MultiView: A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases

Elke A. Rundensteiner

Department of Information and Computer Science
University of California, Irvine
January, 1992

Technical Report 92-07

MultiView: A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases

ELKE A. RUNDENSTEINER

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
e-mail: rundenst@ics.uci.edu
telephone: (714) 856-4101
fax: (714) 856-4056
January/1992

Abstract

It has been widely recognized that object-oriented database (OODB) technology needs to be extended to provide a mechanism similar to *views* in relational database systems. We define an *object-oriented view* to be an arbitrarily complex virtual schema graph with possibly restructured generalization and decomposition hierarchies – rather than just one *virtual class* as has been proposed in the literature. In this paper, we propose a methodology, called *MultiView*, for supporting multiple such *view schemata*. *MultiView* breaks the schema design task into the following independent and well-defined subtasks: (1) the customization of type descriptions and object sets of existing classes by deriving virtual classes, (2) the integration of all derived classes into *one* consistent global schema graph, and (3) the definition of arbitrarily complex view schemata on this augmented global schema. For the first task of *MultiView*, we define a set of object algebra operators that can be used by the view definer for class customization. For the second task of *MultiView*, we propose an algorithm that automatically integrates these newly derived *virtual* classes into the global schema. We solve the third task of *MultiView* by first letting the view definer explicitly select the desired view classes from the global schema using a view definition language and then by automatically generating a view class hierarchy for these selected classes. In addition, we present algorithms that verify the closure property of a view and, if found to be incomplete, transform it into a closed, yet minimal, view. In this paper, we introduce the fundamental concept of *view independence* and show *MultiView* to be *view independent*. We also outline implementation techniques for realizing *MultiView* with existing OODB technology.

Index Terms: Multiple view schemata, object-oriented views, view closure property, view consistency, view independence, view definition language, object algebra, schema design support.

Contents

List of Figures

1 INTRODUCTION

Many databases developed for advanced application domains, such as, Computer-Aided Design and Manufacturing, are now being build using object-oriented database (OODB) models. These applications require customized interfaces to the global information suitable for different types of user groups and tasks. We therefore need to develop a technology for OODBs - similar to the view mechanism in relational databases - that would support the construction of various (possibly conflicting) interfaces to the schema by hiding irrelevant portions of the data, by augmenting it, or by restructuring it.

While the concept of views has been studied extensively in the context of the relational model, it is largely unexplored for the newly emerging more powerful OODBs. Some initial proposals of views on OODBs have emerged that define a view to be a *virtual class* derived by an object-oriented query [5, 14, 7]. Note however that an object-oriented data schema is a complex structure of classes interrelated via various relationships, such as, the orthogonal generalization and decomposition hierarchies [7, 8], whereas a relational schema is simply a set of 'unrelated' relations [3]. An object-based view thus should be defined to be a *virtual, possibly restructured, subschema graph* of the global schema [17] rather than just one individual *virtual class* – disjoint from all other classes of the schema. We call this concept of an object-oriented view a *view schema.* The construction of these view schemata raises a number of challenging research issues in terms of how to restructure view schema graphs and how to relate them with the global schema structure.

In this paper, we propose a methodology, called *MultiView*, for supporting multiple such *view schemata* that successfully solves these problems. There are three aspects to *MultiView*: first, it provides mechanisms to the user for specifying a view, second, it helps the user to enforce the consistency of the view schema structure while defining the view, and third, it offers general guidelines on how to implement and maintain these view schemata, once specified. *MultiView* is anchored on the following complementary ideas: (1) customization and derivation of virtual classes, (2) integration of derived classes into *one* consistent global schema graph and (3) the specification of arbitrarily complex view schemata on this augmented global schema. *MultiView* builds on existing work in as much as it is independent of the class derivation operators chosen from the set of proposed operators in the literature [5, 7, 14, 10]. For the purpose of this work, we formally define a set of object-oriented query operators that can be used for the derivation of virtual classes (Section 4).

One of the problems to be tackled is the issue of how a virtual class relates to the remaining classes in the complete schema. This is accomplished by the second task of *MultiView.* Note that in the relational model, where each relation is physically independent from all other relations, the integration of a virtual relation with the global schema corresponds to simply adding it to the list of existing relations (the data dictionary). In the context of OODBs, however, this is less straightforward. A class in an object schema is interrelated with other classes via an is-a hierarchy (for property inheritance and subsetting) and via a property decomposition hierarchy (for forming complex objects). In this paper, we present *validity* criteria that have to be guaranteed by class integration algorithms in order to preserve the consistency of the global schema graph. We then describe an algorithm for the integration of these newly derived *virtual* classes into the global schema that preserves the consistency of the views.

We cannot simply modify the existing global object schema so that it suits the requirements of one particular user. Instead, we need to support a number of *different, potentially conflicting,*

view schemata of the same data model. each of which supporting a particular user's point of view. Consequently, we are concerned here with the virtual restructuring for each given view while maintaining all other view schemata; rather than with permanently changing the global database as is done in schema evolution [2].

Not just individual virtual classes but complete (possibly conflicting) view schemata have to be integrated with another and with the underlying global schema into one consistent whole. This integration has to maintain the difference in the generalization and decomposition hierarchies of the view schemata. The proposed *MultiView* methodology solves this problem by separating the definition of view schemata into two independent steps, namely, one, the integration of virtual classes into *one* consistent global schema graph and, two, the definition of view schemata composed of both base and virtual classes on top of this augmented global schema. View schemata are consistently integrated with one another simply by being consistently integrated with the same underlying global schema. An additional requirement is that the originally specified object schema (with stored rather than derived classes) remains intact so that it can be used by other users, if so desired. The *MultiView* methodology accomplishes this by treating the original base schema as a special non-modifiable view schema.

We solve the third task of *MultiView*, namely, the specification of arbitrarily complex view schemata on the augmented global schema, by dividing it into the following two subtasks: first the explicit selection of view classes from the global schema and second the generation of a view class hierarchy for these selected classes. For the former, we have developed a view definition language that can be used by the view definer to specify the desired view classes. For the latter, we have developed algorithms that automatically generate a consistent view schema generalization hierarchy [12]. In addition, we have developed *consistency* criteria for the property decomposition and the generalization hierarchies of a view schema in terms of their completeness and consistency with the underlying global object schema. *MultiView* will not only verify the consistency of a specified view schema but, if it is found to be inconsistent, will augment the view schema such as to generate a consistent and closed view schema. In this paper, we give algorithms for the automatic generation of the correct property decomposition hierarchy for an initially given set of view classes (Section 7). We present proofs of correctness and a complexity analysis for the closed-view generation algorithm.

In summary, this paper makes the following contributions. First, we extend the concept of an object-oriented view from an individual *virtual class* to a complete *view schema.* This requires the introduction of new concepts, such as, the view *validity*, the view *closure* property, and the *view independence* concept. This clearly represents a step towards the development of a much needed object-oriented database theory. Second, we present a general methodology for supporting *multiple (possible contradicting) view schemata* in OODBs, called *MultiView. MultiView* supports all necessary functionalities of an object-oriented view support system, such as, (1) the virtual modification of the type structure and of the object membership of existing classes, (2) the sharing of property functions and object instances among stored and derived classes without unnecessary duplication, (3) the virtual restructuring of the generalization and the property decomposition hierarchy, (4) the sharing of classes, property functions, and objects among different view schemata, (5) the construction of an arbitrary complex view schema as required by a particular user task, and (6) the integration of each view schema with all other schemata into one 'consistent whole'. Third, we present solutions to the relevant subtasks of proposed view methodology. In particular, we define an object algebra for the derivation of virtual classes and outline an algorithm for integrating these derived classes into one global schema. We have also developed a language for view schema

specification. We present an algorithm for checking the closure property of a view schema specified using the view definition language. Given a non-closed view schema, this algorithm is guaranteed to transform the non-closed view into a closed, yet minimal, view schema. Lastly, we introduce the concept of *view independence*, which we argue to be a fundamental requirement for any view mechanism developed for object-oriented databases – similar in notion to the well-known concept of data independence. In Section 8, we show the *MultiView* methodology to be *view independent.*

The paper is organized as follows. In Section 2, we introduce object-oriented concepts required for supporting multiple view schemata. In Section 3, we outline the *MultiView* methodology. The object algebra is given in Section 4, while class integration is discussed in Section 5. In Section 6, we introduce the view definition language and in Section 7 we present algorithms for generating a closed view. *MultiView* is shown to be view independent in Section 8. We present initial ideas of the realization of *MultiView* in Section 9, compare *MultiView* to related work in Section 10, and conclude with Section 11.

