Tools for View Generation in Object-Oriented Databases

Elke A. Rundensteiner*
Dept. of Electrical Eng. and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
rundenst@eecs.umich.edu, phone: (313) 936-2971

Abstract

This paper discusses two aspects of the object-oriented view management system, MultiView, which we designed to simplify view specification. First, we introduce a query language for view customization that operates on a complete schema, rather than deriving only individual virtual classes. This graph algebra promises to reduce the effort involved in view specification by lowering the number of queries necessary to define a view. Second, we introduce a tool that guarantees the composition of view classes into arbitrarily complex, yet consistent, view schemata. Unlike for relational views, generalization relationships in OO views must be validated so that they are consistent with the global schema. In our system, we solve this problem by reformulating view schema construction as a classical graph theory problem, called minimal covering. This allows us to develop efficient algorithms that automatically generate a complete, minimal and consistent view schema. Proofs of correctness of these algorithms can be shown.

1 INTRODUCTION

The goal of our project is to develop an environment for the design, specification and management of views for object-oriented databases (OODBs) by non-database specialists. OO views promise to be more powerful than relational views, since they can customize not only data structures but also the associated operations. Furthermore, they can be used to restructure the generalization hierarchy to classify objects in a manner most meaningful for a particular user group. Lastly, the view update problem can be successfully addressed due to the concepts of object identity and encapsulation [6, 17, 9]. The construction of these *virtual, possibly restructured, subschema graphs* of the global schema raises a number of challenging research issues in terms of how to restructure view schema graphs and how to maintain them consistent with the global schema.

We have developed and implemented a view support system, *MultiView*, that successfully addresses these issues [14, 10]. *MultiView* breaks view specification into two tasks: (a) the derivation of virtual classes via an object-oriented query and their integration into *one* consistent global schema and (b) the definition of view schemata composed of both base and virtual classes on top of this augmented global schema. A key feature of this approach is that view classes (both their membership extent and their type description) are completely defined by view derivation (task a) rather than being further modified by view schema construction (task b). In this paper, we describe two components of this environment designed to simplify view specification: (1) a graph-based query language that minimizes the number and size of queries needed for view specification (task a) and algorithms for the automatic generation of consistent view schemata (task b).

When using a simple object algebra for defining views for interfacing CAD tools with a CAD database, we found that the specification of most views was repetitious requiring many similar types of queries [16]. In this paper, we therefore introduce a new query language that simplifies view specification by grouping similar types of simple queries into queries on schema graphs. The graph algebra takes as input a schema and generates as output another schema, whereas conventional object algebra has one class as input and another class as output [17]. Graph algebra operators can be arbitrarily combined with the "regular" object algebra for view generation. This graph algebra promises to lower the number of queries necessary for defining a view. For example, if we want to remove the modify-salary method from the Employee class and from its subclasses Professor, Staff, and TeachingAssistant in a particular user view, then we use the following graph algebra statement: "hide* modify-salary from Employee*." With conventional al-

*The author is grateful for support from the University of Michigan Faculty Award Program, and from NSF (IRI-9309076).

CIKM '93 - 11/93/D.C., USA
© 1993 ACM 0-89791-626-3/93/0011 ....$1.50

gebra, we would have to specify a separate query for each class in the subgraph rooted at the Employee class, i.e., "hide modify-salary from Employee;" "hide modify-salary from Professor;" "hide modify-salary from Staff;" and "hide modify-salary from TeachingAssistant."

Unlike for relational views, generalization relationships in OO views must be validated so that they are consistent with the global schema. Inserting arbitrary *is-a* relationships between view classes may result in an inconsistency. For instance, it is incorrect for the view definer to assert an *is-a* arc between two classes in the view that are not *is-a* related in the global schema. Our characterization of *is-a validity* of a view schema in terms of the *completeness*, *minimality* and *consistency* of its view generalization hierarchy allows for the identification of different types of such inconsistencies. Rather than requiring manual entry of view *is-a* arcs by the view definer and then checking the entered information for *validity*, we propose to automate the generation of the view generalization hierarchy.

For this purpose, we reformulate view schema construction as a classical graph theory problem, called minimal covering. This allows us to apply well-known graph-theoretic algorithms, e.g., transitive reduction, to solve the problem. In this paper, we describe efficient algorithms for automatically generating consistent views, for both single and multiple inheritance schema graphs.

Sections 2 and 3 present background material and describe *MultiView*. The graph algebra is described in Section 4, and Section 5 presents the algorithms for automatic view generation. We compare *MultiView* to related work in Section 6, and conclude with Section 7.

2 OBJECT-ORIENTED VIEWS

2.1 The Object Data Model

Our view system is based on a typical object model like for instance COCOON [17], Morsi et al.'s model [12], and IRIS [7]. Below we review terminology required for the remainder of the paper. A **class** C_i has a unique class name, a type description and a set membership. The type of a class C_i, **type**(C_i), consists of a number of property functions, **properties**(C_i). A property function p $\in$ P could be a value from a simple enumeration type, an object instance from some class, an arbitrarily complex function, or an object method. Each p $\in$ P has a name and signature. For each type t, $properties_t$ denotes the set of property functions of t and $domain_p(t)$ denotes the domain of p in t. A **class** is also a container for a set of object instances, denoted by **extent**(C_i) [13]. Given two classes *C1* and *C2*. C1 is called a **subset** of C2, denoted by C1 $\subseteq$ C2, if and only if ($\forall$ o $\in$ O) ((o$\in$C1) $\Longrightarrow$ (o$\in$C2)). C1 is called a **subtype** of C2, denoted by C1 $\preceq$ C2, if and only if (**properties**(C1) $\supseteq$ **properties**(C2)) and ($\forall$ p $\in$ **properties**($C2$)) (**domain**$_p$($type(C1)$) $\subseteq$ **domain**$_p$($type(C2)$)). C1 is called a **subclass** of C2, denoted by C1 *is-a* C2, if and only if (C1 $\preceq$ C2) and (C1 $\subseteq$ C2).

