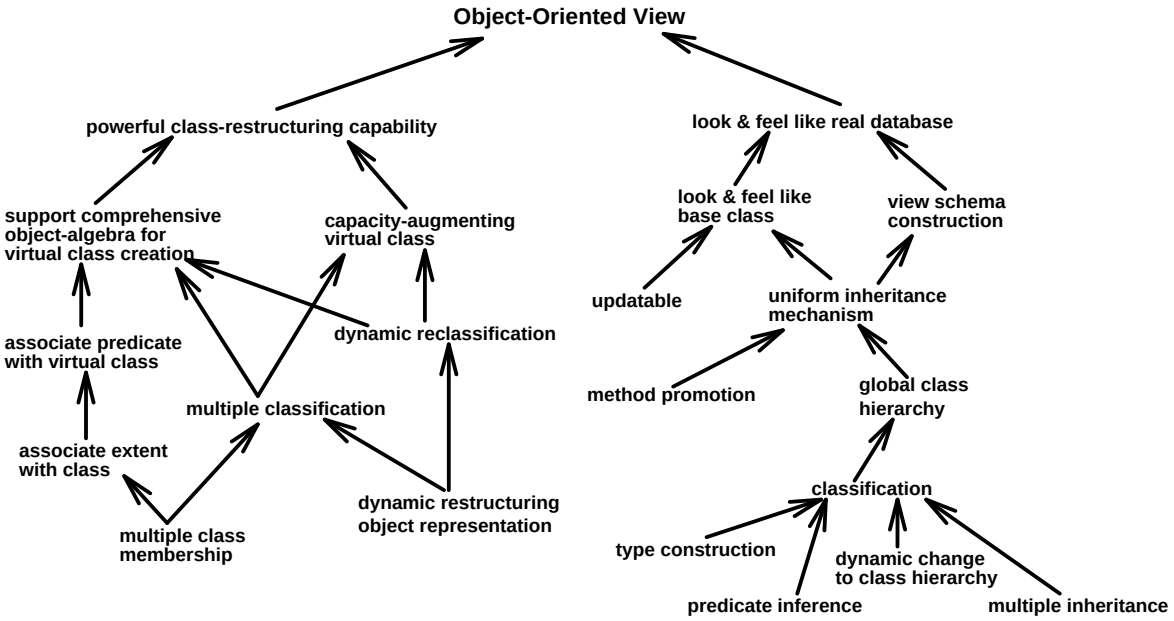

Figure 1: Requirements and Features for Desirable OO Views

Note that object instances that fulfill the predicate conditions of a *select virtual class* and thus become members of the new virtual class should continue to belong to the class from which they were selected. The data model must thus support *multiple class membership*, meaning that an object can be an instance of more than one class. Most available OODB systems do not currently support multiple class membership capabilities.

[1] We define a *type* as the set of methods and instance variables associated with a class, an *extent* as the set of objects that are instances of a class, and a *class* as consisting of a type and an extent. Note that although there is no general agreement on whether or not classes in OODBs should incorporate their own extents rather than requiring users to maintain their own collections of class-instances, several systems that follow this philosophy, including Orion and the system proposed by H. J. Kim [14], have been built. Furthermore, the proposed ODMG standard [2] recently formulated by several key OODB vendors also follows this approach.

Similarly, because object instances that belong to a virtual class should possess the types of both the virtual and the original class, the data model must provide *multiple classification*. This means that an object can be classified as an instance of several types at the same time. An object that is multiply classified could be interpreted as having multiple interfaces, each associated with a set of methods to which the object can respond. As is the case with multiple class membership, most existing OODB systems do not currently support multiple classification. In fact, most OODBs represent an object as a chunk of contiguous storage determined at object creation time, and adhere to the invariant that an object belongs to exactly one class (and indirectly to that class's superclasses).

Furthermore, an object instance could dynamically gain the type of a *(select) virtual class* if its data values change so that they fulfill the class's selection predicate. Similarly, it could dynamically lose the type of a *(select) virtual class* if its data values change so that they no longer fulfill the class's selection predicate. For these reasons, a view system must support the *dynamic reclassification* of object instances. We need this capability in order to let persistent objects flexibly gain and lose types, including both the data that they can store in their state as well as the set of methods to which they can respond.

The support of *capacity-augmentation* virtual classes, which means that virtual classes can be refined to augment the storage capacity of base classes by storing additional instance variables, also has implications for the data model. Again, because object instances should be able to gain and lose virtual types dynamically, while remaining members of their original classes, *dynamic reclassification* and *multiple classification* are required. Because *capacity-augmentation* involves the modification of the structure of object instances (e.g., so that they can store new instance variables), the system must also permit the *dynamic restructuring* of object representations. More specifically, object instances must be capable of efficiently changing their set of stored attributes or relationships over time.

2.2 A view should look and feel like an actual database

Relational views are utilized to operate upon a shared global schema as a customized database interface, and thus they appear identical to real schemata from the perspective of the database users. To preserve this property for object-oriented views (denoted as the *look&feel like real database* requirement in the right hand side of Figure 1), an object-oriented view should form a schema graph (generalization hierarchy) – rather than existing as an isolated individual virtual class disjoint from all other classes of the schema. The hierarchy, which we call a *view schema*, can consist of base classes from the base schema hierarchy and virtual classes that restructure the interfaces and customize the extents of the underlying base classes [2]. The view classes that compose a view schema are organized according to their class relationships in a generalization graph, thus forming a subgraph of the global generalization hierarchy.

In order to have a view schema act like a real database schema, the inheritance mechanism in place between base classes should also hold between the classes of the view schema. This is expressed as the *uniform inheritance* requirement in Figure 1. Traditionally, in object-oriented systems, whenever an object receives a message, the class of the object is first searched for the corresponding method, then if the method is not found there, the superclasses are searched upwards through the class hierarchy. Most OODBs adopt this *upward search rule* as an inheritance mechanism for reasons of efficiency and code reuse. We propose that this inheritance mechanism should be uniformly applied to all classes regardless of whether they are base classes or view classes. For example, suppose that we add a new method to a class C – a common practice in most object-oriented databases. Then this method should be inherited downwards to all subclasses – rather than upwards to all superclasses of the class C. None of the available OODB view systems, however, support uniform inheritance for both virtual and base classes [4, 8, 32].

The issue of inheritance mechanisms for view classes (referred to as "method resolution") has also been discussed by other researchers in OODBs [32, 8]. However, to the best of our knowledge, none of them achieves uniform inheritance semantics. Instead, they either advocate downward search for some virtual classes [32, 1] or else they statically compile all methods with each virtual class, thus forestalling dynamic inheritance. For example, suppose a virtual class projecting some methods from a base class were to be placed above the base class. Other systems' method resolution mechanisms require the virtual class to perform downward search to find the code block of the methods that are stored in the lower base class. We propose that this problem can be solved by *promoting the projected methods* from the base class to the virtual class. The methods are now defined in the virtual class, and the code block is actually stored there. Upward search now suffices for inheritance of the projected methods by the base class. This scheme, allowing the same inheritance strategy for both base and virtual classes, is discussed in more detail in Section 3.2.2.

[2] To simplify the remainder of this discussion, we call the classes of a view schema *view classes*.

Enforcing a uniform inheritance mechanism leads to the concept of a global class hierarchy (middle of right hand side of Figure 1). Because virtual classes are governed by the same inheritance mechanism as base classes, both base and virtual classes can be integrated into a single generalization global hierarchy, which we call *the global schema*. Once the global schema has been formed, view schemata can be constructed simply by selecting necessary classes from the global schema for a customized external view.

The global schema approach can be considered a *natural extension* of the relational view concept. In relational systems, we can build customized database views by first defining virtual relations via queries, storing the virtual relations in the global dictionary, and then constructing a customized view schema and database by selecting desired virtual and/or base relations. To parallel this relational approach in OODBs, we advocate that first virtual classes are derived via object-oriented queries, next the virtual classes are integrated into a consistent global schema with the base classes, and lastly external view schemata are constructed by selecting both base and virtual classes from the augmented global schema. However, one major distinction between object-oriented and relational views is the complex hierarchical structure of object-oriented schemata. In a relational database, the schema is a set of relations that are unrelated to each other (except via foreign keys). On the other hand, an OODB schema corresponds to a hierarchy of classes that are related to each other by generalization and aggregation relationships. Thus the global schema view mechanism is more complicated for OODBs because (1) the virtual classes must be correctly positioned within the global schema, and (2) selected view classes must be arranged into a view hierarchy. This is denoted as the *classification* requirement in Figure 1.

This classification requires the explicit capture of all class relationships between base and derived classes in terms of type inheritance and subset relationships (rather than only between base classes as is typically done in an object-oriented data model). Type and subset relationships of base classes are defined by the user when the base classes are created since base classes are explicitly created as subclasses of existing classes. Derived classes, however, are defined by the object query language without type and subset relationships being necessarily explicitly specified. So, the classification system must be able to *infer the type and extent of a virtual class* from the query definition to capture the relationships of the virtual class with other classes in the hierarchy.