2 BASIC DEFINITIONS

2.1 The Object Data Model

Below, we introduce the basic concepts of OODB models needed for the remainder of the paper. Let O be an infinite set of object instances. Each element o $\in$ O is an instance of an abstract data type (ADT), i.e., it can be manipulated only by means of the interface of the respective ADT. Let P be an infinite set of property functions. Each property function p $\in$ P can be a value from a predefined enumeration type, an object instance from some class, or an arbitrarily complex function. Each property function p $\in$ P has a name and signature (i.e., domain types). Without loss of generality, we assume that the name of a property corresponds to a unique property identifier. Let C be the set of all classes. A **class** $C_i \in$ C has a unique class name, a type description and a set membership. The type associated with a class corresponds to a common interface for all instances of the class, that is, the collection of applicable property functions. We refer to the name of the type associated with a class C by **type**(C) and to the set of property functions defined for C by **properties**(**type**(C)), or short **properties**(C). If p $\in$ P is a property function defined for C, i.e., p $\in$ **properties**(C), then we refer to the domain of the property function p for C by $\mathbf{domain}_p$(C). A **class** is also a container for a set of objects. The collection of objects that belong to a **class** C is denoted by **extent**(C) := $\{o \mid o \in C\}$ with the member-of predicate "$\in$" defined based on the object identities of the object instances [11]. We can now define the class relationships.

Definition 1. *For two classes C1 and C2* $\in$ *C, 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)).*

Definition 2. *For two classes C1 and C2* $\in$ *C, 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**$_p$(*C1*) $\subseteq$ **domain**$_p$(*C2*)).

The first condition of Definition 2 states that a subtype must have the same attribute as its supertype and possibly additional ones. The second condition states that the domains of the attributes of a subtype must be contained within the domains of the attributes of the supertype, but that they could possibly be restricted.

Definition 3. *For two classes C1 and C2 $\in$ C, C1 is called a* **subclass** *of C2, denoted by C1 is-a C2, if and only if (C1 $\preceq$ C2) and (C1 $\subseteq$ C2).*

Informally, we say that C1 is *is-a* related to C2 if (1) every member of C1 is a member of C2 (the subset relationship) and (2) every property defined for C2 is also defined for C1 (the subtype relationship). The three class relationships are *reflexive, antisymmetric* and *transitive.* By reflexivity, the *is-a* relationship (C_i *is-a* C_i) holds for all C_i. By antisymmetry, the *is-a* relationships (C_i *is-a* C_j) and (C_j *is-a* C_i) imply (C_i=C_j). By transitivity, the *is-a* relationships (C_i *is-a* C_j) and (C_j *is-a* C_k) imply (C_i *is-a* C_k).

Next, we introduce operations on type descriptions which form a new type based on the type descriptions of two classes.

Definition 4. *Let C1 and C2 be two classes with the types t1 and t2 in T, respectively. Then $\sqcup$ is a function from $T^2 \rightarrow T$ that defines a new type t3 by t3 := t1 $\sqcup$ t2. The property functions of the new type t3,* **properties***(t3), are defined by* **properties***(t3) :=* **properties***(t1) $\cup$* **properties***(t2.). For each property function p $\in$* **properties***(t3), the domain* $\mathbf{domain}_p(t3) :=$ $\mathbf{domain}_p(t1) \cap \mathbf{domain}_p(t2)$ *is defined.*

Definition 5. *Let C1 and C2 be two classes with the types t1 and t2 in T, respectively. Then $\sqcap$ is a function from $T^2 \rightarrow T$ that defines a new type t3 by t3 := t1 $\sqcap$ t2. The property functions of t2 are defined by* **properties***(t3) :=* **properties***(t1) $\cap$* **properties***(t2). For each property p $\in$* **properties***(t3), the domain* $\mathbf{domain}_p(t3) :=$ $\mathbf{domain}_p(t1) \cup \mathbf{domain}_p(t2)$ *is defined.*

Definition 4 defines t3 := t1 $\sqcup$ t2 to be the *greatest common subtype* of t1 and t2, and Definition 5 defines t3 := t1 $\sqcap$ t2 to be the *lowest common supertype* of t1 and t2.

Given a collection of classes for a particular database application, we want to organize them in a fashion such that these class relationships are explicitly represented rather than having to recompute them continuously. The subset class relationship can be used to determine the containment of the object instances associated with one class within the extent of another class. This may for instance be useful for query processing where we need to build the union of two classes. If it is known that one of the two classes is a subset of the other, then the union result corresponds simply to the larger of the two classes. No actual query processing is required. The maintenance of the subtype relationship on the other hand is useful for the reuse of property function code; this feature is commonly known as property inheritance.

Let S = $\{C_i | i = 1, ..., n\}$ be a set of classes. We call C_1 a *direct subclass* of C_n and C_n a *direct superclass* of C_1 if (C_1 *is-a* C_n) and ($C_1 \neq C_n$) and there are no other classes $C_{k_j} \in$ S (with j=1, ..., m) for which the following *is-a* relationships hold: (C_1 *is-a* C_{k_1}) and (C_{k_1} *is-a* C_{k_2}) and ... and (C_{k_m} *is-a* C_n). C_1 is called an *(indirect) subclass* of C_n and C_n an *(indirect) superclass* of C_1 if there are one or more classes $C_{k_j} \in$ S (with j= 1,2, ..., m) for which the above *is-a* relationships hold.

This *indirect subclass* relationship between C_1 and C_n is denoted by (C_1 *is-a** C_n) for (j≥0) and by (C_1 *is-a+* C_n) for (j≥1). A graph-theoretic representation of a set of classes S that explicitly represents all *direct subclass* relationships among the classes in terms of edges is defined below.

Definition 6. *An* **object schema** *is a directed acyclic graph*[1] *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 × V that represents all direct is-a relationships between all pairs of classes in V. In particular, each directed edge e from C_1 to C_2, denoted by e = <C_1, C_2>, represents the direct is-a relationship between the two classes (C_1 is-a C_2).*

We refer to the collection of *is-a* relationships of a set of classes as the **generalization hierarchy** of the object schema. Since the *is-a* relationship is *reflexive, antisymmetric* and *transitive*, the generalization hierarchy graph (or schema graph) is a directed acyclic graph without any loops. Furthermore, since we only store the direct subclass relationships, there will be no self-loops in a schema graph. An edge e=<C_i, C_j> is called a self-loop if its source node C_i and its sink node C_j are identical, i.e., i=j. The schema graph also has no multi-edges, since each direct subclass relationship is stored but once. Two or more edges are called multi-edges if they have the same source and the same sink node, respectively. For instance, the edges e1=<C_i, C_j> and e2=<C_k, C_l> with (i=k) and (j=l) are multi-edges.

Once these class relationships are compiled and maintained in this graph format, we can read them directly from the structure of the graph without having to repeatedly compute the *subclass* relationships. For instance, C_1 is a *direct subclass* of C_n if the edge e = <C_1, C_n> exists in E. C_1 is an *indirect subclass* of C_n, denoted by (C_1 *is-a** C_n), if there is a path through the class hierarchy of length one or longer connecting C_1 and C_n. More formally, if there are one or more classes C_{k_j} ∈ V (with j=1,2, ..., m) with the edges e_1 = <C_1, C_{k_1}>, e_2 = <C_{k_1}, C_{k_2}>, ..., e_{m+1} = <C_{k_m}, C_n> in E. Finally, a path of length two or larger represents the subclass relationship (C_1 *is-a+* C_2).

A schema has one designated root node, the class called Object, which is the superclass for all classes in the schema. This Object class contains all object instances of the database and its type description is empty. All edges in a schema are directed from the designated root node Object to the leaf nodes of the graph. This assures that the schema graph is one DAG rather than consisting of multiple possibly disconnected subgraphs.

As discussed earlier in this section, a class is related to other classes via property relationships. For example, if the class C1 has defined a property function p with the **domain**$_p$(C1):=C2, then we say that there is a property decomposition arc between C1 and C2 labeled 'p'. We refer to the set of all property relationships among the classes of a schema as its **property decomposition hierarchy**.

Definition 7. *Let S=(V,E) be an object schema as defined above. Let L be a set of labels that correspond to the names of the property functions in P. Then the* **property decomposition hierarchy** *of the schema S is defined to be a directed graph PD=(V,A,L) with V the set of vertices and A the set of arcs. A is a ternary relation on V × V × L, called the (labeled) property decomposition edges. An edge a = (C1,C2,l) ∈ A if and only if there is a property function defined for class C1 with the property label l and the domain class C2.*

[1] A schema without multiple inheritance corresponds to a tree rather than a DAG.

A property decomposition hierarchy consists of one or more disconnected subgraphs with possibly loops, self-loops, and multi-edges. The latter are distinguished based on their associated labels. An object schema has both a property decomposition hierarchy and a generalization hierarchy and thus could be defined to be a graph G = (V,E,A,L) with V, E, A, and L given in Definition 7.