Definition 1. *An **object schema** is a rooted directed acyclic graph S=(V,E), where V is a finite set of vertices and E is a finite set of directed edges. Each element in V corresponds to a class C_i, while E corresponds to a binary relation on V $\times$ V. Each directed edge e from C_1 to C_2, denoted by e = <C_1, C_2>, represents the direct is-a relationship (C_1 is-a^d C_n). There is one designated root node, called **Object**, which contains all object instances of the database and its type description is empty*[1].

2.2 Object-Oriented Views

We distinguish between **base** and **virtual** classes. **Base classes** are defined during initial schema definition, with their object instances stored as base objects. **Virtual classes** are defined during the lifetime of the database using object-oriented queries. A virtual class has an associated membership derivation function that determines its exact membership based on the state of the database. The extent of a virtual class is not explicitly stored, but rather computed upon demand.

Definition 2. .

1. *The **base schema** (BS) is an object schema S=(V,E), with all nodes in V being base classes.*
2. *The **global schema** (GS) is an extension of BS augmented by all virtual classes defined during the lifetime of the database.*
3. *Given a global schema GS=(V,E), then a **view schema** (VS), or short, a **view**, is a schema VS= (VV,VE) with: (a) VS has a unique view identifier denoted by < VS >, (b) VV $\subseteq$ V, and (c) VE $\subseteq$ transitive-closure(E).*

View classes (including their extent and type description) are completely defined by class derivation, and this definition is not modified by organizing them into a view schema. In other systems [12], the same class often exhibits different behavior in different views.

2.3 The Validity of the View Generalization Hierarchy

Next, we introduce criteria that indicate whether the class generalization hierarchy of a view schema is consistent with the one of the underlying global schema.

[1]The root class provides a unique entry point into the database, as done in GemStone.

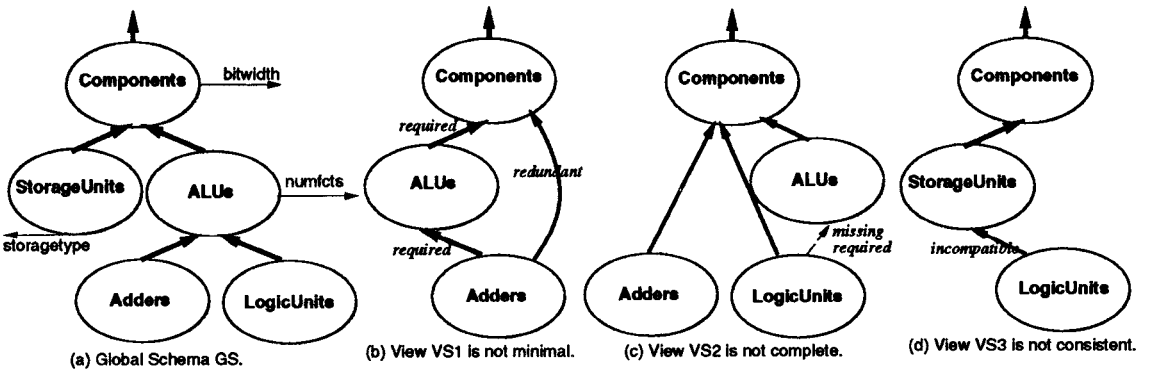

Figure 1: View Validity Example: Minimality, Completeness and Consistency Criteria.

Definition 3. *For all classes C_1,C_2 in VV, an is-a arc from C_1 to C_2 is defined to be:*

- ***required** in VS, if (C_1 is-a C_2) in GS and there is no C_x in VV such that (C_1 is-a C_x) and (C_x is-a C_2) in GS. VS is **complete**, if VE contains all required view is-a relationships in VS.*
- ***redundant** in VS, if there is a class C_x in VV such that (C_1 is-a C_x) in GS and (C_x is-a C_2) in GS. VS is **minimal**, if none of the view is-a arcs in VE is redundant in VS.*
- ***incompatible** in VS if the edge <C_1,C_2> is in VE and not(C_1 is-a C_2) in GS. The view VS is **consistent**, if none of its view is-a arcs in the set VE is incompatible.*

Definition 4. *A view VS=(VV,VE) for GS=(V,E) is **is-a valid (or valid)** if the set of all view is-a relationships VE among its view classes VV is complete and minimal and consistent*[2].

Example 1. *Figures 1.b, 1.c, and 1.d depict views defined on the GS schema shown in Figure 1.a. In Figure 1.b, the edge e = <Adders,Components> can be removed from the view VS1 without losing the information that (Adders is-a Components). Therefore VS1 violates the minimality criterion. The view VS2 in Figure 1.c violates the completeness criterion, since the required edge e=<LogicUnits,ALUs> is missing. In Figure 1.d, the edge e = < LogicUnits,StorageUnits > is incompatible in VS, since the relationship (LogicUnitsis − aStorageUnits) does not hold in GS. VS3 is not is-a valid.* □

3 THE *MultiView* METHODOLOGY

We provide a short overview of *MultiView* here to set the context for the remainder of the paper [14]. *MultiView* breaks view specification into two tasks:

a. the derivation of virtual classes and their integration into *one* consistent global schema graph and
b. the specification of arbitrarily complex view schemata composed of both base and virtual classes on top of the augmented global schema.

The first task of *MultiView* supports the virtual customization of existing classes by deriving new classes with a modified type description and/or membership content, which effectively controls the visibility of data and the access privileges to property functions. The current prototype of *MultiView* utilizes an object algebra for this purpose [14][3]. In this paper, we present a new language for class derivation, called graph algebra, that simplifies view specification by grouping similar types of simple queries into queries on schema graphs.