Classification also requires a more *flexible, dynamically changeable class hierarchy* than is provided by currently available OODBs, most of which don't allow the insertion of a class into the middle of the hierarchy even if it wouldn't affect the types or extents of existing classes [5, 23, 9]. To integrate virtual classes into the global schema, we must be able to insert a newly created virtual class into the global class hierarchy at the proper position. This insertion will not affect the type or extent of any existing class, assuming that the new class is correctly classified. In addition, *multiple inheritance* is necessary for classification, since some virtual classes, such as the intersection class, must be classified as direct subclasses of at least two classes.

Finally, most views in relational systems are not updatable due to the update ambiguity problem, but this ambiguity can be overcome in the context of object-oriented systems due to the concepts of object identity [28] and of class-specific update methods. View schemata, being updatable, thus behave more like base schemata.

3 The *MultiView* Object Model and Its Implementation

Our goal is to design an object model that supports all required features outlined in Section 2, such as multiple classification, capacity-augmentation, etc. No current OODB supports all of the features we identified as necessary for view support in Section 2. In this section, we therefore provide a general implementation approach that we have designed and implemented to support *all* required features. This has been developed in the context of the *MultiView* project focusing on OO view technology; in fact, this implementation is *MultiView* version 2, which extends the first object model to include capacity-augmentation.

3.1 The object-slicing paradigm for supporting multiple classification

In Section 2.2, we identified multiple classification as a key requirement on the object representation underlying view mechanisms. Multiple classification is particularly necessary in a capacity-augmenting view system, since an object may have to be an instance of different virtual classes (as well as its base class) [3]. To the best of our knowledge, current OODB systems do not support multiple classification — with the exception of IRIS [10],

[3] While regular view systems (i.e., that do not support capacity-augmenting views) also must permit an object to be an instance of multiple virtual classes (in addition to its base class), note that virtual classes do not carry any additional stored data – and it is thus trivial to make the object a transient member of the virtual classes on access. This is no longer sufficient for capacity-augmenting views.

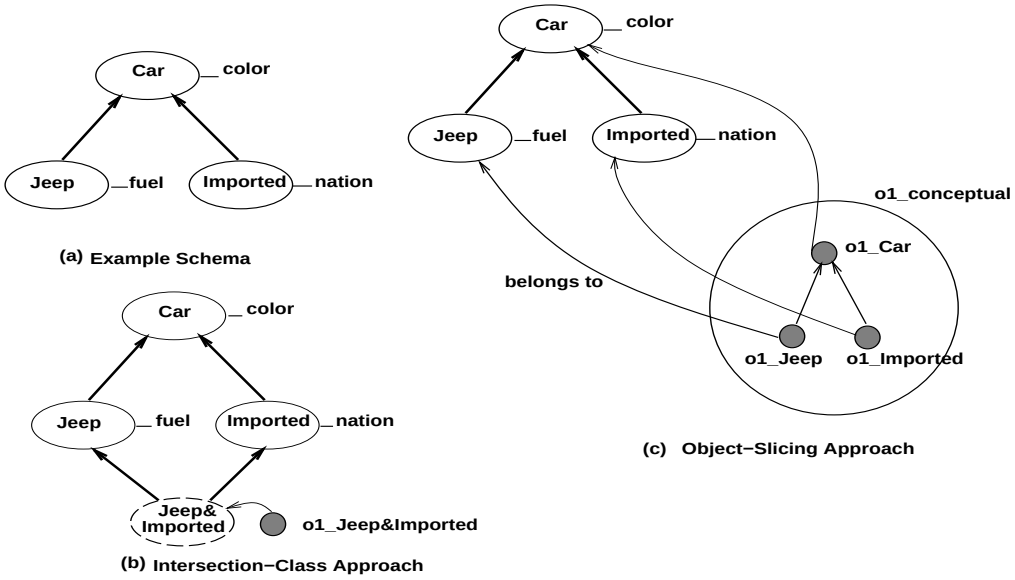

Figure 2: Two Approaches For Implementing Multiple Classification.

which is a functional database system that uses a relational database as a storage system, to store data from one object across many relations. Most OODBs typically represent an object as a chunk of contiguous storage determined at object creation time. They adhere to the invariant that *an object belongs to exactly one class only* — and indirectly also to all the class's superclasses.

We identify two approaches for overcoming this limitation of current OODB systems: (1) the intersection-class approach and (2) the object-slicing approach [19]. Both approaches provide explicit support for multiple classification in the object model. In the first approach, whenever an object is an instance of two classes, a new class that is an intersection of the two classes must be created to accommodate the instance. In the second approach, a real-world object corresponds to a hierarchy of implementation objects linked to a conceptual object rather than one contiguous sequence of stored data. The latter approach is more flexible, permitting the object to store the data associated with being an instance of multiple classes without the creation of these artificial intersection classes.

These approaches can be explained via an example. Given the schema in Figure 2 (a), we want to create a new car object $o1$ that has both type *Jeep* and type *Imported*. We cannot find a class in which to instantiate $o1$ without violating the invariant that an object belongs to exactly one class. To resolve this dilemma, the intersection-class approach would create a new intersection class *Jeep&Imported*, subclass of both *Jeep* and *Imported* classes, on the fly. We then could create $o1$ as a member of the new class (Figure 2 (b)). Furthermore, suppose that $o1$ were already a member of the *Jeep* class and we wanted to reclassify it dynamically as a member of the *Imported* class. This would require us to create a new object $o2$ as member of the *Imported* class, to copy all attribute values from $o1$ to $o2$, and lastly to swap the object identifiers of these two objects. If this dynamic reclassification had the goal of $o1$ not losing its membership in the *Jeep* class, then this would again cause the creation of the *Jeep&Imported* intersection class. Changes to individual object instances would thus force the awkward situation of dynamic schema changes. This approach does not provide support for true multiple classification, but rather simulates it by creating artificial intersection classes.

On the other hand, the object-slicing approach (Figure 2 (c)) would implement multiple classification by creating three objects to represent the $o1$ object, each of which carries data and behavior specific to its corresponding class. As we can see, the $o1$ object corresponds to a hierarchy consisting of the $o1_{Car}$, $o1_{Jeep}$ and $o1_{Imported}$ objects. We call the $o1$ object itself the *conceptual object* and the three type-specific objects that are linked to $o1$ the *implementation objects*. When the *current* class[4] of the $o1$ object is Jeep, the $o1_{Jeep}$ object represents the $o1$ object. Note that although the implementation objects are actual object instances of classes to which the object belongs, implementation objects are linked into a single conceptual object. From the user's perspective, implementation objects are transparent – the user perceives only the conceptual objects. Thus one of the functions our system must perform is overriding of identity and equality methods, so that each implementation object uses the object-identifiers of its conceptual object for these comparison operators.

[4] The *current* class of an object is the class through which the user is accessing the object.

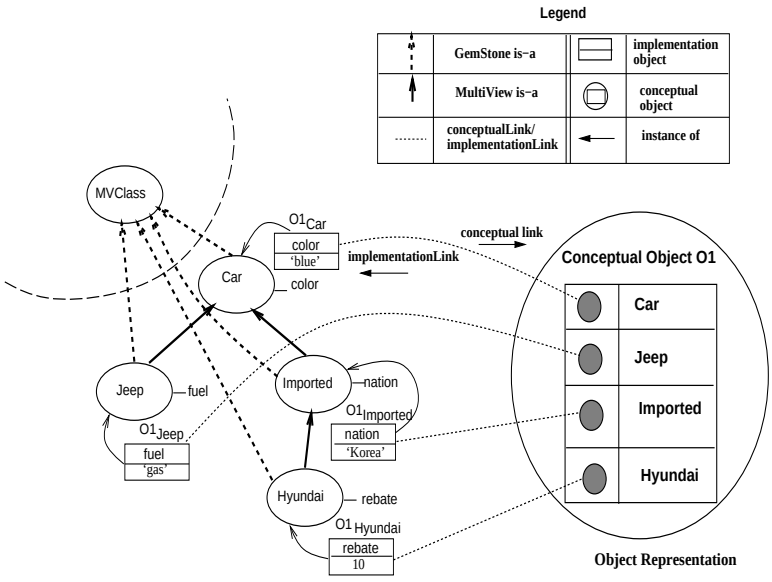

Figure 3: Implementing Object-Slicing Approach Using GemStone

In this object representation, dynamic restructuring and reclassification of an object involve only a small part of the conceptual object, namely only the implementation objects containing the directly affected data. In other words, typically only one implementation object may be touched during restructuring and reclassification, whereas in the intersection-class approach, the entire object must be removed and recreated. For example, a new class representing the intersection of the *Imported* and *Jeep* classes would be created and the object $o1$ would be reclassified to be a member of that class. Thus, these requirements are efficiently supported in the object-slicing approach.