2.2 Object-Oriented Views

We distinguish between **base** and **virtual** classes. **Base classes** are defined during the initial schema definition. Object instances that are members of base classes are explicitly stored as base objects. **Virtual classes** are defined during the lifetime of the database using some object-oriented queries, i.e., their definitions are dynamically added to the existing schema. A virtual class has an associated membership derivation function that will determine its exact membership based on the state of the database. The extent of a virtual class is generally not explicitly stored, but rather computed upon demand.

Definition 8. *The* **base schema** *(BS) is an object schema S=(V,E), where all nodes in V correspond to base classes with stored rather than derived object instances.*

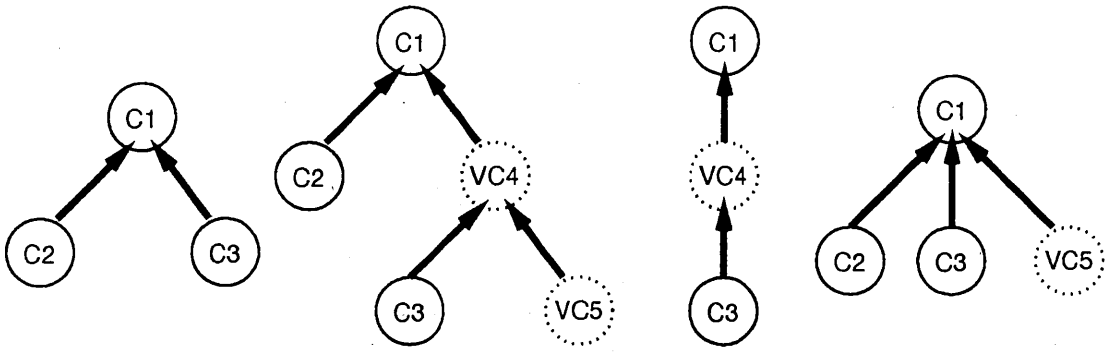
Definition 9. *Let BS be a* **base schema**. *The* **global schema** *(GS) is an extension of the base schema that is augmented by the collection of all virtual classes defined during the lifetime of the database as well as is-a relationships among this extended set of classes.*

A subgraph of the global schema which contains only virtual classes and their *is-a* relationships is commonly called a *virtual schema* [17].

Definition 10. *Given a global schema GS=(V,E), then a* **view schema** *(VS), or short, a* **view**, *is defined to be a schema VS= (VV,VE) with the following properties:*

1. *VS has a unique view identifier denoted by* $< VS >$,
2. $VV \subseteq V$, *and*
3. $VE \subseteq$ *transitive-closure(E).*

The first condition states that each view schema is uniquely identifiable. The second property states that all classes of VS also have to be classes in *GS*, i.e., they have been properly integrated with the global information. The third property states that the view schema maintains only *is-a* relationships among its view classes that are directly derivable from *GS*. In other words, an edge $< C_i, C_j >$ can only exist in VE if either $< C_i, C_j >$ exists directly in E or if it is indirectly derivable via the transitivity of the *is-a* relationship, i.e., only if $(C_i \ isa^* \ C_j)$ in GS. A view schema is a special case of an object schema. Therefore all properties of a general schema defined in Section 2.1. must also hold. We call the classes in a view schema (both the base and the virtual ones) *view classes* and the *is-a* relationships among these view classes *view is-a relationships.*



(a) Base schema BS. (b) Global schema GS. (c) View schema VS1. (d) View schema VS2.

Figure 1: Examples of Base, Global and View Schemata.

Example 1. *Figure 1 shows the relationship between (a) the base schema, (b) the global schema, and (c) and (d) two different view schemata. We depict base and virtual classes by circles and dotted circles, respectively. The global schema in Figure 1.b is derived from the base schema in Figure 1.a by adding virtual classes, namely, VC4 and VC5, and by interconnecting them with the remaining classes to create a valid schema. The view schematas in Figure 1.c and 1.d are derived from the global schema by selecting a subset of its classes and interconnected them into a valid schema using view is-a arcs.*

Note that the base schema is a special case of a view schema that consists exclusively of all base classes and no virtual classes. We will maintain the base schema as a view schema, i.e., there will be a view object table (or base table) that lists all base classes and their *is-a* relationships (See Section 9). This is important so that users of the data model can see the original data model of the application domain without having to consider derived information. This base table is a special view table in as much as it is predefined and not modifiable.

2.3 The Validity Criterion of the Generalization Hierarchy of a View

Next, we introduce criteria for evaluating the consistency of view schemata with the underlying global schema. An object schema is composed of two orthogonal hierarchies, namely, the class generalization hierarchy and the property decomposition hierarchy. The validity of a view schema thus has to assure the consistency of both hierarchies. This section addresses the consistency of the generalization hierarchy, called *is-a validity*, while the next section will discuss the consistency of the property decomposition hierarchy, called *closure*. These two consistency criteria are orthogonal concepts similar to their underlying class relationship hierarchies. Hence a view can be *closed* without being *is-a valid*, and vice versa.

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

Definition 11. *Given a view schema VS=(VV,VE) defined on the global schema GS=(V,E).*

a. *For all classes C_1,C_2 in VV, an is-a arc from source C_1 to sink C_2 is* **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) in GS and (C_x is-a* C_2) in GS.*

b. *For all classes C_1,C_2 in VV, an is-a arc from source C_1 to sink C_2 is* **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.*

c. *For all classes C_1,C_2 in VV, an is-a arc from source C_1 to sink C_2 in VS is* **inconsistent** *if the edge <C_1,C_2> is in VE and not(C_1 is-a* C_2) in GS.*

d. *The view schema VS=(VV,VE) is* **is-a valid** *(or* **valid***) if the set VE of all its view is-a relationships contains* **all required** *and* **no redundant** *and* **no inconsistent** *arcs in VS.*

An **is-a valid** view schema is complete since, by Definition 11.a, two classes C_1 and C_2 in VS are *is-a* related in VS if and only if they are also *is-a* related in GS. An **is-a valid** view schema is minimal since, by Definition 11.b, there is no direct *is-a* arc between two classes if if there is already an indirect *is-a* path between them. Lastly, an **is-a valid** view schema is consistent since, by Definition 11.c, an *is-a* arc from source C_1 to sink C_2 exists in *VS* if and only if the two classes are *is-a* related in GS. We demonstrate these concepts with the example given below.

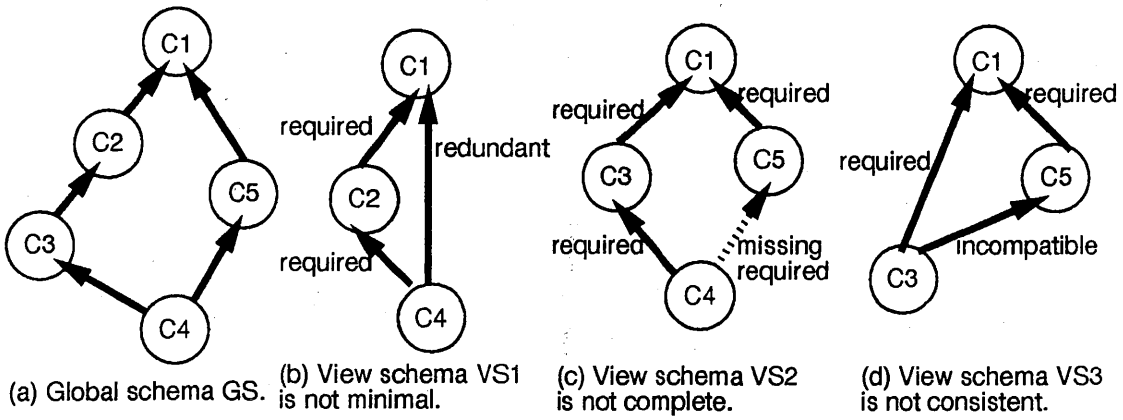


Figure 2: Examples of Valid and Invalid View Generalization Hierarchies.

Example 2. *Figures 2.b, 2.c, and 2.d depict three different view schemata defined on the global schema GS depicted in Figure 2.a. The view VS1 in Figure 2.b is not a valid view schema, since it contains the redundant edge $e_{4,1} = < C_4,C_1>$. The edge $e_{4,1}$ can be removed from VS1 without losing the information that (C_4 is-a C_1), since by transitivity, the relationships (C_4 is-a C_2) and (C_2 is-a C_1) imply the relationship (C_4 is-a* C_1). The view VS2 in Figure 2.c is not is-a valid, since the required edge $e_{4,5} = < C_4, C_5 >$ is missing. Edge $e_{4,5}$ has to be added to the schema to indicate the information that (C_4 is-a C_5). The view VS3 in Figure 2.d is not is-a valid, since it violates the consistency criterion. The edge $e_{3,5} = < C_3, C_5 >$ is inconsistent in VS, since the relationship (C_3 is-a* C_5) does not hold in GS.*

2.4 The Closure Criterion of the Property Decomposition Hierarchy of a View

The type closure concept has been proposed in the literature as a criterion for the validity of a property decomposition hierarchy of a view schema [17, 5]. Here, we define a variation of this closure criterion suitable for our underlying object model and the *MultiView* methodology. In Section 7, we present an algorithm for (1) checking a view schema for the closure property and (2) transforming a non-closed view into a closed view.