MultiView integrates all virtual classes into one global schema [15]. Global schema integration ensures the explicit capture of all class relationships between stored and derived classes in terms of type inheritance and subset relationships (rather than only between base classes as typically done in OODBs). This is useful for sharing (inheriting) property functions and object instances consistently among classes without unnecessary duplication. It also assures the consistency of all views with the global schema and with one another. Last but not least, it is vital for the consistent derivation of semantically correct view schema composed of both base and virtual classes (i.e., for the second task).

The second task of *MultiView* utilizes the augmented global schema for selection of both base and virtual classes and for arranging these view classes in a consistent *view schema* hierarchy. This supports for instance the virtual restructuring of the *is-a* hierarchy by allowing to hide from and to expose classes within a view schema. Since during the first task of *MultiView* the semantics of a virtual class (including its type description

[2] While the consistency criterion is necessary to assure the correctness of the view schema, the completeness and minimality criteria are useful but not always mandatory for some situations.

[3] Because we restrict the query language used for virtual class derivation to be an object-preserving algebra, views in MultiView are updatable [14, 17].

and membership content) are determined, the specification of a view schema consists simply of picking out the classes and/or subschema graphs that are of interest for the particular application. We do not support further modification of this 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 of the view schemata in which it is included. This feature of *MultiView* is a significant difference to other approaches [12]. Rather than requiring the manual insertion of *view is-a* relationships by the view definer, we have developed algorithms that automatically augment the set of selected view classes by these relationships to generate a *valid* view schema. Algorithms and examples can be found in Section 5.

4 Using Graph Algebra for Class Customization

4.1 Graph Algebra

While the query languages used by other view systems generally derive individual view classes [1, 6, 17]. we propose a query language for deriving complete view schemata. The development of this graph algebra was driven by our observations obtained from using ordinary object algebra for defining views for interfacing CAD tools with a design database [16]. Namely, we found that the specification of most views was repetitious requiring many similar types of queries. We hence have developed a solution to this problem by extending the well-known object algebra which operates on individual classes to operate on complete schema graphs. Our definition of this graph algebra is based on the 'ordinary' object algebra we presented in [14], with some example operators depicted in Figure 2.

In the following, we demonstrate that these operators can be built by composing primitive operators on individual classes into meaningful operators on schemata. These graph operators serve two purposes. First, they simplify the creation of a virtual schema by grouping the primitive class operators into more complex, yet well-defined, graph operators. Second, they simplify the integration of the set of virtual classes into the global schema by explicitly stating (and thus preserving) the is-a relationship between the set of virtual classes that make up the resulting virtual subschema.

For this section we assume the following notations. The term <class>* with <class> the name of a class of a schema S refers to the complete subschema of S that is rooted at <class>. The system will generate a unique class name for each newly generated class in <*virtual-class*>* (e.g., append the 'prime' symbol to the names of classes), but explicit renaming of these classes by the view definer can and should take place.

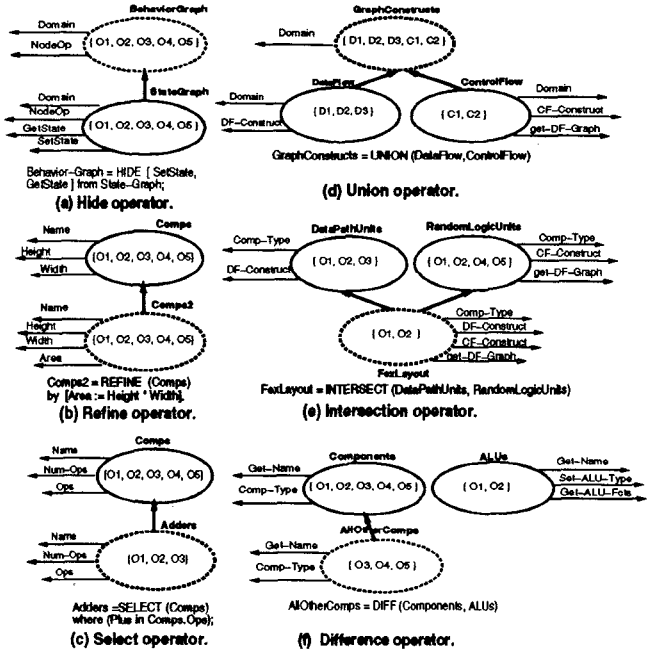

Figure 2: Examples of Class Derivation.

4.2 The Hide* Graph Operator

The **hide*** operator, an extension of the **hide** operator (Figure 2.a), takes on a complete schema graph as input and returns a virtual schema graph as result. It removes one or more functions from all classes in the source schema, while preserving all other attributes visible in the source schema[4]. It has the following syntax:

<*virtual*-class>* := **hide*** [<prop-functions>] **from** (<*source*-class>*).

The semantics of the **hide*** operator are: to derive a virtual class Ci' for all classes Ci in the input schema <*source*-class>* by the query Ci' := **hide** [<prop-functions>] **from** (Ci). We can deduce the following is-a relationships:

(∀ Ci, Cj in <*source*-class>*) (∀ Ci', Cj' in <*virtual*-class>*) (Ci *is-a* Cj ⇔ Ci' *is-a* Cj').

(∀ Ci in <*source*-class>*) (∀ Ci' in <*virtual*-class>*) (Ci *is-a* Ci').

Example 2. *In a floorplan view, the structure of the design may not be permitted to be modified (via the*

[4] It now becomes apparent why we prefer the **hide** over the **project** operator. Both accomplish the same effect when applied to an individual class, namely, for the **hide** operator we specify which property functions to remove (hide) while for the **project** operator we specify which property functions to keep. For a complete schema, **project*** projects out a *fixed* set of attributes for all its result classes; and thus all resulting classes will have the same projected result type. The **hide*** operator hides the same *fixed* set of attributes from all classes; and thus all resulting classes will potentially have different types - namely their original functions minus the hidden functions.