The object-slicing approach also enables flexible resolution of name conflicts due to multiple inheritance. In traditional object-oriented systems, the inheritance resolution scheme must be determined when the system is developed. This is because the representation of an object is affected by the resolution scheme. Suppose a class inherits the same named attributes from both superclasses. Then, depending on the resolution scheme, storage is allocated to only one of the attributes or to both of them. For the object-slicing approach, the object representation is the same regardless of which resolution scheme is chosen for multiple inheritance. This means that we have the flexibility to adopt *various resolution schemes* dynamically.

Both approaches have their advantages and their disadvantages and a detailed comparison is presented elsewhere [24]. We have chosen the object-slicing approach as the basic architecture of our object model because an explosion of intersection classes is likely to be generated in the intersection-class approach. In the worst case, the number of intersection classes could grow exponentially with respect to the number of user-defined classes. Also, as demonstrated above, dynamic classification may require the creation and/or removal of intersection-classes on the fly. Note that it is transparent to a user whether the implementation is based on the object-slicing or intersection-class approach, because both approaches can provide the same interfaces to the user for data definition and manipulation by encapsulating implementation details.

3.2 Implementing the object-slicing approach using GemStone

3.2.1 The basic representation

Figure 3 describes the implementation approach of realizing this object-slicing model on top of **GemStone** [5]. In the figure, the thick dashed line arrows represent is-a relationships of GemStone, the thick solid arrows depict the is-a relationships of our system and the thin dotted lines link the conceptual object with its implementation objects. Each user-defined class of our system is implemented as a GemStone class. Every user-defined class is created as a direct GemStone subclass of `MVClass`, which is a meta-class we developed to provide functionalities

[5] More specifically, we use GemStone version 3.2, using the Opal interface. GemStone is registered trademark of Servio Corporation

```
method:  MVClass                                                    "MVClass defines doesNotUnderstand method"
doesNotUnderstand:  msg                                             "msg is a parameter"
{ 1: O1 := conceptualLink:  self.                                   "getting the conceptual object of self"
  2: superClass := pick from class variable supers of self.class.   "getting a superclass of self's class."
  3: superObj := O1 implementationLink:  superClass.                "getting an implementation object of the superclass"
  4: superObj msg. }                                                "delegate msg to superObj."
```

Figure 4: Pseudo Code of Inheritance Mechanism

and data structures for our system (Figure 3). One of the main functionalities provided by the `MVClass` is the message forwarding mechanism, which enables the development of an inheritance mechanism for the object-slicing approach. Details of the inheritance mechanism are explained in Section 3.2.2.

Every user-defined class C inherits the class variables of `supers` and `subs` from `MVClass`, which keep track of the direct superclasses and subclasses of C within our object-slicing model, respectively. For example, in Figure 3, the *Imported* class has the `supers` variable that references the *Car* class object, and the `subs` variable that references the *Hyundai* class object. These is-a relationships are denoted as thick dashed-line arrows in Figure 3. We keep track of subclasses using the `subs` class variable for three reasons: (1) the *global extent* of a class C, which includes the instances of all subclasses as well as of the class C, can thus be collected efficiently, (2) the class hierarchy can be restructured efficiently for virtual class integration, and (3) method polymorphism is supported, as described in Section 3.2.6.

Let us explain the implementation approach using an example. Object O1 in Figure 3 consists of a conceptual object and four implementation objects of type *Car*, *Jeep*, *Imported* and *Hyundai*, respectively. The implementation object of the *Car* class carries the data for the *color* attribute, that of the *Jeep* class the data for the *fuel* attribute, that of the *Imported* class for the data for the *nation* attribute and that of the *Hyndai* class the data for the *rebate* attribute. Each implementation object is implemented as an instance of a user-defined class, which is subclass of `MVClass`. Each carries a system-defined attribute `conceptualLink` pointing to its conceptual object O1 as well as class-specific data. The conceptual object does not carry any state but rather holds the references to its implementation objects. More specifically, the conceptual object O1 carries a *dictionary* holding associations of pointers to O1's implementation objects (`implementationLink`) and their respective classes [6] (depicted as a table inside the conceptual object in Figure 3). So, each implementation object can be referenced by its class through the dictionary associated with its conceptual object. For example, the implementation object of the *Imported* class for O1 can be found by sending the message "`implementationLink:` *Imported*" to the conceptual object of O1. In summary, an object is represented in our system by a hierarchy of implementation objects linked to each other via a conceptual object.

3.2.2 Inheritance in object-slicing model

Since the data associated with each conceptual object is now distributed, we had to develop our own inheritance mechanism for the object-slicing model. This means that objects of user-defined classes in our object-slicing representation can't directly use the GemStone inheritance mechanism. We build our own inheritance mechanism, in which implementation objects search upwards for methods through other implementation objects, on top of GemStone. This mechanism effectively replaces the pre-existing GemStone inheritance mechanism by enforcing every user-defined class to be a GemStone subclass of the system "meta-class" `MVClass` (in Figure 3). We have designed the `MVClass` meta-class to have the "`doesNotUnderstand:`" method of the *Object* class overridden so that it will perform upwards delegation of "unknown" methods and thereby achieve upwards inheritance. The Smalltalk-like pseudo-code of the delegation is presented in Figure 4.

The algorithm of Figure 4 is best explained by an example (again in Figure 3). Assume that an implementation object $O1_{Hyundai}$ for the *Hyundai* class receives a message for the method "color," which is an access method for the attribute "color" defined in *Car*. Because *Hyundai* is not a GemStone subclass of *Car*, the method "color" is not understood by the receiving implementation object. Then, the code of the `doesNotUnderstand` is executed as defined in Figure 4. In the first line, the conceptual object O1 of the receiving object is found. In the next line, the superclass of the O1's class `superClass` is found using the class variable `supers`. In our example, this would be the *Imported* class. In the third line, the implementation object of superclass `superObj` ($O1_{Imported}$) is found from the dictionary of the conceptual object. Finally the `msg` "color" is sent to the `superObj` $O1_{Imported}$. Since the "color" message is not locally defined in the *Imported* class, this algorithm

[6] This dictionary is a set of the associations indexed by class names. Thus, associations can easily be added to or removed from the dictionary.

is recursively called until it reaches the *Car* implementation object [7]. When the `msg` "color" is sent to the implementation object of O1 for the *Car* class, the `doesNotUnderstand` method is not executed because it now can respond to the "color" method. $O1_{Car}$ thus recognizes the method, performs the appropriate operation and returns the value "blue" for the element "color" to the user.

Because our system supports multiple inheritance [8], the class variable `supers` may contain more than one class as superclass. Then the second line of the above algorithm must be extended so that every path has to be searched. However, if there is more than one class that locally defines the desired method, then the well-known issue of naming conflict is encountered. Our solution is to force users to resolve the conflict by renaming, otherwise the invocation is rejected with an error message. This simple solution, also adopted by other OODB systems for dealing with name conflicts [34], could be modified to avoid such rejections by allowing, for example, (1) a fixed ordering of superclasses to determine the choice of the method resolution or (2) user-constructs indicating priority among superclasses for the purposes of name resolution.

3.2.3 Creating and deleting an object

Since the physical representation of a real-world object in our system is different from that of GemStone [9], we provide our own object creation and deletion methods `create` and `delete`. Suppose the `create` method is sent to the *Hyundai* class (Figure 5.a). Then, it would create first the conceptual object O1 (Figure 5), second the implementation objects for O1 for all classes lying in the path from the receiving class *Hyundai* to the *Root* [10] class, and finally it would link the implementation objects and the conceptual object. In our example, it would create the implementation objects $O1_{Hyundai}$, $O1_{Imported}$ and $O1_{Car}$ and connect them to O1 by adding the appropriate associations into O1's dictionary. The deletion of an object is achieved by sending the message "`delete`" to the object O1. This would first delete all the implementation objects of O1, and then delete the conceptual object itself [11].

3.2.4 Dynamic reclassification and restructuring

This section shows that we can achieve *dynamic reclassification* and *dynamic restructuring* with the help of the two methods `getType:` and `loseType:`. Suppose an object O1 is created as instance of *Hyundai* by the command "*Hyundai* `create`" (Figure 5 (a)). The object O1 can also be made an instance of *Jeep* by sending the message "`getType:` *Jeep*" to the object O1, which changes the representation of O1 depicted in Figure 5 (a) into that of Figure 5 (b). The method "`getType:` <*class*>" also creates the implementation objects of classes lying in the path from <*class*> to *Root* if they do not already exist. In this example, the implementation object $O1_{Car}$ already exists. Thus the creation of the implementation objects terminates at the *Car* class. New implementation objects are connected to the conceptual object of O1.