Let the function *Uses(C)* represent the set of classes that are used by C's type interface, i.e., the domain classes used by the property functions p defined for the class C. For example, if p corresponds to an object pointer defined by $\mathbf{domain}_p$(C):=C2, then *Uses(C)* contains the domain class C2 of the property p. Below, we define the *Uses()* function in graph-theoretic terms based on Definition 7.

Definition 12. *Let G=(V,E) be an object schema as defined in Definition 6 and PD=(V,A,L) be the matching property decomposition hierarchy of G as defined in Definition 7. Let C be a finite set of classes. Then Uses:* $C^2 \longrightarrow C^2$ *is a function defined as follows:*

For all Ci, Cj $\in$ *C and for all property labels pk* $\in$ *L,*

$$Uses(Ci) := \{\ Cj \in C \mid a_{ij} =< Ci, Cj, pk > \in A\ \}.$$

For sets of classes $S \subseteq C$,

$$Uses(S) := \cup_{Ci \in S} Uses(Ci).$$

*We define the unary closure operator * on the Uses relationship by*

$$Uses^*(Ci) := \cup_{Ci \in S} Uses^j(Ci)$$

with

$$Uses^1(Ci) := Uses(Ci)$$

and

$$Uses^j(Ci) := Uses(Uses^{j-1}(Ci)) \text{ for } j > 1.$$

Uses(Ci) corresponds to the domain classes of property functions that are directly used by Ci, while *Uses*(Ci)* corresponds to the domain classes of property functions that are directly or indirectly via transitive closure used by Ci. For the following we assume that a *Uses(Ci)* set is explicitly associated with each class Ci.

Informally, we define a view VS to be *closed* if it defines all classes that it uses, i.e., if it contains all classes that are in the *Uses*(C)* sets of its view classes. A formal definition of the *closure* criterion of a view is given next.

Definition 13. *A view schema VS=(VV,VE) is defined to be a* **type-closed** *(or* **closed***) view if the following holds:* $VV = (\bigcup_{C_i \in VV}(Uses^*(C_i))) \cup VV$.

The closure criterion assures that all classes that are explicitly being used in a view schema (i.e., whose class names are visible in the *Uses** set of a view class) are also defined within the view (i.e., they themselves are view classes). This closure property can be checked by determining for each class C in the view whether all classes that it uses (i.e., *Uses*(C)*) are also in the view. If this test returns true, then the view is *closed.* If the test fails for one or more classes, then the view is not *closed.*

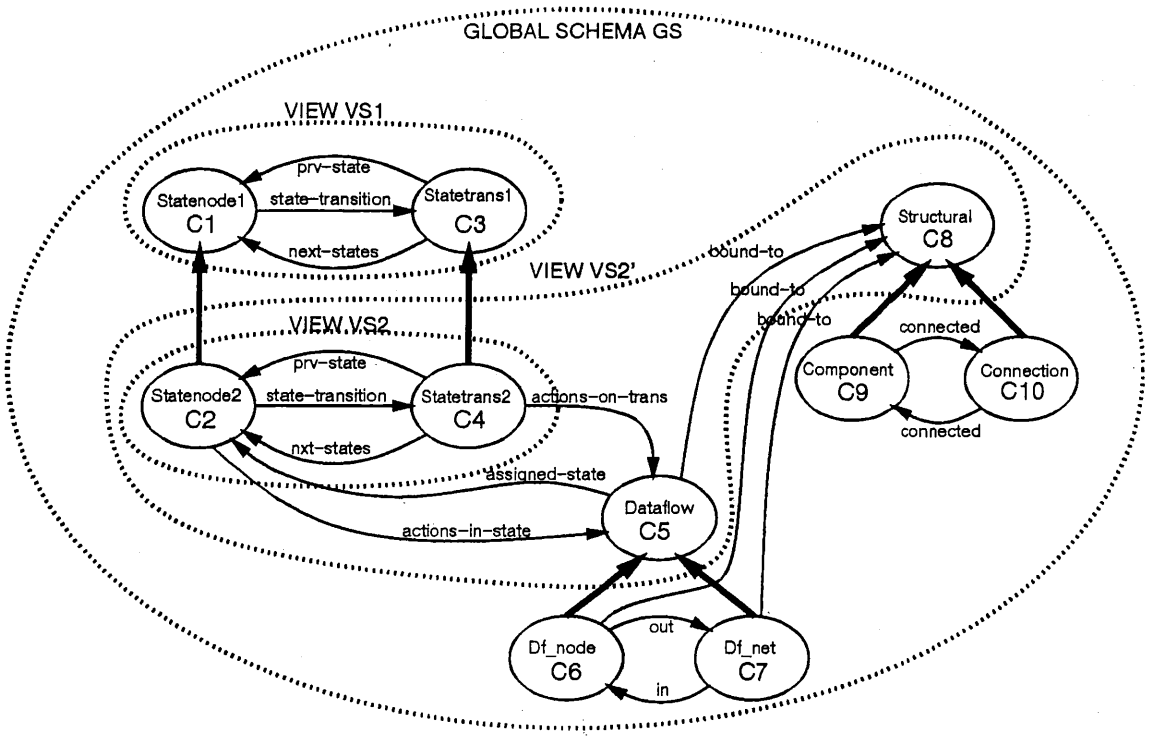


Figure 3: Examples of Closed and Non-Closed Views.

Example 3. *In Figure 3, we present examples of closed and non-closed view schemata defined on the global schema GS modeling a CAD application. In this figure, the is-a and property decomposition relationships are depicted by bold dark arrows without labels and by regular arrows with labels, respectively. The labels for the later correspond to the property names of the represented property functions. A (view) schema is denoted by encircling its (view) classes by a dotted line. The view VS1 =* $\{Statenode1, Statetrans1\}$ *is type-closed, since the domains of the properties defined for its view classes Statenode1 and Statetrans1 are members of the view schema. The view VS2 =* $\{Statenode2, Statetrans2\}$ *is not type-closed. As can easily be seen, the 'actions-in-state' property defined for the view class Statenode2 has the domain class Dataflow, which is not contained in the view VS2. This is graphically depicted in Figure 3 by the dangling property relationship arrow labeled 'actions-in-state' leaving the view VS2. The view VS2' =* $\{Statenode2, Statetrans2, Dataflow, Structural\}$ *is type-closed. Note that the view VS2' contains all view classes of VS2, and thus is a superset of VS2. We will show in a later section that VS2' is the unique minimal view that is both a superset of VS2 and that is type-closed.*

3 THE *MultiView* METHODOLOGY

In this section, we outline our approach for supporting *multiple view schemata* in OODBs, called the *MultiView* methodology. *MultiView* is anchored on the following complementary ideas:

1. customization of existing type structures and object sets by deriving virtual classes via object-oriented queries,
2. integration of derived classes into *one* consistent global schema graph, and
3. the specification of arbitrarily complex view schemata composed of both base and virtual classes on top of this augmented global schema.

The separation of the view schema design process into a number of well-defined subtasks has several advantages. First, it simplifies the view specification and maintenance process, since each of the subtasks can be solved independently from the others. Second, it increases the level of schema design support that can be provided to the view definer by allowing for the automation of some of the subtasks. In Section 7, we present, for instance, algorithms that automate the second subtask of integrating derived classes into *one* consistent global schema graph. Similarly, we have proposed algorithms for the automatic generation of the view schema hierarchy. The later reduces the third subtask of view schema specification to the simple task of selecting classes to be included in the view schema. Furtheremore, the integration of virtual into one global schema assures the consistency of all views with the global schema and with one another. Lastly, the definition of an arbitrary view schema on top of the augmented global schema provides the flexibility to define practically any desired view schema.

The first subtask of *MultiView* supports the virtual customization of existing classes by deriving new virtual classes with a possibly modified type description and membership extent. *MultiView* uses these class derivation mechanisms for a number of different purposes, e.g., to customize type descriptions, to limit the access to property functions, to collect object instances into groups meaningful for the task at hand, and so on. For this we assume that virtual classes are derived from the global schema using object-oriented queries. This fulfills the first feature required for a view support system listed in Section 1.

While there is no generally agreed-upon object algebra, there are a number of proposals for object algebras in the literature (e.g., see [7, 5, 14]). For the purpose of this work (and for the first prototype implementation of *MultiView*), we define our own object algebra, which is similar in flavor to the ones proposed in the literature. (see Section 4). Our treatment of the object algebra focuses on the *subset*, *subtype* and *subclass* relationships among the source and result classes, since this is the foundation for successfully addressing the class integration problem. This issue is generally ignored in the literature on object algebras. We want to stress that *MultiView* is independent from the particular choice of operators.