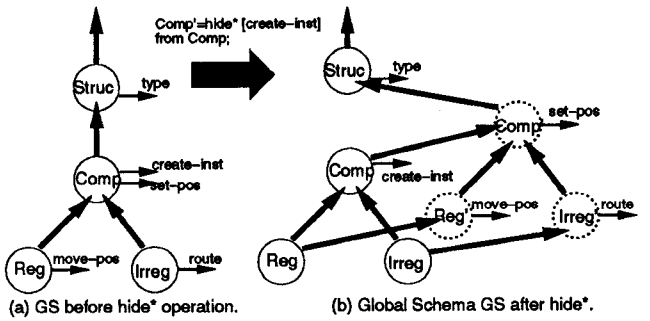

Figure 3: The **hide*** Graph Operator.

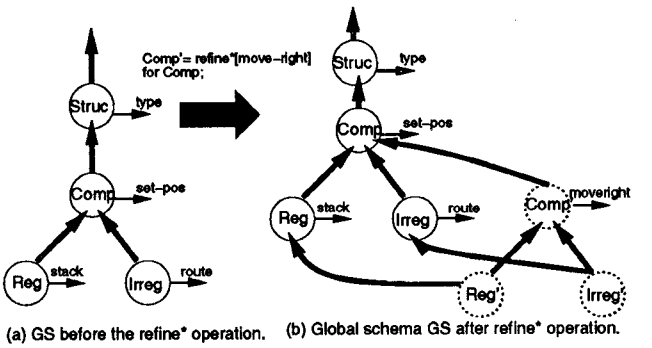

Figure 4: The **refine*** Graph Operator.

create-instance or delete-instance functions). However, the geometric placement of components can be modified, e.g., by changing its position or resizing. We thus could use the hide operator to remove the 'forbidden' functions from the schema while leaving all other operators intact. In Figure 3, the query "Comp'* := hide* [create-instance] from (Comp*)" removes these methods from all classes in the Comp* subschema. This creates a virtual subschema Comp'* composed of three classes, Comp, Reg and Irreg, which are identical to their source classes except for the removal of the forbidden function create-instance.*

4.3 The Refine* Graph Operator

The **refine*** operator adds one or more new derived property functions to all classes in a subschema, while preserving all visible ones. Its syntax is:

<*virtual*-class>* := **refine*** [<prop-function-defs>] **for** (<*source*-class>*).

with <prop-function-def> the definition of one or more new property functions. Its semantics are to refine the types of all source classes by adding the property functions listed in <prop-function-defs> to their type definitions. This can be achieved by deriving a virtual class Ci' for all Ci in the source schema <*source*-class>* by the query Ci' := **refine** [<prop-function-defs>] **for** (Ci).

(∀ Ci, Cj in <*source*-class>*) (∀ Ci', Cj' in <*virtual*-class>*) (Ci *is-a* Cj ⇔ Ci' *is-a* Cj').

(∀ Ci in <*source*-class>*) (∀ Ci' in <*virtual*-class>*) (Ci' *is-a* Ci).

Example 3. *In Figure 4, the query "Comp'* := refine* [movepos() = {fct − body}] for (Comp*)" extends the component classes with a method to modify the floorplan position.*

4.4 The Select* Graph Operator

The **select*** operator has the following syntax:

<*virtual*-class>* := **select* from** (<*source*-class>*) **where** (<predicate>),

with <*predicate*> some possibly complex predicate function on the source class and its type description. Its semantics are to select a subset of objects from all classes in the given source subschema based on the given predicate. For all classes Ci in the source schema <*source*-class>*, derive a virtual class Ci' by the query Ci' := **select* from** (Ci) **where** (<predicate>). We can deduce the following:

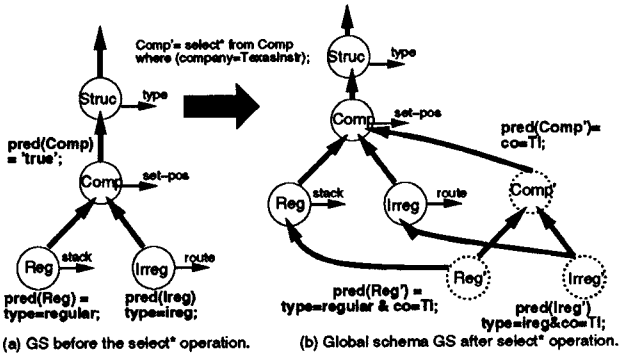

Figure 5: The **selection*** Graph Operator.

(∀ Ci, Cj in <*source*-class>*) (∀ Ci', Cj' in <*virtual*-class>*) (Ci *is-a* Cj ⇔ Ci' *is-a* Cj').

(∀ Ci in <*source*-class>*) (∀ Ci' in <*virtual*-class>*) (Ci' *is-a* Ci).

Example 4. *In Figure 5, the query Comp'* := select* from (Component*) where (company=Texas Instruments) is used to derive a virtual view schema Comp'*. The select* operator effectively removes all components manufactured by Texas Instruments from all classes in Comp*.*

4.5 The Union* Graph Operator

The extended set operators have two input arguments, a schema and a class. The **union*** operator

adds additional object instances to all classes in the input schema, with the additional object instances gotten from its second argument:

<*virtual*-class>* := **union***(<*source*-class1>*, <*source*-classx>).

The **union*** operator creates the virtual schema <*virtual*-class>* that is composed of Ci' defined as follows: for all Ci in <*source*-class>*, derive a class Ci' by the query Ci' := **union**(Ci,<*source*-classx>). The following *is-a* relationships hold:

(∀ Ci, Cj in <*source*-class1>*) (∀ Ci', Cj' in <*virtual*-class>*) (Ci *is-a* Cj ⇔ Ci' *is-a* Cj').

(∀ Ci in <*source*-class1 >*) (∀ Ci' in <*virtual*-class>*) (Ci *is-a* Ci' and (<*source*-classx> *is-a* Ci').

Since <*source*-classx> is a subclass of all virtual classes Ci', it is sufficient to only maintain the *is-a* relationships between <*source*-classx> and the leaf classes Ci' of <*source*-class1>*. All others are derivable by transitive closure.