In the above example, the object O1 is restructured from the presentation of Figure 5 (a) to that of Figure 5 (b). One implementation object $O1_{Jeep}$ is created and linked to the conceptual object O1 by adding the new association ($O1_{Jeep}$, *Jeep* class) into O1's dictionary. Note that this dictionary is unordered and its associations keep track of pointers to implementation objects rather than actual data. This assures that the associations, which are homogeneous regardless of the size of data that the implementation objects carry, can quickly be added or removed. This *restructuring of the object representation* is *relatively efficient and simple* compared with the conventional architecture where each object carries all of its state information in a contiguous block of memory and belongs to only one class. The restructuring in the conventional architecture involves creating a new object of a target class, copying the values of the object to be restructured into this newly created object, copying the object identity of the old object to the new object by utilizing a object identity swap mechanism to preserve object identity, and destroying the old object. In short, our object-slicing approach enables efficient dynamic restructuring of the object representation.

We also support the reverse operation of `getType`, called `loseType`. For example, the conceptual object O1 is changed from Figure 5 (b) to Figure 5 (a) by sending the message "`loseType:` *Jeep*" to O1. The method "`loseType:` <*class*>" destroys the implementation objects of <*class*> and its subclasses.

[7] In our actual prototype, this is optimized by only accessing an implementation object if the desired method is locally defined for its class.

[8] GemStone only supports single inheritance; however we implement multiple inheritance as required for the support of view systems.

[9] GemStone uses the conventional approach of representing each object as a piece of contiguous storage

[10] The *Root* class is the system class that is the topmost class of every user-defined class hierarchy. It is not shown in the figures for simplicity, but it is implicitly assumed.

[11] Due to space limitations, the description we give here is simplistic and does not address referential integrity issues.

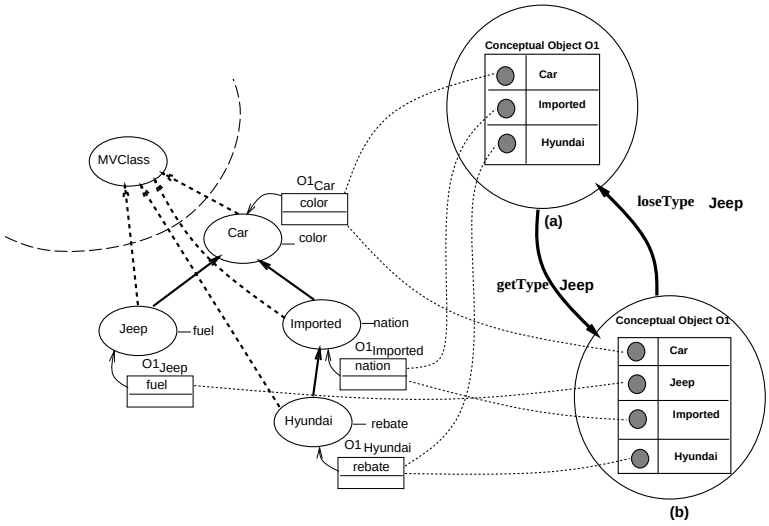

Figure 5: Getting and Losing a Type

It now can easily be seen that we can achieve *dynamic classification* by combining `getType` and `loseType` methods. For example, the O1 object of the class *Hyundai* can be reclassified as instance of *Jeep* by executing the two methods "`getType:` *Jeep*" and "`loseType:` *Hyundai*."

3.2.5 Switching between types

While an object can have multiple types (roles), at times we may want to treat an object in the context of one of its types. In a strongly typed language like C++, the type of a variable pointing to an object determines the *current type* of an object, while in a loosely typed language like Smalltalk and Lisp, the current type is fixed to the type of the class to which it directly belongs. Even if our system is built using a loosely typed environment, language, we can *cast* an object to another type taking advantage of our unique object-slicing technique. We provide a method, "`asClassOf: <class>`", which changes the current type of a receiving instance to the `<class>`. Internally, this method works as follows: (1) The message `conceptualLink` is executed to find the conceptual object of the receiving (implementation) instance; (2) the message "`implementationLink: <class>`", sent to the conceptual object, returns the implementation object of `<class>` type. In summary, our system requires the explicit call of the method `asClassOf` to cast an object to any of its types. For example, casting $O1_{Jeep}$ to *Imported* can be performed by sending the message "`asClassOf: Imported`" to $O1_{Jeep}$, (Figure 3).

3.2.6 Polymorphism

Polymorphism is often defined as the capability of objects to respond differently to a given message depending on their type. In C++, polymorphism is achieved via the *virtual function* concept; in Smalltalk, by fixing the current type of an object to its most specific type. Traditional OODBs, where there is one most specific class, can always determine the most specific method unambiguously. This is no longer true in our more flexible model. Polymorphism in our context means that when multiple classes have identically named methods, the method defined in the class with the most specific type should be invoked for the object. This requires us to introduce the explicit method "`asMostSpecific: <aMethod>`" to our implementation. The method "`asMostSpecific: <aMethod>`" searches downward from the current class and returns the implementation object whose type is the most specific from the object types having `<aMethod>`. For example, in Figure 6, suppose the object $O1$ receives a message, "`asMostSpecific: #dealer-margin`" and its current type is *Imported*. In this example, the implementation object $O1_{Imported}$ receives the message and forwards it to $O1_{Hyundai}$, because *Hyundai* is the class with most specific type defining the method `#dealer-margin` starting from the *Imported* class.

However, in some cases, multiple classification can cause ambiguity regarding polymorphism. Suppose the current type were *Car* in the above example. Following the same procedure, the $O1_{Car}$ object would receive the message and would search downward for the most specific type. Unfortunately, for this case, there are two

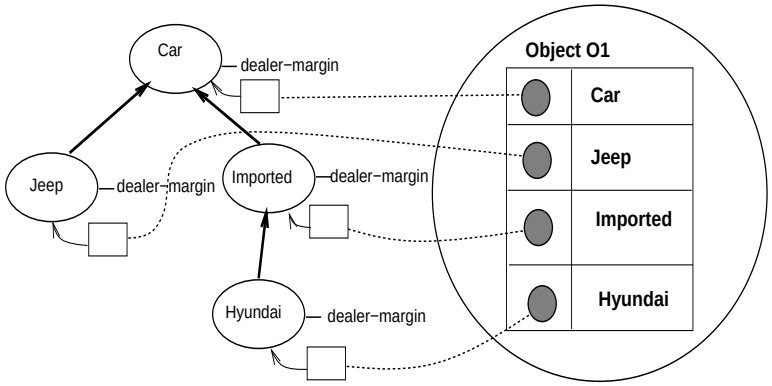

Figure 6: Example of Polymorphism in the Object-Slicing Approach

most specific classes, *Jeep* and *Hyundai*, that both define the method. This ambiguity doesn't happen in single-classification systems because each object always belongs to only one most specific class. When this ambiguity is encountered, our system gives an error message and the user must disambiguate by explicitly casting an object to the desired class (or at least to a class which is specific enough to allow only *one downward search path*). This approach parallels our solution for dealing with multiple inheritance.

4 Implementation of Object-Oriented View Mechanisms

4.1 Virtual class creation

In Section 2.2, we argued that the extent should be associated with each class to support, for instance, *select* virtual classes created by selection predicates. Since GemStone does not support extents, we have addressed this issue in our current implementation by associating with every class a collection class variable `extent` to keep track of all instances of the class type. We provide methods for object creation and deletion that assure the correct handling of the class extent. For example, an object can only be created by the predefined method "`create`", which automatically adds the created object into the collection representing the class extent. These predefined methods are implemented as methods of meta-classes of our view system, and thus are inherited by all user-defined subclasses. Of course, users are not permitted to modify them.

While the extent of a base class is a collection of real object instances, the extent of a virtual class is derived from the class(es) on which it is based on. In other words, the extent of a virtual class is typically expressed as a predicate on the extents of base classes. Besides the extent, the system also determines the type of the virtual class to find out the taxonomical position of the class. The upper half of Figure 8 shows the meta-classes of *MultiView* that provide administrative functionalities for base and virtual classes. One major function of the meta-classes is to provide polymorphism to the `getExtent` method, which returns the extent of base as well as of virtual classes. The `getExtent` method of both base and virtual classes returns the collection of the implementation objects kept by the class. The virtual classes' extents must, of course, be kept consistent with the base classes [12].