MultiView supports the integration of virtual classes into one underlying global schema. This integration takes care of the maintenance of explicit relationships between stored and derived classes in terms of type inheritance and subset relationships. This is useful for sharing property functions and object instances consistently among classes without unnecessary duplication. It also is a necessary basis for the third subtask of *MultiView*, namely, for the formation of arbitrarily

complex view schema graphs composed of both base and virtual classes. If the virtual classes are not integrated with the classes in the global schema, then a view schema would correspond to a collection of possibly 'unrelated' classes rather than a generalization schema graph as defined in Definition 10. Details of the class integration process are given in Section 5.

The third subtask of *MultiView* utilizes this augmented global schema graph for the selection of both base and virtual classes and for arranging these view classes in a consistent class hierarchy, called a *view schema.* This phase handles all remaining requirements for a view support system listed in Section 1. It supports for instance the virtual restructuring of the *is-a* hierarchy by allowing to hide from and to expose classes within a view schema. For the explicit selection of view classes from the global schema, we have developed a view schema definition language that can be used by the view definer to specify the classes required for a particular view schema (see Section 6).

In addition, we present in this paper an algorithm for checking the closure property of a view schema graph. Given a non-closed view schema, the algorithm will automatically generate a closed view schema that contains the minimal number of view classes required to make the view closed (Section 7). Lastly note that the *is-a* relationships among the set of selected view classes of a view schema are dictated by their subset and subtype relationships as defined in Section 2. Inserting arbitrary *is-a* relationships between classes in a view schema may result in an incorrect schema in terms of property inheritance and subset relationships. Therefore, rather than requiring the manual insertion of *view is-a* arcs by the view definer, we have developed algorithms that automatically augment the set of selected view classes to generate a *valid* view schema class hierarchy [12].

To make the presented ideas more concrete we now give an example of the steps involved in constructing a *view schema* in *MultiView.*

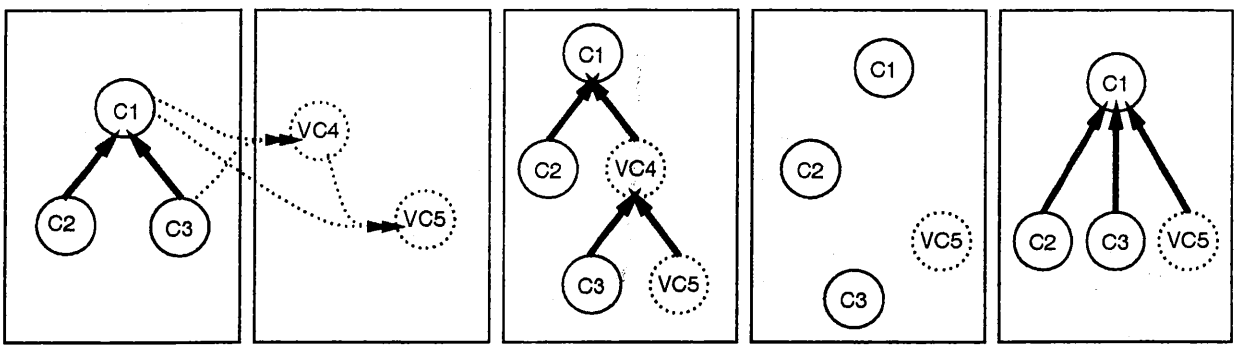


Figure 4: The *MultiView* Approach: From Base over Global to View Schematas.

Example 4. *This example of the view schema construction process is based on Figure 4. In this figure we depict base and virtual classes by circles and dotted circles, respectively. Given the global schema GS in Figure 4.a, the view definer first specifies the two virtual classes VC4 and VC5 using object-oriented queries (Figure 4.b). Class VC4, for instance, is derived based on the two source classes C1 and C3 as depicted by the dotted arrows pointing from Figure 4.a to Figure 4.b. The integration of the virtual classes VC4 and VC5 into GS is given in Figure 4.c. View schema definition now proceeds by selecting a subset of classes from the augmented schema GS. As depicted in Figure 4.d, the selected view classes can be both base and virtual classes. Lastly, the chosen view*

classes are interconnected into one schema graph. The resulting virtual schema graph, called a view schema, is given in Figure 4.e.

4 CLASS CUSTOMIZATION USING OBJECT ALGEBRA

4.1 The Type versus Set Aspect of a Class

The *MultiView* methodology is independent from the object algebra chosen for the class derivation subtask. However, since there is no agreed-upon standard for object algebra, we present below a representative set of algebra operators. The result of a class derivation using an algebra operator is a virtual class VC which has a possibly derived type description and a derived membership extent. We have shown the distinction between the type and the set aspect of classes to be a valuable tool for characterizing the semantics of query operators on object-based data models [10]. In this paper, we will also utilize this distinction for the definition of the object algebra operators. As defined in Section 2, a *type* description of a class determines which property functions can be used to accessthe instances associated with the class. A *set* aspect of a class refers to the set of objects that are members of this class. In the section, we describe the semantics of object algebra operators by defining their effect on the type and the set aspect of the resulting virtual class.

We distinguish between *type* and *set manipulating* query operators. *Type manipulating* operators restrict or elaborate on the type description of an existing class and determine which property functions can be applied to the set of objects associated with the class. They thus limit the visibility of property functions and the access rights to the underlying object instances. A typical example is the **hide** operator, which is similar to the **project** operator used in relational algebra. *Set manipulating* operators group sets of objects into smaller constrained sets or combine several sets into larger sets of objects. A typical example is the **select** operator, also called a predicate-based query [7], which is similar to the **selection** operator used in relational algebra. Some operators manipulate both the type and the set properties of classes. The set operators, such as, **union**, **intersection**, and **difference**, fall into this category.

Note that the 'derivation relationship' between the source classes (arguments to a query operator) and the derived class (the result class of the query operation) do not necessarily correspond to *is-a* relationships between these classes. As we will demonstrate in this section, the resulting class relationships depend on the type of the query operator. The determination of these *subclass* relationships is a necessary basis for the integration of virtual classes into the global schema; and thus is generally not covered by object algebra proposals presented in the literature.

4.2 The Hide Operator

The **hide** operator modifies the type description of a class by hiding some of its property functions. It is similar to the **project** operator in the classical relational algebra, which projects some columns from a relation. It has the following syntax:

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

with <prop-functions> being one or more property functions defined for the class <*source*-class>. The semantics of the **hide** operator are to remove the property functions listed in the set <prop-functions> from the source class while preserving all other property functions visible in the class. More formally stated,

type(<*virtual*-class>) := {p ∈ P | p ∈ **properties**(<*source*-class>) ∧ p ∉ <prop-functions>},

By Definition 2, the type of the <*virtual*-class> is a supertype of the type of the <*source*-class>, since some of the property functions defined for the <*source*-class> are not defined for the <*virtual*-class>. This is denoted by <*source*-class> ⪯ <*virtual*-class>. The extent of the result class is equal to the extent of the source class, denoted by

extent(<*virtual*-class>) := **extent**(<*source*-class>).

By default, this means that <*source*-class> ⊆ <*virtual*-class>. By Definition 3, this implies that the <*source*-class> is a subclass of the <*virtual*-class>, denoted by <*source*-class> *is-a* <*virtual*-class>.

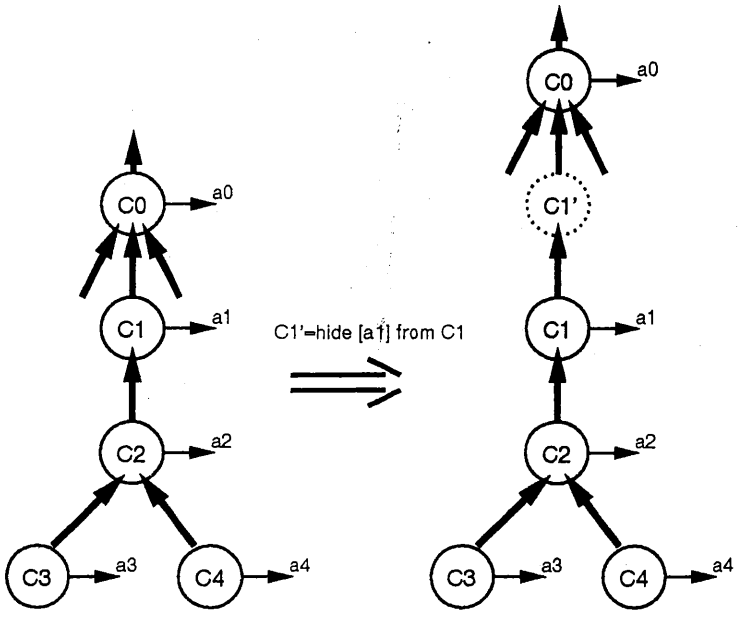


Figure 5: The Hide Class Operator.