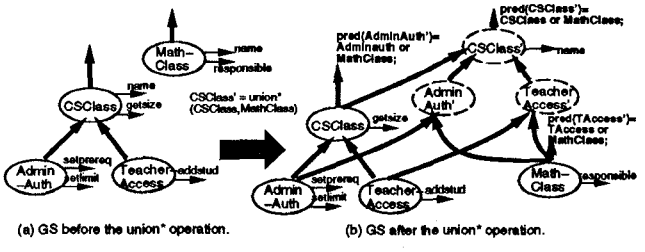

Figure 6: The **union*** Graph Operator.

Example 5. *Assume that the Mathematics and Computer Science departments are to be merged, and hence their courses should be combined for administrative purposes. In Figure 6, the query CSClass'* := **union***(CSClass*,MathClass) is used to add the Math-Courses to all classes describing the CS-Courses. Methods that have been defined for both types of courses are now applicable to these combined courses, i.e., the name method is applicable to CSClasses', AdminAuth' and TeacherAccess'. The other methods, such as addstudent or responsible-instructor, are only applicable to their original source classes.*

4.6 The Intersection* Graph Operator

The **intersect*** operator removes all object instances that are members of both source classes from all classes in a subschema. Its syntax is:

<*virtual*-class>* := **intersect***(<*source*-class1>*, <*source*-classx>).

The **intersect*** operator creates the virtual schema <*virtual*-class>* that is composed of Ci' defined as follows: for all Ci in <*source*-class1>*, derive a Ci' := **intersect**(Ci,<*source*-classx>).

(∀ Ci, Cj in <*source*-class1>*) (∀ Ci', Cj' in <*virtual*-class>*) (Ci *is-a* Cj ⇔ Ci' *is-a* Cj').

(∀Ci in <*source*-class1>*) (∀Ci' in <*virtual*-class>*) (Ci' *is-a* Ci and Ci' *is-a* (<*source*-classx>).

Since all Ci' are subclasses of the class <*source*-classx>, it is sufficient to only maintain the *is-a* relationship between the root <*source*-class1> of the schema <*source*-class>* and the class <*source*-classx>).

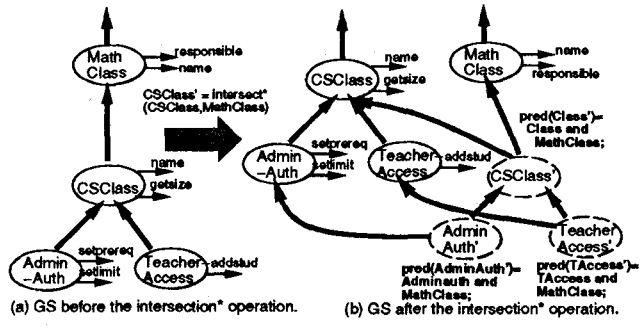

Figure 7: The **intersection*** Graph Operator.

Example 6. *We want to review courses taught by both the Mathematics and Computer Science departments, e.g., their prerequisites, etc. In Figure 6, we use the query CSClass'* := **intersect***(CSClass*,MathClass) to isolate these courses. Methods that have been defined for either of these two types of courses are now applicable to the collection of these combined courses.*

4.7 The Difference* Graph Operator

The **difference*** operator removes a subset of object instances from all classes in the input schema (its first argument) which are contained in the set of objects modeled by its second argument. This corresponds to the **select*** operator with the predicate of selection defined by the expression *(o ∈ C_i and not(o ∈ C_x))* with C_i, i for i=1,..., n, referring to all classes in the <*source*-class1>* schema.

4.8 Operator Composition

In this section, we have developed an algebra of graph-operators. We have demonstrated how this extended graph algebra can be defined in terms of a set of queries expressed in the well-known object algebra. It is straightforward to combine both algebras into one query, as long as we make the following assumptions: (1) the "regular" object algebra operators work on the root class of the schema when applied to a schema, and (2) the graph algebra operators treat an individual class as a trivial schema graph with one single class when applied to a class. Furthermore, we have shown how the

virtual schemata created by the graph operators are integrated with their source schemata.

5 ALGORITHMS FOR AUTOMATIC VIEW GENERATION

5.1 Motivation and Problem Definition

As explained in Section 3, we automate the specification of the view generalization hierarchy rather than requiring manual entry of the *view is-a* arcs by the view definer. Automatic view generation offers numerous advantages, some of which are detailed below:

1. It simplifies the view specification process for the users by automating tedious tasks.
2. It guarantees the consistency of the view schema (i.e., correctness of view query processing).
3. It prevents the introduction of redundant subclass relationships into the view (and thus supports a cleaner model of application domains).
4. It may reduce execution times for query processing on the view.
5. It assures the completeness of the view semantics by guaranteeing the presence of all required subclass relationships (provide maximal information).

In *MultiView*, all subclass relationships are calculated a priori for each pair of classes and the result of this evaluation is entered into the global schema. Hence, a valid view schema can be derived from a valid global schema (Definition 4) by exploiting the *syntactic* graph structure of the global schema rather than by requiring the *semantic* comparison of class specifications for each pair of classes. For a global schema classification algorithm see [15].

The view generation problem can thus be reformulated as a graph-theoretic problem. Let $GS = (V,E)$ be a global schema. Assume that a subset of classes $VV \subseteq V$ of GS has been selected (marked) to belong to $< VS >$ (see Section 3). We wish to develop an algorithm that automatically determines a set VE of *is-a* edges among classes in VV, such that $VS = (VV, VE)$ is a valid view schema (Definition 4). View generation has thus ben reduced to the classical graph problem called graph covering. In the next sections, we will discuss graph algorithms that automate the view creation for both single and multiple inheritance graphs [5].