We present below the procedures by which a virtual class is created in our system, the type determined from the class definition, and the `getExtent` method implemented. We support an object-preserving object algebra for virtual class specification [26], composed of the following operators:

- The **select** operator defined by (<*class*> `createSelectClass:` <*new-class-name*> `query:` < *predicate* >) creates a *select* virtual class that is a GemStone subclass of the `SelectClass` meta-class (Figure 8). As a result, it inherits the methods and variables of the `SelectClass` such as the `getExtent` method, `sourceClass` and `pred` variables. The `sourceClass` class variable points to the receiving <*class*> and the `pred` class variable holds the <*predicate*>. The `getExtent` computes the extent by applying the `pred`

[12] Note that because *MultiView* does not replicate data, materialization in our system is significantly different from materialization in traditional systems. Our base and materialized instances actually share implementation objects when appropriate, thus simplifying view updates.

to the extent of the `sourceClass`. The type of the resulting class is unchanged from *type(<class>)*. This is achieved by placing the virtual class as subclass of the source class in our *MultiView* generalization hierarchy with no locally defined properties. In Figure 8, the select virtual class *BlueCar* is created by the command "Car `createSelectClass: #BlueCar query: [ :c | c color = 'blue' ].`"

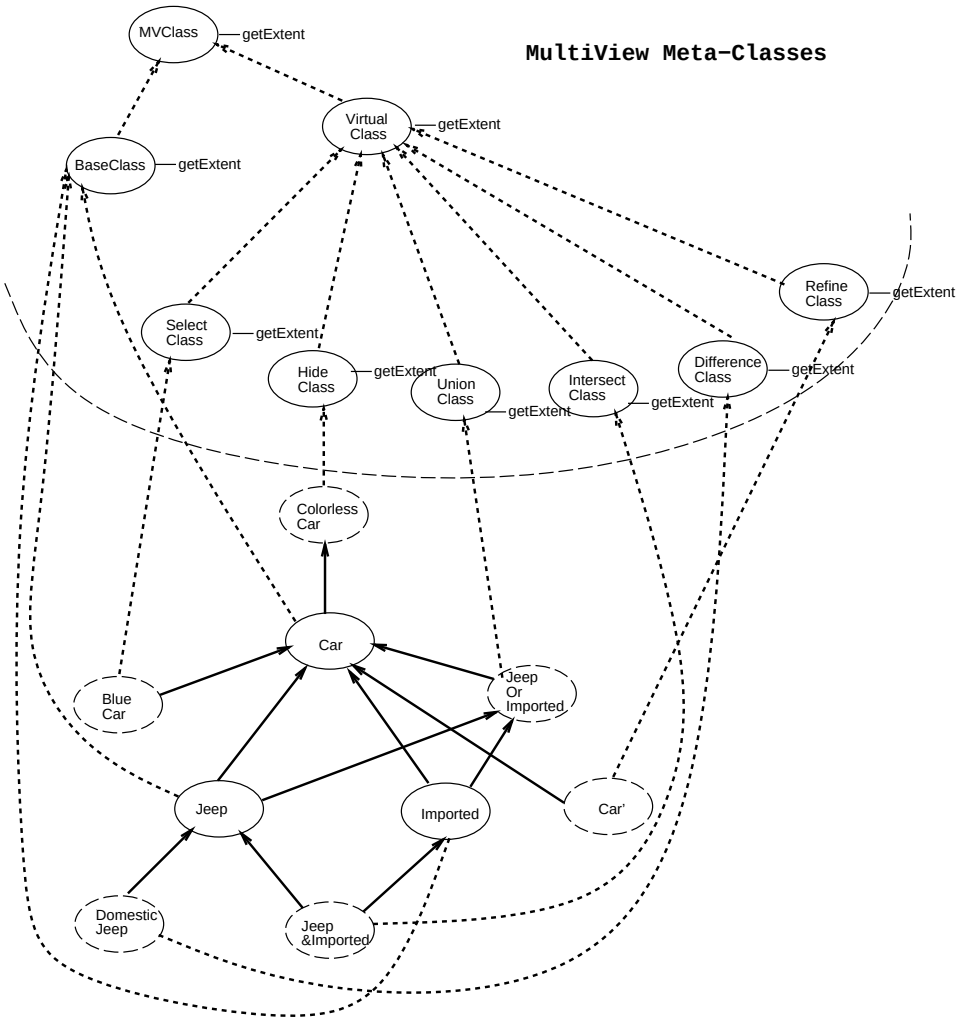

Figure 7: Virtual Class Implementation Using GemStone

- The **hide** operator defined by (<*class*> `createHideClass:` <new-class-name>] `hideProperties:` <*properties*>) returns a GemStone subclass of the `HideClass` meta-class, the type of which excludes properties listed in <*properties*> from the type of <*class*> while preserving all other properties defined for the type of (<*class*>). The resulting hide class is always a superclass of its source class, because its type is a supertype of the source class's type. This type is constructed by placing the new class as superclass of the source class and promoting *non-hidden* methods from the source class [13]. The `getExtent` method returns the same extent as that of the source class (pointed to by the class variable `sourceClass`). In Figure 8, the hide virtual class *ColorlessCar* is created by the command "`Car createHideClass: #ColorlessCar hideProperties: #[ color ]`".

- The **union** operator defined by (<*class1*> **createUnionClassWith:** <*class2*>) creates a GemStone subclass of the `UnionClass` meta-class. The type of the new class is the lowest common supertype of the input types, which is constructed by promoting the common properties of the source classes into the new class, if necessary. `getExtent` returns all objects which are instances of <*class1*> or <*class2*>. In Figure 8, the union virtual class *JeepOrImported* is created by the command "`Jeep createUnionClassWith: Imported nameOf: #JeepOrImported`".

[13] This type construction usually involves the movement of methods and attributes, as explained in detail in Section 4.2.2.

- The **intersect** operator defined by (<*class1*> `createIntersectClassWith:` <*class2*> `nameOf:` <*new-class-name*>) creates a GemStone subclass of the `IntersectClass` meta-class. The type of the new class is the greatest common subtype of the input types; and this type is achieved by placing the new class as a common subclass of the source classes. `getExtent` returns all objects that are instances of both <*class1*> and <*class2*>. In Figure 8, the intersect virtual class *Jeep&Imported* is created by the command "`Jeep createIntersectClass: Imported nameOf: #Jeep&Imported`".

- The **difference** operator defined by (<*class1*> `createDifferenceClassWith:` <*class2*> `nameOf:` <*new-class-name*>) creates a GemStone subclass of the `DifferenceClass` meta-class. The type of the new class is the same as the type of <*class1*>. It is constructed by placing the new class as a subclass of <*class1*>. `getExtent` returns the objects that are in <*class1*> but not in <*class2*>. In Figure 8, the difference virtual class *DomesticJeep* is created by the command "`Jeep createDifferenceClassWith: Imported nameOf: #DomesticJeep`".

- The **refine** operator defined by (<*class*> `createRefineClass:` <new-class-name> `withProperties:` <*property-defs*>) creates a GemStone subclass of the `RefineClass` class. The `getExtent` returns the same extent as that of the input class. The type of the new class is a subtype of the input type as all the old properties plus the new one are defined on it. This type is achieved by placing the class as a subclass of the source class with the refining properties. In Figure 8, the refine virtual class *Car'* is created by the command "`Car createRefineClass: #Car' withProperties: #[maker]`". The type of the new class is that of *Car* augmented by *maker*, the extent is the same as that of *Car* and it is classified as subclass of *Car*.

From the above discussion, we can also see that *multiple class membership* is vital for view systems, because an object may belong to any number of virtual classes whenever it satisfies the predicate membership conditions of its virtual classes.

While traditional view definition only derives new data as a function of existing stored data, we also support the extension of the database with new (non-derived) data for *capacity-augmenting* views. In particular, the `refine` operator creates a *capacity-augmenting* virtual class by allowing the parameter <*property-defs*> to contain new stored attributes as well as methods (derived attributes). When we refine a class with stored attributes, the representation of each object in the class has to be restructured such that the object can carry the information associated with the new stored attributes. This *dynamic restructuring capability* is supported by our underlying OODB implementation as we have demonstrated in Section 3.

4.2 Classification

Based on previous sections, it becomes apparent that most virtual classes "share" methods and attributes with their source classes. As stated in Section 2.2, we thus propose that the same inheritance mechanism should be utilized for both virtual and base classes, which would lead to the integration of all database classes into a global class hierarchy. As commonly assumed, if two classes in our model share some common property then they most both have inherited it from some common superclass, regardless of whether the classes are base or virtual. If the two same-named methods have been defined in two distinct origin classes, then we consider them to be distinct. Hence, with the integration of virtual classes into the global schema graph, we promote method code upwards to this single point of inheritance to preserve this inheritance property in the integrated graph. This approach supports true upwards inheritance of methods by both base and virtual classes, and also avoids the duplication of code and attributes. Besides enabling uniform inheritance, the global class hierarchy also facilitates the formation of view schemata (as described below).

A view management system must support a flexible classification mechanism in order to maintain this global class hierarchy. For example, the system must be able to make dynamic changes to the class hierarchy, possibly inserting a new class between two existing classes. We thus have developed *MultiView* global and view schema manager classes that provide their own classification methods. These include support for placing a class within the generalization hierarchy, for comparing two classes (using their types, extent-defining predicates, and derivation histories), and for promoting and demoting methods and instance variables (as classes are dynamically placed in the hierarchy). We thus have a mechanism in place to satisfy all the classification requirements described in Section 2. In particular, our system supports the ability to make dynamic changes to the generalization hierarchy and to construct a global class hierarchy with a uniform and consistent inheritance mechanism. In this section, we show how such properties are achieved under the object-slicing approach.