Example 5. *An example of the* **hide** *class operator is given in Figure 5. In this figure, we only represent the properties that are explicitly defined for a class and not those that are inherited from other classes. The following query is used to derive the virtual class C1' from the source class C1:*

C1' := **hide** *[a1]* **from** *(C1). As discussed above, we have* **extent***(C1') :=* **extent***(C1), i.e., C1* $\subseteq$ *C1'. Also* **type***(C1)* $:= \{a0, a1, ...\}$, **type***(C1')* $:= \{a0, ...\}$, *and C1* $\preceq$ *C1'. The is-a relationship (C1 is-a C1') is indicated in the figure by the edge from C1 to C1'. In this example, the virtual class C1' has been integrated into the global schema by placing C1' as direct superclass above C1.*

Note that if the property function a1 is not directly defined for the class C1 but inherited from another class, then this integration of the virtual class C1' into the global schema may create other intermediate classes up to the class where the attribute a1 has been originally defined. A discussion of this is beyond the scope of the paper.

4.3 The Refine Operator

The **refine** operator is a type manipulating operator that adds additional property functions to a type rather than removing existing ones. It is similar in flavor to calculating a derived value for each tuple of a relation and then adding (joining) this derived value to the relation in the form of an additional column. It has the following syntax:

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

with <prop-function-def> being the definition of a new property function in form of a new property name and a function body with the latter a legal arithmetic, boolean or set expression. For instance, the expression "age := today-date - birth-date" is an example of a legal <prop-function-defs>. The property functions in <prop-function-defs> are assumed to be distinct from all other property functions in the global schema; and we therefore associate a unique property identifier with them. The semantics of the **refine** operator are to refine the type description of the source class by adding one or more new derived attributes to the source class, namely, those listed in <prop-function-defs>. All other attributes visible in the source class are preserved. More formally stated,

type(<*virtual*-class>) := {p $\in$ P | p $\in$ **properties**(<*source*-class>) $\vee$ p $\in$ <prop-function-def>}.

The type of the <*virtual*-class> is a subtype of the type of the <*source*-class>, since all of the property functions defined for the <*source*-class> are also defined for the <*virtual*-class> (Definition 2), i.e., <*virtual*-class> $\preceq$ <*source*-class>. The content of the result class is again equal to the content of the source class,

extent(<*virtual*-class>) := **extent**(<*source*-class>).

By default, <*virtual*-class> $\subseteq$ <*source*-class>. By Definition 3, this implies <*virtual*-class> *is-a* <*source*-class>.

Example 6. *An example of the* **refine** *class operator is given in Figure 6. The following query is used to derive the virtual class C2' from the source class C2: C2' :=* **refine** *[m1 := fct]* **for** *(C2). We have* **extent***(C2') :=* **extent***(C2), i.e., C2'* $\subseteq$ *C2. The type of C2' has been extended by the new method m1,* **type***(C2)* $:= \{a0, a1, ...\}$, **type***(C2')* $:= \{a0, a1, m1, ...\}$, *and C2'* $\preceq$ *C2. In this example, the virtual class C2' is integrated into the global schema by placing C2' below C2 as direct subclass. This is-a relationship (C2' is-a C2) is indicated in the figure by the edge from C2' to C2.*

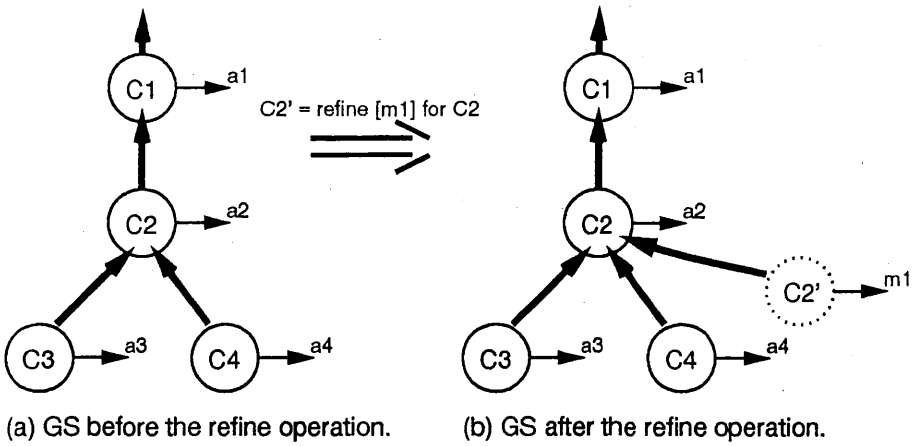


Figure 6: The Refine Class Operator.

4.4 The Select Operator

The **Select** operator is a set manipulating operator that selects a subset of object instances from a given set of objects – similar to the selection operator defined for relational algebra [3]. The select operator has the following syntax:

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

with the <predicate> being some possibly complex function on the source class and its type description. Its semantics are to return a subset of object instances of the source class based on the evaluation of the associated predicate, namely, all object instances that satisfy the predicate are collected into the virtual class. More formally stated,

extent(<*virtual*-class>) := {o ∈ O | o ∈ <*source*-class> ∧ <predicate>(o) = true}.

The extent of the virtual class derived by selection thus is a subset of the extent of the source class, denoted by <*virtual*-class> ⊆ <*source*-class>. Furthermore, the type description defined for the derived class is unchanged, i.e., we have

type(<*virtual*-class>) := **type**(<*source*-class>).

By default, <*virtual*-class> ⪯ <*source*-class>. By Definition 3, this implies <*virtual*-class> *is-a* <*source*-class>.

Example 7. *An example of the* **select** *class operator is given in Figure 7. The query "C2' :=* **select from** *(C2)* **where** *(<pred>)" is used to derive the virtual class C2' from the source class C2. Then C2' ⊆ C2, and* **type***(C2') :=* **type***(C2). The is-a relationship (C2' is-a C2) has been added to Figure 7.b as indicated by the edge from C2' to C2.*

4.5 The Union Operator

Set operators, like the **union** operator, manipulate both the type description and the set membership of their two source classes. A detailed analysis of these set operators for OODBs can be found

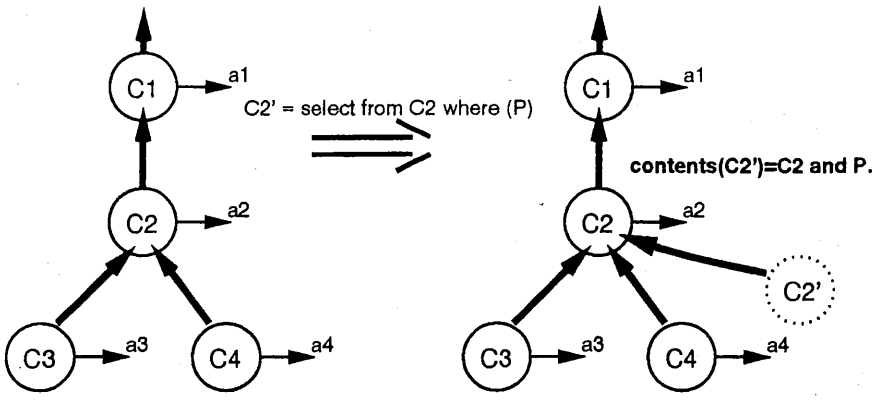


Figure 7: The Selection Class Operator.

in [10]. For the purpose of this work, we use simple semantics (rather than utilizing a more flexible scheme for property inheritance proposed in [10]). The **union** operator has the following syntax:

<*virtual*-class> := **union**(<*source*-class1>,<*source*-class2>).

Its semantics are to return a set of object instances composed of the members of both source classes. More formally stated,

extent(<*virtual*-class>) := {o ∈ O | o ∈ <*source*-class1> ∨ o ∈ <*source*-class2>}.

The semantics of the union operator imply the following subset relationships <*source*-class1> ⊆ <*virtual*-class> and <*source*-class2> ⊆ <*virtual*-class>. Furthermore, the type description of the virtual class is equal to the lowest common supertype of the two sources classes as defined in Definition 5. This is denoted by

type(<*virtual*-class>) := **type**(<*source*-class1>) ⊓ **type**(<*source*-class2>).

This implies the following two subtype relationships <*source*-class1> ≼ <*virtual*-class> and <*source*-class2> ≼ <*virtual*-class>. This then implies the subclass relationships <*source*-class1> *is-a* <*virtual*-class> and <*source*-class2> *is-a* <*virtual*-class>.