[5] The algorithms are based on the view validity criterion (Definition 4). They enforce the resulting view to be *minimal*, *complete* and *consistent*. Sometimes, it may be desirable to relax this strict requirement such as to allow for redundancy or incompleteness (but of course not inconsistency) in the view. However, to avoid inconsistent class relationships, a view consistency checker would have to be developed for manually specified views.

5.2 View Schema Generation For Global Schemata with Single Inheritance

Assuming single inheritance in GS, the algorithm for the automatic view generation is based on a simple graph traversal of GS. The algorithm traverses GS in a breadth-first manner from the root down to the leaves. For each node C_i in GS that is marked by $<VS>$ it searches all branches in the subtree rooted at C_i. An *is-a* edge is inserted into VS between the parent C_i of a subtree and all subclasses of C_i that (1) are also marked by $<VS>$ and (2) are the closest to C_i in the tree. More formally, if $(C_1, C_2 \in VS)$ and $(C_1$ *is-a** $C_2)$ in GS and $(\nexists\ C_i$ *in* $VS)((C_1$ *is-a** $C_i)$ and $(C_i$ *is-a** $C_2))$, then the Edge-Creation algorithm inserts the edge $(C_1$ *is-a* $C_2)$ into VS. By Definition 4, this newly inserted edge is a *required* edge. The algorithm terminates when all marked classes C_i have been used as parent nodes once. See Figure 8 for the detailed algorithm.

```
Assumption: Single inheritance GS.
Input:  GS = (V,E) and View VS=(VV,VE)
   with VV ⊆ V marked by <VS>, VE=∅.
Output: Determine set of is-a edges VE on VV,
   such that VS = (VV,VE) is valid on GS.
Data Structures:
   PQueue, CQueue: queues for nodes of GS.
   Parent, Child: hold one class each.

Algorithm A1: Creation of Is-A Arcs for a View Schema.
algorithm Edge-Creation(GS,VS) is
   Add the root of GS onto PQueue[6].
   while (Parent := remove(PQueue)) do
      Add all children of Parent in GS on CQueue.
      while (Child := remove(CQueue)) do
         if Child is in VS then
         insert isa(Child,Parent) into edges(VS);
         Add Child onto PQueue;
         else
         Add children of Child in GS on CQueue;
end algorithm;
```

Figure 8: The Edge-Creation Algorithm.

Theorem 1. *(Correctness) The Edge-Creation algorithm generates a valid view schema VS=(VV,VE) assuming the global schema GS=(V,E) does not have multiple inheritance.*

Proof: See [15] for the proof. ■

Theorem 2. *(Complexity) The complexity of the Edge-Creation algorithm is linear in the number of nodes in GS, i.e., $O(|GS|)$, assuming the class hierarchy of GS is a tree.*

Proof: See [15] for the proof. ■

[6] We assume that each view schema includes the root object class of GS, i.e., it's a DAG.

5.3 View Generation For Global Schemata with Multiple Inheritance

For a schema with multiple inheritance (a DAG) the Edge-Creation algorithm does no longer guarantee the creation of a valid view schema.

Lemma 1. *For a global schema GS with multiple inheritance, the Edge-Creation algorithm in Figure 8 generates a view schema with all required but possibly also redundant is-a arcs.*

Proof: See [15] for the proof. ■

```
Input:  GS=(V,E), VS=(VV,VE) with VV⊆V, VE=∅.
Output: VS=(VV,VE) a valid view
   (with M_required its incidence matrix).
Data Structures:
   D, M_global are boolean matrices of size n = |GS|.
   A, B, C, M_view, M_consistent, M_redundant, M_required are
boolean matrices of size n = |VS|.
Algorithm A2: View Generation1.
   procedure Transitive-Closure (A) return B is
      for k from 1 to |A| do
         B := A;
         for i,j from 1 to |A| do
         A[i,j] := B[i,j] or (B[i,k] and B[k,j]);
         endfor endfor

   procedure Bool-Multiply (A,B) return C is
      for i,k from 1 to |VS| do
         C[i,k] := ⋁_{j=1}^{n} ( A[i,j] and B[j,k] );

   procedure Graph-Subtract (A,B) return C is
      for i,j from 1 to |VS| do
         if (A[i,j]=1 and B[j,k]=0)
         then C[i,j] := 1; else C[i,j] := 0; endif

   procedure Reduce-Matrix (D) return B is
      for i,j from 1 to |D| do
         if (C_i ∈ VS) and (C_j ∈ VS) then
         B[C_i, C_j] := D[C_i, C_j]; endif
   algorithm View-Generation2(GS, VS) is
      0.  M_global incidence matrix for GS.
      1.  M_global := Transitive-Closure( M_global );
      2.  M_consistent = Reduce-Matrix( M_global );
      3.  M_redundant :=
            Bool-Multiply( M_view, M_consistent );
      4.  M_required :=
            Graph-Subtract( M_view, M_redundant );
      5.  M_required is incidence matrix for VS;
   end algorithm;
```

Figure 9: View Generation1 Algorithm.

In this case, we need to remove all redundant arcs from the view schema graph. For this, we reformulate our problem of view generation as the graph-theoretic problem called transitive reduction [2]. Namely, the requirement of keeping all *required* and removing all *redundant* edges from VS is equivalent to finding a graph G for a given directed acyclic graph G such that (1) there is a directed path from vertex u to v in G' whenever there is a path from u to v in G and (2) there is no graph with fewer arcs than G' satisfying the first condition" [2]. G' is called the transitive reduction of G. Since the transitive reduction removes all redundant edges from a view schema, we are not concerned with preventing the creation of redundant edges during the edge creation stage. We therefore use a transitive closure algorithm for the initial generation of all required but also *all* redundant *is-a* edges of the view schema. See Figure 9 for the complete algorithm.