Recall that the `MVClass` meta-class uses class variables to maintain the sets of the direct super- and subclasses of each class, and representing the super- and subclass relationships between classes. Our inheritance mechanism (discussed in Section 3.2.2) uses these class variables to determine the generalization/specialization relationships

of classes in the hierarchy. Thus, in order to insert a new class into the hierarchy, all we must do is reset the appropriate class variables of the classes directly above and directly below the new class. We provide a dynamic method resolution and code migration scheme that assures that no side-effects arise from this.

4.2.1 Algorithm for dynamic insertion of classes into the generalization hierarchy

As new virtual classes are created in *MultiView*, they are automatically incorporated into the system's global class hierarchy so that it is below all classes that subsume it and above all classes that it subsumes. We have developed a classification algorithm for *MultiView* based on type lattice theory that successfully solves this classification problem [25]. The automatic classification of virtual classes into the global generalization hierarchy is a unique feature of *MultiView* with other view systems either avoiding the issue of integration and/or requiring manual graph manipulation [25, 4, 12, 13, 14, 28, 32]. After a virtual class has been created, the MultiView classifier performs the following steps: First the system generates intermediate classes to serve as unique point of inheritance, if warranted by the addition of the new class into the hierarchy. Next, the system calculates which existing classes should be direct children and parents of the new class, and updates the generalization hierarchy's edges, removing any redundant edges. Lastly, the system promotes the code of any methods or attributes that now should be located at a new superclass.

The introduction of intermediate classes above is our solution of the following two problems of the automatic integration: (1) the inheritance mismatch problem in the type hierarchy, and (2) the problem of integrating *is-a* incompatible subset and subtype hierarchies into one class hierarchy. The inheritance mismatch problem occurs when a new virtual class is created for which there is no existent correct place in the global type hierarchy, as illustrated in Figure 9(a). The *ColorlessHyundai* class cannot be placed directly between any existing classes in the hierarchy because there is no class whose type is a strict subtype of the *ColorlessHyundai* type, yet the *ColorlessHyundai* class shares properties with other classes in the hierarchy. For example, *ColorlessHyundai* cannot be placed either above or below the *Imported* class, because although it shares the `nation` property with the *Imported* class, *ColorlessHyundai* does not have the `color` property and *Imported* does not have the `rebate` property. The *is-a* incompatibility problem results when the subtype and subset relationships between two or more classes conflict. For example, when a virtual class's set content may be lower in the corresponding set hierarchy than its place in the corresponding type hierarchy then neither can be classified as a direct superclass nor a direct subclass of the other.

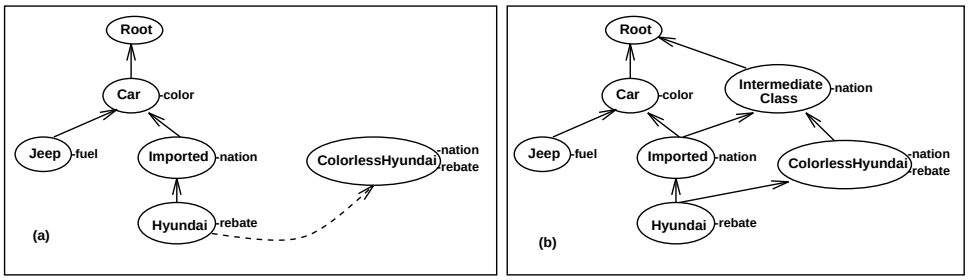

Figure 8: Inheritance mismatch problem and solution.

Our solution to both these problems is to create and insert additional intermediate classes into the global class hierarchy, as shown in Figure 9(b). For instance, the *Intermediate Class* class holds the *nation* property that can now be inherited by both the *Hyundai* and *ColorlessHyundai* classes. The addition of intermediate classes ensures that a complete global schema can be calculated for any configuration of base and virtual classes - with the inform inheritance mechanism outlined above [25].

When inserting a new class into the global schema, our classifier creates a set of intermediate classes that is both necessary and sufficient to guarantee the closure of the resulting class hierarchy [25]. The creation of intermediate classes is a key feature for achieving the object-slicing paradigm, because it assures a unique source of inheritance for all properties. Each property in the database, whether method or attribute (instance variable), is associated with exactly one class, which serves as the point of inheritance for that property. Thus when a conceptual object in our object-slicing implementation acquires an implementation object of a class, it is guaranteed that the class of the new implementation object is the point of inheritance for any properties locally associated with the class. Furthermore, there is a unique location for each stored attribute, namely, with the implementation object associated with the class that represents this unique point of inheritance.

After the class hierarchy has been prepared by the insertion of necessary intermediate classes, the new virtual class can easily be inserted into the global schema by identifying all direct *is-a* relationships between the new class and the other classes using a depth-first downwards traversal algorithm. At each node, we apply the *subsumes* function to determine the relationship between the newly inserted class and the existing class [17]. The resulting global schema incorporates the virtual class in a consistent and efficient manner, as shown in [25].

4.2.2 Method and attribute promotion

As described in Section 2.2, in order to preserve uniform upwards inheritance, it may be necessary to promote methods and/or attributes from a subclass to a new superclass. This will ensure that the property will be located above all the classes that inherit that property. For instance, in Figure 9, the `nation` property is inherited by both the *Imported* and *ColorlessHyundai* classes, and thus should be located at the *IntermediateClass*. When methods or attributes are moved from an existing class to a new superclass, the system performs the following tasks (also illustrated by an example in Figure 10): First the new superclass is extended to include the migrating (to be locally defined) methods and/or attributes. Next, the migrating methods and attributes are removed from their original class. Finally, each implementation object instance of the original class is *split* into an implementation object of the modified original class and an implementation object of the new superclass [14]. These new implementation objects are linked together by their corresponding conceptual objects.

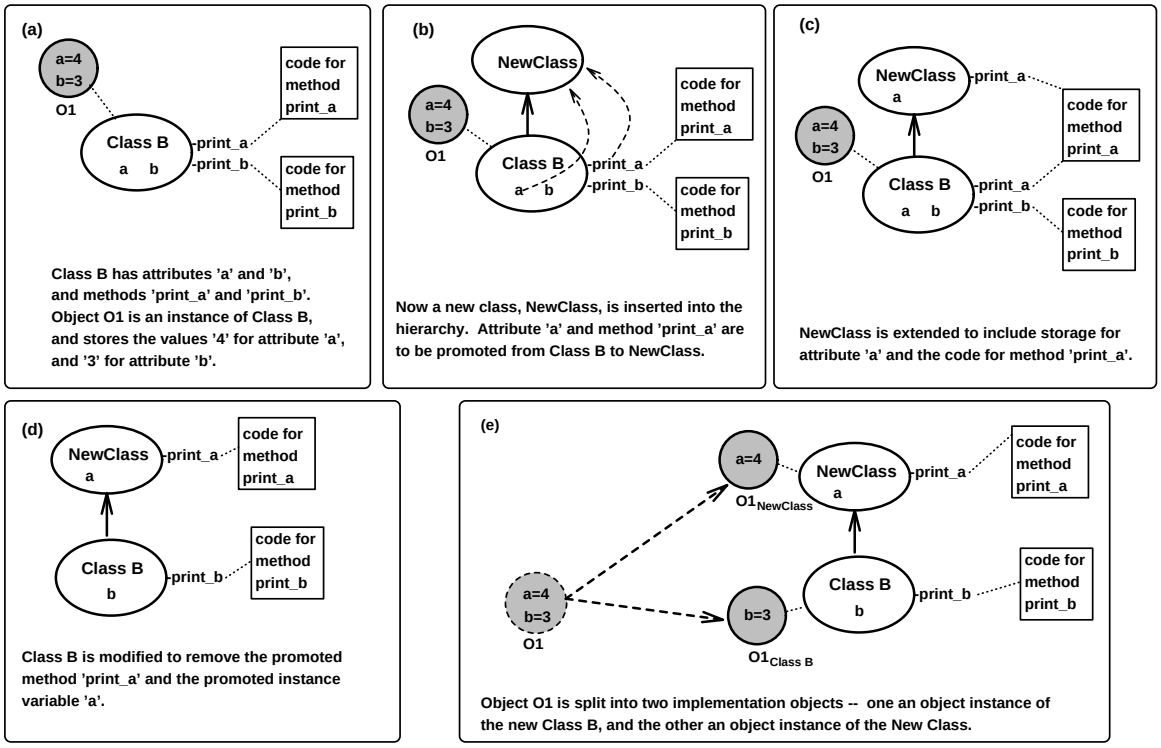

Figure 9: Attribute a and method print_a are moved from class B to a new superclass.

Figure 10 shows an example of attribute and method migration. Figure 10(a) shows a class *Class B* that has two attributes `a` and `b` and two corresponding methods `print_a` and `print_b`. Figure 10(b) shows a new class, *NewClass*, inserted into the hierarchy in such a way that attribute `a` and method `print_a` should be promoted from *Class B* to *NewClass*. Figure 10(c) shows the first task of the migration process, where *NewClass* is extended to include code for the to-be-migrated method `print_a` and storage for the migrating attribute `a`. Figure 10(d) shows the modification of *Class B* so as to remove the `print_a` method and storage for the migrating attribute `a`. Finally, Figure 10(e) shows how an implementation object (`O1`), originally belonging to *Class B* is split into an implementation object of *Class B* and an implementation object of *NewClass*. The system will make the appropriate links between the new implementation objects and the conceptual object for `O1`.