Example 8. *An example of the* **union** *class operator is given in Figure 8. The query "Cx' :=* **union***(C2,C5)" is used to derive the virtual class Cx' from the source classes C2 and C5. Then* **extent***(Cx') :=* **extent***(C2) ∪* **extent***(C5). Hence C2 ⊆ Cx' and C5 ⊆ Cx'. Also* **type***(Cx') :=* **type***(C2) ⊓* **type***(C5). Hence C2 ≼ Cx' and C5 ≼ Cx'. The is-a relationships (C2 is-a Cx') and (C5 is-a Cx') are indicated in Figure 8 by the edges from C2 to Cx' and from C5 to Cx', respectively.*

4.6 The Intersection Operator

The **intersect** operator manipulates both the type description and the set membership of the two source classes. It has the following syntax:

<*virtual*-class> := **intersect**(<*source*-class1>,<*source*-class2>).

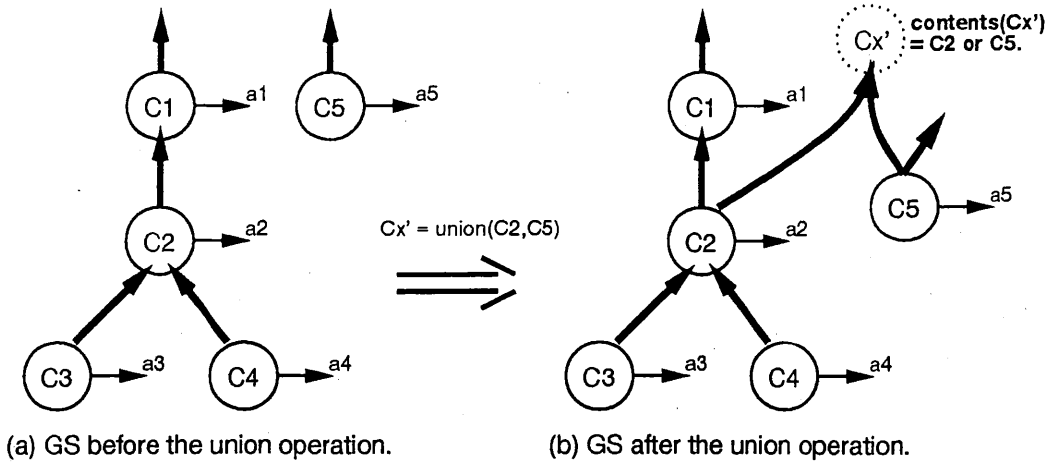


Figure 8: The Union Class Operator.

Its semantics are to return a set of object instances that are members of both source classes. More formally stated,

extent(<*virtual*-class>) := {o ∈ O | o ∈ <*source*-class1> ∧ o ∈ <*source*-class2>}.

The semantics of the intersection operator imply the following subset relationships, <*virtual*-class> ⊆ <*source*-class1> and <*virtual*-class> ⊆ <*source*-class2>. Furthermore, the type description of the virtual class is equal to the greatest common subtype of the two sources classes as defined in Definition 4. This is denoted by

type(<*virtual*-class>) := **type**(<*source*-class1>) ⊔ **type**(<*source*-class2>).

This implies the subtype relationships <*virtual*-class> ⪯ <*source*-class1> and <*virtual*-class> ⪯ <*source*-class2>, and finally also the subclass relationships <*virtual*-class> *is-a* <*source*-class1> and <*virtual*-class> *is-a* <*source*-class2>.

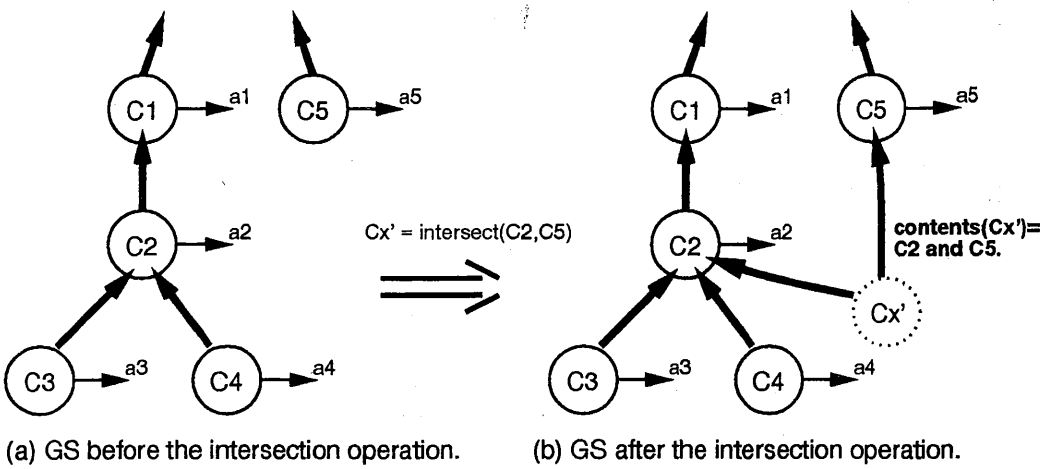


Figure 9: The Intersection Class Operator.

Example 9. *The* **intersection** *class operator is used in Figure 9 to derive the virtual class Cx' from the source classes C2 and C5, namely, the query Cx' :=* **intersect***(C2,C5). Then* **extent***(Cx') :=* **extent***(C2)* $\cap$ **extent***(C5). Hence Cx'* $\subseteq$ *C2 and Cx'* $\subseteq$ *C5. Also* **type***(Cx') :=* **type***(C2)* $\sqcup$ **type***(C5). Hence Cx'* $\preceq$ *C2 and Cx'* $\preceq$ *C5. The is-a relationships (Cx' is-a C2) and (Cx' is-a C5) are indicated in Figure 9 by the edges from Cx' to C2 and from Cx' to C5, respectively.*

4.7 The Difference Operator

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

<*virtual*-class> := **diff**(<*source*-class1>,<*source*-class2>).

Its semantics are to return a set of object instances that are members of the first but not of the second source class. More formally stated,

extent(<*virtual*-class>) := {o $\in$ O | o $\in$ <*source*-class1> $\wedge$ o $\notin$ <*source*-class2>}.

The following subset relationship holds <*virtual*-class> $\subseteq$ <*source*-class1>. Furthermore, the type description of the virtual class is equal to the type description of the first source class, i.e.,

type(<*virtual*-class>) := **type**(<*source*-class1>).

By default, <*virtual*-class> $\preceq$ <*source*-class1>. This then implies the subclass relationship <*virtual*-class> *is-a* <*source*-class1>. No subset, subtype or subclass relationships hold between the second <*source*-class2> and the <*virtual*-class>.

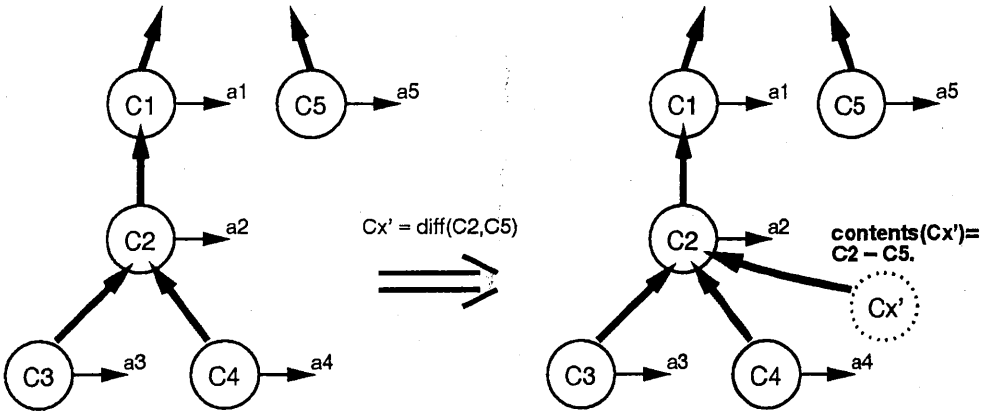


Figure 10: The Difference Class Operator.

Example 10. *An example of the* **difference** *class operator is given in Figure 10. The query "Cx' :=* **diff***(C2,C5)" is used to derive the virtual class Cx' from the source classes C2 and C5. We have* **extent***(Cx') :=* **extent***(C2) –* **extent***(C5) and Cx'* $\subseteq$ *C2. Also* **type***(Cx') :=* **type***(C2), and thus Cx'* $\preceq$ *C2. The is-a relationship (Cx' is-a C2) is indicated in the Figure 10 by the edge from Cx' to C2.*

In this section, we have shown how the virtual classes created by the schema operators are integrated with their source schema. A more thorough treatment of how to integrate them with the complete global schema rather than only the source subschema is presented in the next section.

5 CLASS INTEGRATION INTO THE GLOBAL SCHEMA

MultiView integrates all virtual classes derived using the algebra operator defined in the previous section into the global schema in order to explicitly represent the generalization relationships between virtual and base classes. Algorithms for special classification subproblems have been proposed in the literature. For instance, Schmolze and Lipkis [13] describe a classifier for 'concepts' in the KL-ONE Knowledge Representation System. Scholl et al. [14] sketch the class integration process for a selected subset of the operators of the query language COOL. In [10], we describe the integration of virtual classes derived using set operations into the underlying schema graph. In this section we sketch an overall approach for the class integration problem. A detailed treatment of this problem is, however, beyond the scope of this paper.