Example 7. *In Figure 10, the View-Generation1 algorithm is applied to an example. The input is GS depicted in Figure 10.a. Step 0 initializes the incidence matrix M_{global} for GS (Figure 10.b). Step 1 computes the transitive closure on the is-a relationships in GS as shown in Figure 10.c. Step 2 reduces the incidence matrix M_{global} for GS down to the incidence matrix M_{view} for VS by selecting all classes that belong to VS and all arcs of GS among these classes. (Figure 10.d). The corresponding view schema is depicted in Figure 10.e. We apply the Transitive-Closure procedure to M_{view} to find all required and redundant is-a relationships between all pairs of classes in VS. The result is shown in Figure 10.f and 10.g. Then we apply boolean matrix multiplication to M_{view} and $M_{consistent}$ to find all redundant edges. For instance, the edge <C5,C1> is redundant because <C5,C3> in M_{view} and <C3,C1> in $M_{consistent}$. Figures 10.h and 10.i present the resulting incidence matrix $M_{redundant}$ and the schema, respectively. Lastly, the graph subtraction used in step 3 removes all (redundant) edges of the matrix $M_{redundant}$ in Figure 10.h from those (all required and redundant edges) in the matrix $M_{consistent}$ in Figure 10.f. Clearly, the resulting incidence matrix $M_{required}$ contains all edges required for VS and no others (Figures 10.j and 10.k).* □

Theorem 3. *The View-Generation1 algorithm (Figure 9) generates a valid view schema.*

Proof: Due to the reformulation of view generation as transitive reduction, the proof of Theorem 3 can be directly derived from the proof for the transitive reduction [15]. ■

Theorem 4. ***(Complexity)*** *The worst-case complexity of the View-Generation1 algorithm is $O(|GS|^3)$ with $|GS|$ the number of classes in the global schema GS.*

Proof: See [15]. ■

The complexity of the transitive closure and boolean multiplication algorithms have been shown to be $O(n^{2.37}(logn)^2)$ for large n [3]. Since the size of a schema graph is generally not large, a straightforward implementation of these algorithms as shown in Figure 9 with cube complexity is generally preferable.

Note that the worst-case complexity $O(|GS|^3)$ of the View-Generation1 algorithm is based on calculating the transitive closure on the global schema during the first part of the algorithm (See Figure 9). The second part of the algorithm, which essentially reduces the edges of the view graph, actual operates on the view schema only. We can therefore replace the transitive closure algorithm by the simpler Edge-Creation algorithm which

we introduced earlier (Figure 8). now called View-Generation2. While the Edge-Creation algorithm may generate some redundant *is-a* edges, those would be removed by the second part of the View-Generation2 algorithm.

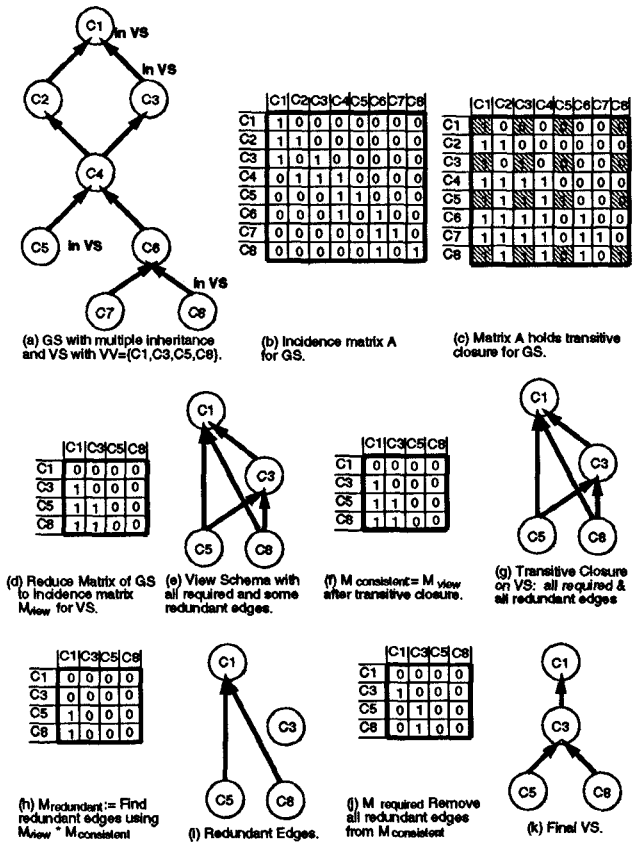

Figure 10: Example of View-Generation1 Algorithm for Creating A Valid Schema.

Theorem 5. *(Correctness) Given a global schema GS=(V,E) with multiple inheritance and a set of view classes VV ⊆ V, the View-Generation2 algorithm generates a valid view VS=(VV,VE).*

Proof: This can easily be shown based on Lemma 1 and Theorem 3. ■

While the Edge-Creation algorithm defined in Figure 8 is of linear complexity when applied to a tree-structured schema graph (Theorem 2), for a DAG structure[7] the complexity is quadratic in the number of nodes and edges in *GS*, i.e., $O(nodes(GS) * edges(GS))$.

Theorem 6. *(Complexity) The worst-case complexity of the View-Generation2 algorithm is* $O(min(|GS| * edges(GS), |VS|^3))$ *with* $|GS|$ *the number of classes in the global schema GS and* $|VS|$ *the number of classes in the view schema VS.*

Proof: For a proof see [15]. ■

[7]We have modified the Edge-Creation procedure defined for tree-structures (Section 5.2) to run efficiently on a DAG graph structure, e.g., by using markers.

6 RELATED WORK

Most proposals for defining views for OODBs suggest the use of the query language of their respective object model to derive a virtual class, e.g., [9], [6], [8], [17], and [1]. Bancilhon and Kim [4] have indicated that current OODBs offer at best some degraded form of views, either through exports of schemas or through encapsulation, but no complete and simple mechanism is yet available. Our *MultiView* approach represents a solution to this problem.

Most approaches in the literature do not discuss the integration of derived classes into the global schema. Instead, the derived classes are treated as 'stand-alone' objects [6] or they are attached directly as subclasses of the schema root [9]. Scholl et al. [17] and Abiteboul et al. [1] both indicate the need for the classification of virtual classes into one schema, but they keep the derived classes into several view schema graphs rather than generating one underlying global schema. Furthermore, they do not consider the problem of enforcing the consistency of view schemata using a view generation approach.