[14] We use the `become:` method to give the implementation object of the modified existing class the same object identifier as the original implementation object.

The existence of a single point of inheritance combined with the object-slicing approach guarantees that no more than two classes (the original class where the properties currently reside, and the new class to which they are being moved) will ever be involved in such a migration. The proof for this is presented elsewhere [16].

4.3 Construction of view schemata

MultiView uses the augmented global schema graph for the selection of both base and virtual classes and for arranging these classes in a consistent class hierarchy, called a *view schema*. View schemata represent the virtual restructuring of the *is-a* hierarchy, allowing users to hide and/or highlight classes. At any time, the user can specify view schemata from the global schema by adding classes to (and removing classes from) a view schema.

We do not support the further modification of a virtual class specification due to its inclusion in a view schema; rather a virtual class will look the same and exhibit the same behavior in any view schemata in which it is included. This feature of *MultiView* is significantly different from other approaches. For instance, in [20], the specification of a virtual class (both type and extent) must be dynamically recomputed for each view schema it is inserted in, since, for example, the addition of an is-a relationship may add new inherited attributes to the virtual type. In *MultiView* a view schema is instead defined simply by collecting all virtual classes that are to be made available to a particular user into one schema.

While this selection of classes for a particular view is done explicitly by the user, the generation of view generalization relationships among the set of selected classes of a view schema is automated in the current version of our system. Automatic view generation simplifies the view specification process for the users by automating tedious tasks, and guarantees the consistency of the view schema.

Because the global schema maintains the relationships between all base and virtual classes, we can generate the view schema relationships in polynomial time. The process of constructing a view schema from a set of classes can be divided into three tasks [27]: First, let $V = c_1, c_2, \ldots, c_n$ be the subset of the classes in the global hierarchy chosen to participate in the view schema. For each pair of classes $c_i, c_j (i \neq j)$ we define the value, $subsumed(c_i, c_j)$, calculated as follows:

$$subsumed(c_i, c_j) = \begin{cases} 1 & \text{if } c_i \text{ is-a } c_j \\ 0 & \text{otherwise} \end{cases}$$

The $subsumed(c_i, c_j)$ values for each set of classes can be directly inferred from the global hierarchy. The result of this is an nxn matrix called *edge-matrix* (c_i, c_j), which contains a 1 for every possible edge in the view schema. The second task is to remove redundant edges from the $edge - matrix$. This can be done by first multiplying the edge-matrix by itself, then subtracting it from itself. The resulting edge-matrix contains only non-redundant edges. Once the correct edge-matrix has been calculated, the final task is to set the edge relations within the view schema. This can be done by walking through the non-redundant edge-matrix and generating a subsumption relationship between pairs of classes in the view schema (using the class variables of the view schema class) wherever a 1 value exists.

5 Discussion

5.1 Status of implementation

The object-slicing data model described in this paper has been implemented using the GemStone OODB, version 3.2. Our prototype provides a set of about 40 meta-classes that use object-slicing to extend GemStone's Opal model to include multiple inheritance, multiple type instantiation, multiple class membership, and dynamic type changes. This layer of meta-classes is self-contained and thus could easily be retargeted to any other Smalltalk-based OODB system (or even persistent store) to provide a powerful view-based object model. Members of our team are currently using this extended architecture as a foundation for quickly constructing advanced OODB tools, such as object-oriented view management systems [13] and the Transparent Schema Evolution manager for OODBs [24].

5.2 View updates in object-slicing model

In relational systems, updates on views (virtual tables) are usually translated into updates onto the appropriate base relations. Unfortunately, many view queries cannot be unambiguously translated into base queries, and thus relational views are often not updatable. Two reasons why object-oriented systems more naturally permit updatable views are that (1) objects have unique, system-generated object identifiers, and (2) class-specific methods are associated with each object [26]. An object-oriented query operates upon objects rather than values, so all updates are specified in terms of these object-identifiers. This is one reason why although in relational systems most views are not updatable, in object-oriented systems, a large class of views are updatable (subject to the intentions of the user).

Our current MultiView implementation provides fully updatable views, since we utilize an object-preserving algebra for class derivation [26, 28]. Because attributes for both base and virtual classes are stored in a single place in our object-slicing model, update methods applied to instances of virtual classes automatically propagate to the classes where the attributes are stored. When a conceptual object is modified, the access method is propagated (via the inheritance hierarchy) to the appropriate implementation object holding the value of the affected instance variable. Note that the modification of an attribute could also affect the virtual classes derived from any class that inherits from or is based on the class where the modification took place. Since as an intrinsic property of our model, all classes that inherit an instance variable will inherit it from a *single source*, we would be safe in propagating notification of the change downwards from the modified class. We avoid unnecessary propagation of view updates by associating with each class a list of the selection classes whose predicates involve attributes belonging to that class. In this way modifications are passed efficiently to only those affected classes. A more detailed treatment of view materialization issues is beyond the scope of this paper.

5.3 Performance studies using the OO7 benchmark

To justify our object-slicing representation paradigm, we have run several test queries from the OO7 benchmark with the goal of comparing GemStone's native implementation versus our *MultiView* object model implementation [6]. GemStone is a Smalltalk-based system while the four systems compared in the OO7 benchmark paper [6, 7] are all C++ based. GemStone thus supports dynamic method resolution, run-time augmentation of the schema with new methods, etc. For these reasons, we did not compare GemStone against other systems - but limited our study to comparing "pure" GemStone with *MultiView*. For this study, we used GemStone, version 3.2 Opal; and created a randomly populated database of the parts-assembly benchmark example with 10,000 Atomic Parts.

First, we compare results for navigation-type queries, e.g., for the "Traversal 1" query, which required more than ten minutes on the average to run. The "Traversal 1" query tests raw pointer traversal speed with a high degree of locality [6]. The query requires a traversal of the assembly hierarchy (shown in Figure 11) and performs a depth-first search on each part's graph of atomic parts. For this type of queries, *MultiView* slightly improved upon GemStone's time (by $\approx 4\%$). We achieved this improved performance in spite of having built *MultiView* on top of GemStone rather than directly into the GemStone kernel – and thus having an extra layer of indirection. The improved performance can be explained as follows. First, the navigation was limited to access of local instance variables (rather than inherited ones), thus there was no overhead of finding appropriate implementation objects for *MultiView*. Consequently, the search is limited to one implementation object per atomic part - rather than possibly several implementation objects. Most importantly, these navigated implementation objects are much smaller in size (containing only local instance variables) compared to GemStone's native objects (containing both local and inherited instance variables in one contiguous allocation).

Next, we ran exact match lookup queries, in particular, "Query 1 Search" that does random lookup of an atomic part based on the inherited property "id" as shown in Figure 11. For this lookup, *MultiView* performed 3-1/2 times slower than GemStone. Given that our first *MultiView* prototype is not optimized for performance, this result was to be expected. The major overhead stems from the fact that for each of the 10,000 lookups, we would have to traverse from each Atomic Part's implementation objects to its corresponding DesignObj's implementation object to retrieve the "id" data value. Also, since we do not have full access to GemStone's source code, the *MultiView* prototype is layered on top of GemStone, causing additional overhead.

Accessing an inherited attribute in *MultiView* requires two additional traversals than in GemStone: One to traverse the `conceptualLink` to get from the implementation object to its conceptual object, and the other to traverse the `implementationLink` to get from the conceptual object to the correct implementation object holding the desired data value. As demonstrated in this paper, these two links improve the flexibility of our object model greatly but they also cause the observed performance degradation. However, we anticipate that this can be alleviated by clustering the conceptual object and all implementation objects of the same entity into the same page. This technique would result in the same number of pages faults as in the conventional

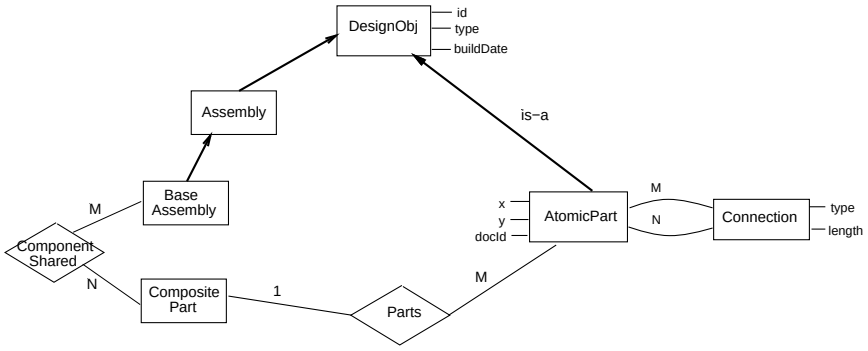

Figure 10: A Portion of OO7 Benchmark Database.

system, and all additional traversals would be in-memory operations. It is well known that increased in-memory operations do not degrade the performance significantly because they are very fast compared to I/O operations. In the future, we may also be able to improve upon this cost by maintaining partial caches of methods with classes in order to speed method lookups.