Morsi et al. [12] are developing a graphical interface for DAG rearrangement views of a class hierarchy. This tool is part of a graphical environment for schema evolution and version support, and hence focuses on defining a view using manipulation operations similar in flavor to typical schema evolution operators. The use of a query language to define arbitrary view classes or the classification of such classes are not discussed by their work. Tanaka et al.s early work on schema virtualization [19] does not distinguish between the task of integrating derived classes into a common schema and the task of generating view schemata. Since they allow for the arbitrary addition of *is-a* edges in a virtual schema, their approach would have to deal with identifying and correcting inconsistent schema. They point out that work is needed for developing a definition language for view schemata. In this paper, we have provided a solution for this. Shilling and Sweeney's approach [18] propose that a class has multiple type interfaces instead of having one type. Since their approach focuses on one class only, neither a query language on a complete schema nor the consistent generation of a view schema graph are considered.

7 IMPLEMENTATION STATUS AND CONCLUSIONS

In this paper, we have proposed two mechanisms for simplifying the specification of views in OODBs. First, we introduced a query language for view customization that operates on a complete schema rather than on individual classes. This graph algebra promises to reduce the number of queries necessary for defining a given view. Second, we introduced a tool for generating arbitrarily complex, yet consistent, view schemata. This tool automatically generates a *valid* and *most informative* view generalization hierarchy for a set of user selected view classes, making the manual entry of view *is-a* arcs by the view definer obsolete. Proofs of correctness of these algorithms can be found in [15]. We

have implemented a preliminary prototype of the *MultiView* system (including the view schema generator) using GemStone. Further description of this implementation effort is reported in [10]. We are currently extending the system to also function as a consistency checking tool for user-specified class relationships.

Acknowledgements. My thanks goes to several students at the University of Michigan for their effort on implementing the MultiView prototype, in particular, Harumi A. Kuno, Chris Ma, and Doug L. Moore.

References

[1] Abiteboul, S., and Bonner, A., "Objects and Views," *SIGMOD*, 1991, pp. 238–247.

[2] Aho, A. V., Garey, M.R., and Ullman, J.D., "The Transitive Reduction of a Directed Graph," *SIAM J. Computing*, Vol. 1, No. 2, June 1972.

[3] Aho, A. V., Hopcroft, J. E., and Jeffrey, D. U., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Pub. Company, 1974.

[4] Bancilhon and W. Kim, "OODB Systems: In Transition," *SIGMOD RECORD*, Vol. 19, No. 4, Dec. 1990, pp. 49 – 53.

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

[6] Gilbert, J. P., "Supporting User Views", *OODB Task Group Workshop Proceedings*, Ottawa, Canada, Oct. 1990.

[7] Heiler, S., and Zdonik, S. B., Object views: Extending the vision, In *Proc. IEEE Data Engineering Conf.*, Los Angeles, Feb. 1990, pg. 86 - 93.

[8] D. Fishman et al., "Iris: An Object-Oriented Database Management System," *ACM TOIS*, vol. 5, no. 1, Jan. 1987, pp. 48 – 69.

[9] Kaul, M., Drosten, K., and Neuhold, E.J., "ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views", *Proc. IEEE Data Eng. Conf.*, Feb. 1990, pp. 2 - 10.

[10] Kim, W., A model of queries in object-oriented databases, In *Proc. Int. Conf. on Very Large Databases*, pp. 423 – 432, Aug. 1989.

[11] Kuno, H. and Rundensteiner, E. A., Building A View Manager System Using GemStone, Univ. of Michigan, Technical Report, June 1993.

[12] Maier, D., Stein, J., Otis, A., and Purdy, A., "Development of an Object-Oriented DBMS," in *Proc. OOPSLA'86*, Sep. 1986, pp. 472–482.

[13] Morsi, M. M., Navathe, s. B., Kim, H. K., An Extensible Object-Oriented Database Testbed, *Int. Conf. on Data Eng.*, 1992, pp. 150 – 157.

[14] Richardson, J. P. Schwarz, Aspects: Extending Objects to Support Multiple, Independent Roles; Proc. ACM SIGMOD 1991, 298–307

[15] Rundensteiner, E. A., and Bic, L., "Set Operations in Object-Based Data Models", in *IEEE Trans. on Data and Knowledge Eng.*, Vol. 4, Iss. 4, Aug. 1992, pp. 382 - 398

[16] Rundensteiner, E. A., "*MultiView:* A Methodology for Supporting Multiple View Schemata in OODBs", *VLDB'92*, Aug. 1992.

[17] Rundensteiner, E. A., "Object-Oriented Views: An Approach to Tool Integration in Design Environments," Diss., Info. and Comp. Scie. Dept. Univ. of Cal. Irvine, Tech. Rep. 92-83, Aug. 1992.

[18] Rundensteiner, E. A., "Design Tool Integration using Object-Oriented Database Views," *IEEE Int. Conf. on CAD*, Nov. 1993.

[19] Scholl, M. H., Laasch, C. and Tresch, M., Updatable Views in Object-Oriented Databases, *Proc. 2nd DOOD Conf.*, Muenich, Dec. 1991.

[20] Shilling, J. J., and Sweeney, P. F., Three Steps to Views: Extending the Object-Oriented Paradigm, *OOPSLA'89*, New Orleans , Sep. 1989, 353 – 361.

[21] Tanaka, K., Yoshikawa, M., and Ishihara, K., Schema Virtualization in Object-Oriented Databases, In *Proc. IEEE Data Engineering Conf.*, Feb. 1988, pg. 23 – 30.

[22] Yu and Osborn, "An Evaluation Framework for Algebraic Object-Oriented Query Models," in *Proc. IEEE Data Eng. Conf.*, Feb. 1991.