6 Related Work

In recent years, several proposals for object-oriented view systems have appeared in the literature [1, 4, 12, 21, 28, 31]. Below, we first compare other approaches towards view management with the *MultiView* approach, and then we compare the object-slicing approach used in our implementation with similar approaches.

MultiView differs from other view systems in that it does not simply adopt assumptions made by current OODB architectures, but rather we re-examined key features required as foundation for views. For example, we overcame the problem of each object belonging to one and only one most-specific type – which is an unreasonable assumption for view systems. Rather than use contiguous storage for objects, a *MultiView* object is distributed among multiple object-slicing implementation objects. The Iris functional database system resembles our implementation in that, being built on top of a relational engine, it distributes data over several relational tables [10]. Iris does not support view mechanisms, and does not address issues of classification, inheritance for virtual classes, etc.

Most of the current proposals for view management systems have not yet been implemented. Furthermore, none of those which have been implemented support all of the features we identify as desirable for view systems (Section 2.2). O2 Views [21] [8], based on Abiteboul and Bonner [1], is the first and only commercial implementation of an object-oriented view management system, currently realized. The O2 Views approach does include the integration of view classes into a view schema, but rather than supporting a global class hierarchy, it daisy-chains views to enable selective upward versus downwards inheritance (instead of creating intermediate classes and propagating methods). It thus has limited update capabilities. O2 Views does not provide type closure in views, does not support union classes (because of its lack of a type inference mechanism), and does not support capacity-augmenting views.

Scholl et al's work on views comes closest to our work [28, 29]. They suggest use of an object-preserving subset of their algebra to define virtual classes and thus achieve updatable views. However, they do not address the classification of view classes into a global schema, the automatic generation of complete view schemata, nor the implementation of capacity-augmenting views.

The object-slicing implementation underlying the current implementation of *MultiView* can also be compared to mechanisms used in *role modeling* approaches [11, 22]. In *role modeling* systems, objects dynamically gain and lose multiple interfaces (aka *roles*) throughout their lifetimes. These roles can be compared to the implementation objects of an object-slicing implementation, in that both permit objects to belong to multiple classes and change types dynamically. In some sense, accessing an object through one of its implementation objects is like accessing an object while it is playing one of its roles. However, role systems and views systems have different goals. Role systems strive to increase the flexibility of objects by enabling them to dynamically change types and class membership. View systems, on the other hand, enable users to restructure the types and class membership of classes - based on content-based queries.

Unlike many role systems, which allow object hierarchies to exist independently from class hierarchies [30], objects in our implementation always conform to the existing global class hierarchy in that if an object possesses an implementation object of a given class's type, it must also possess an implementation object for every class that is a superclass of that given type. This achieves an efficient and uniform inheritance scheme. Also, unlike many role systems, in our implementation conceptual objects can be associated with at most one implementation object of a given type [11]. Role systems do not deal with the issues of virtual class derivation, classification, nor with method promotion.

Finally, the role system discussed in [11] was implemented using techniques similar to object-slicing. This system, like ours, is implemented in Smalltalk by overriding the `doesNotUnderstand:` method. The difference between [11] and our implementation is that [11] is a role system while our implementation is a view system. For example, unlike [11], we do not permit entities to occur several times in the same type of role. Also, the [11] system does not permit the derivation of new virtual classes, thus not addressing any of the issues related to view management.

7 Conclusions and Future Work

In this paper, we have re-examined the representation assumptions underlying most OODB systems from the perspective of achieving a flexible and powerful view system. We have identified several key features that are not provided by current OODB systems. In particular, we found that none of the available OODBs provide the features required of an object model for capacity-augmenting views, such as multiple classification and dynamic restructuring of object representations. We propose a novel object-oriented modeling approach based on the object-slicing paradigm which addresses these limitations. We then describe a design of the object model, addressing issues of inheritance, type changes, classification, and capacity-augmentation. Using this paradigm to support a flexible object model implementation offering efficient re-structuring, re-classification, *multiple classification*, updatability, and *capacity-augmenting views*, we have successfully completed a prototype implementation using the commercial GemStone OODB system.

We have evaluated our system using the OO7 benchmark in order to determine the overhead associated with our approach. In the course of this evaluation, we identified classes of queries in which the *MultiView* overhead degraded GemStone's performance, and other classes of queries in which *MultiView* actually improved upon GemStone's performance. In the future, we plan to reduce the associated overhead, improve the update strategies, and continue to employ the system as a testbed for research into topics such as transparent schema evolution and flexible tool integration.

References

[1] S. Abiteboul and A. Bonner. Objects and views. *SIGMOD*, pages 238–247, 1991.

[2] T. Atwood, R. Cattell, J. Duhl, G. Ferran, and D. Wade. The odmg object model. *Journal of Object Oriented Programming*, pages 64–69, June 1993.

[3] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD*, pages 311–322, 1987.

[4] E. Bertino. A view mechanism for object-oriented databases. In *3rd International Conference on Extending Database Technology*, pages 136–151, March 1992.

[5] P. Butterworth, A. Otis, and J. Stein. The gemstone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[6] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The oo7 benchmark. *SIGMOD*, 1993.

[7] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The oo7 benchmark. Technical report, University of Wisconsin-Madison, January 1994.

[8] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual schemas and bases. To appear in EDBT '94.

[9] O. Deux et al. The story of o2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[10] D.H. Fishman. Iris: An object oriented database management system. In *ACM Transactions on Office Information Systems*, volume 5, pages 48–69, January 1987.

[11] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. manuscript, 1993.

[12] S. Heiler and S. B. Zdonik. Object views: Extending the vision. In *IEEE International Conference on Data Engineering*, pages 86–93, 1990.

[13] H. J. Kim. *Issues in Object Oriented Database Systems.* PhD thesis, University of Texas at Austin, May 1988.

[14] W. Kim. A model of queries in object-oriented databases. In *Proceedings of the International Conference on Very Large Databases*, pages 423–432, August 1989.

[15] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the orion next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[16] H. A. Kuno. View management issues in object-oriented databases. Dissertation Proposal, 1994.

[17] H. A. Kuno and E. A. Rundensteiner. Implementation experience with building an object-oriented view management system. Technical Report CSE-TR-191-93, University of Michigan, Ann Arbor, August 1993.

[18] S. Marche. Measuring the Stability of Data Models. *European Journal of Information Systems*, 2(1):37–47, 1993.

[19] J. Martin and J. Odell. *Object-Oriented Analysis and Design.* Prentice-Hall, Inc., 1992.

[20] M. A. Morsi, S. B. Navath, and H. J. Kim. A schema management and prototyping interface for an object-oriented database environment. In F. Van Assche, B. Moulin, and C. Rolland, editors, *Object Oriented Approach in Information Systems*, pages 157–180. Elsevier Science Publishers B. V. (North Holland), 1991.

[21] O2 Technology. *O2 Views User Manual*, version 1 edition, December 1993.

[22] M. P. Papazoglou. Roles: A methodology for representing multifaceted objects. In *International Conference on Database and Expert Systems Applications*, pages 7–12. Springer-Verlag, 1991.

[23] D. J. Penney and J. Stein. Class modification in the gemstone object-oriented dbms. In *OOPSLA*, pages 111–117, 1987.

[24] Y. G. Ra and E. A. Rundensteiner. A transparent object-oriented schema change approach using view schema evolution. Technical Report CSE-TR-211-94, University of Michigan, 1994.

[25] E. A. Rundensteiner. A class integration algorithm and its application for supporting consistent object views. Technical Report 92-50, University of California, Irvine, May 1992.

[26] E. A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *18th VLDB Conference*, 1992.

[27] E. A. Rundensteiner. Tools for view generation in oodbs. In *ACM 2nd Int. Conf on Information and Knowledge Management (CIKM)*, November 1993.

[28] M. H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proceedings of the Second DOOD Conference*, December 1991.

[29] M. H. Scholl and H. J. Schek. Survey of the cocoon project. *Objektbanken fur Experten*, October 1992.

[30] E. Sciore. Object specialization. *ACM Transactions on Information Systems*, pages 103–122, April 1989.

[31] J. Shilling and P. Sweeney. Three steps to views: Extending the object-oriented paradigm. In *OOPSLA*, pages 353 – 361, October 1989.

[32] K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema virtualization in object-oriented databases. *IEEE International Conference on Data Engineering*, February 1988.

[33] M. Tresch and M. H. Scholl. Schema Transformation without Database Reorganization. In *SIGMOD RECORD*, pages 21–27, 1993.

[34] R. Zicari. A Framework for O_2 Schema Updates. In *7th IEEE International Conf. on Data Engineering*, pages 146–182, April 1991.