Class integration is concerned with finding the most appropriate location in the schema graph for a given virtual class with the term 'appropriate' meaning correct in terms of property inheritance and subset relationships between classes. We exploit the *subtype*, *subset* and *is-a* relationships between the virtual class and the classes in the global schema to solve this classification problem. The classifier determines the *is-a* relationships between the virtual class VC and all other classes in the global schema by comparing both their type descriptions and their membership predicates. This comparison then deduces the correct location of VC by placing VC between its most direct sub- and superclasses. Note that the requirements of a *valid* generalization hierarchy as defined in Section 2.3 are met by the resulting schema graph.

The algorithm for finding the correct position for the class VC in the schema G=(V,E) can be summarized as follows. First, we find all classes in G that subsume VC, i.e., they are the direct superclasses of VC defined by direct-parents(VC) := $\{C_i \mid (VC \textit{ is-a } C_i) \wedge (\nexists C_j \in V)(j \neq i)((VC \textit{ is-a } C_j) \wedge (C_j \textit{ is-a } C_i))\}$. Similarly, we find all classes in G that VC subsumes, i.e., they are the direct subclasses of VC defined by direct-children(VC) := $\{C_i \mid (C_i \textit{is-a } VC) \wedge (\nexists C_j \in V)(j \neq i)((C_i \textit{is-a } C_j) \wedge (C_j \textit{is-a } VC))\}$. VC then is placed directly below all classes in the direct-parents set and directly above all classes in the direct-children set. Edges connecting classes in the direct-children(VC) set with classes in the direct-parents(VC) set are removed, since these relationships are now represented indirectly via VC. In general, the classification problem is not decidable since it may involve the comparison of arbitrary functions and predicates. In the worst case, if some *is-a* relationship is not discovered, then this means that the virtual class is placed higher in the class hierarchy than would theoretically be possible. This would still be a correct but possibly not the most informative class arrangement.

Note that the above described algorithm is inefficient since it always searches through all classes in the schema graph. This process can be optimized by fine-tuning it for each query operator. For instance, for the **refine** operator, which produces a virtual class with a new property function distinct from all existing ones in the schema, this algorithm can be reduced to a simple O(1) algorithm requiring no search. As shown in Section 4.3, the **refined** virtual class is always a (direct or indirect) subclass of its source class. Furthermore, it cannot be placed any lower in the class hierarchy for the following reason: it has a type description distinct from all other existing types

due to the newly defined function and therefore it would be incorrect for any of the existing classes to inherit this new property. For the same reason, it cannot have any subclasses itself. We can thus conclude that the **refined** virtual class always has to be placed as direct subclass of its source class with no children of its own, i.e., direct-parents(<**refined**-*virtual*-class>) := { <*source*-class> } and direct-children(<**refined**-*virtual*-class>) := { }. This customization of the classification algorithm for particular operators allows us to limit the search to a smaller portion of the global schema based on the semantics of the operator and the position of the respective source classes.

Rather than presenting detailed classification algorithms here, we demonstrate this process on an example.

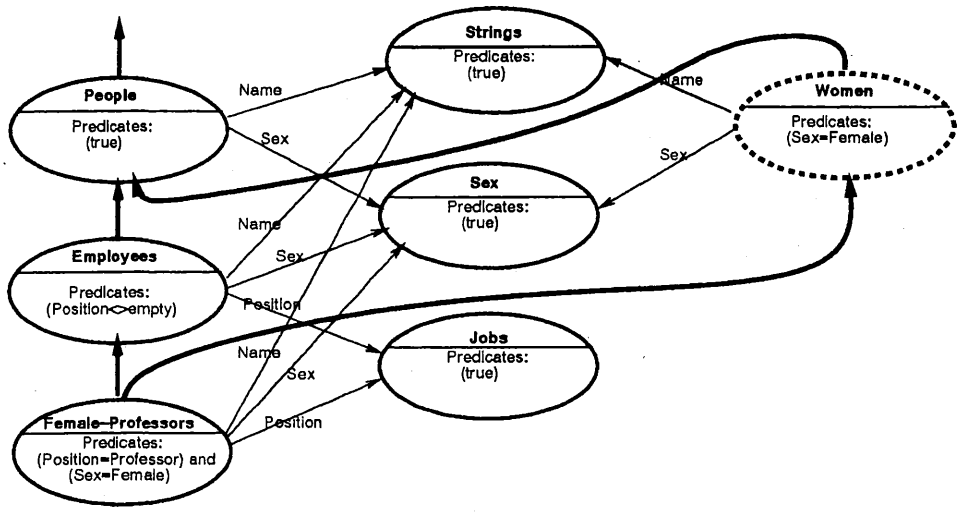


Figure 11: Integrating the Virtual Class **Women** Into the Global Schema.

Example 11. *In Figure 11, the virtual class* **Women** *is derived by the query "***Women** := **select from** (**People**) **where** *(Sex="female")". Women contains a subset of the object instances that are members of the source class* **People**, *i.e.,* **Women** $\subseteq$ **People**. *Furthermore,* **Women** *inherits the type description from its source class* **People**, *while constraining the domain of the 'Sex' property function. Hence the subtype relationship* **Women** $\preceq$ **People** *holds. From these two class relationships we can conclude that (***Women** *is-a* **People**). *We therefore insert the edge (***Women** *is-a* **People**) *into the global schema. The classification process is however not complete. Instead, we now traverse the schema graph downwards from the source class to find the most specialized classes – lowest in the is-a hierarchy – that are still is-a related with the* **Women** *class. Since the* **Employees** *class has a more refined type definition than the* **Women** *class, the type relationship (***Employees** $\preceq$ **Women**) *holds. We can however not determine any subset relationship between these two classes. Hence, neither (***Women** *is-a* **Employees**) *nor (***Employees** *is-a* **Women**) *is true. However, the type relationship (***Female-Professor** $\preceq$ **Women**) *holds, because the* **Female-Professor** *class inherits the additional property function 'Position' from the* **Employees** *class. We can also establish a subset relationship between these two classes based on their associated predicates. Namely, the predicate "(Sex=Female)" of the* **Women** *class clearly subsumes the predicate "(Sex=Female) and (Position=Professor)" of the* **Female-Professor** *class. Thus, we can infer the subset relationship (***Female-Professor** $\subseteq$ **Women**). *By Definition 3, this implies (***Female-Professor** *is-a* **Women**) *and we add an is-a edge between the two classes as depicted in Figure 11.*

In terms of the algorithm outlined above, we would have direct-parents(**Women**) := {**People**} and direct-children(**Women**) := { **Female-Professor** }. The **Women** class is indeed placed between these two classes in the global schema.

6 VIEW SCHEMA DEFINITION

In this section, we assume that the first two tasks of *MultiView*, namely, the definition of virtual classes and their integration into one underlying global schema, have been taken care of. We are now concerned with the third task of *MultiView*, namely, the definition of a view schema on top of the augmented global schema. For this we define a view definition language that can be utilized by the view definer for the specification of view schemata. In Figure 12, we present the BNF syntax of this view definition language. Note that the view definition language is concerned only with the manipulation of view classes and not with view *is-a* relationships. Rather than specifying *is-a* arcs manually, *MultiView* will automatically generate the set of view *is-a* arcs that has to be inserted in order to make the view schema *valid*. This is the topic of [12].

```
<view-definition> ::= <view-creation>; | <view-modification>;
<view-creation> ::=
  DEFINE-VIEW <view-name>
    <class-creations> | <view-schema-manipulation>
    <view-manipulation>
  END-VIEW
<view-modification> ::=
  MODIFY-VIEW <view-name>
    <class-creations> | <view-schema-manipulation>
    <view-manipulation>
  END-VIEW
<view-manipulation> ::= SAVE-VIEW; | DELETE-VIEW;
<view-schema-manipulation> ::=
  ADD-CLASS(<class-name>);
  | ADD-CLASS-DAG(<class-name>);
  | ADD-VIEW-SCHEMA(<view-name>);
  | REMOVE-CLASS(<class-name>);
  | REMOVE-CLASS-DAG(<class-name>);
  | REMOVE-VIEW-SCHEMA(<view-name>);
  | RENAME-CLASS(<old-class-name>) by (<new-class-name>);
<class-creations> ::=
  <class-name> := <class-derivation-operator>;
```

Figure 12: The BNF Syntax Of the View Definition Language.

The following operators either initiate or terminate a transaction on a view schema: `DEFINE-VIEW`, `MODIFY-VIEW`, `SAVE-VIEW`, `DELETE-VIEW` and `END-VIEW`. The `DEFINE-VIEW` command initializes a new empty view schema and assigns a unique view identifier to it. The creation of virtual classes or the modification of a view schema VS can be done only in the context of a view definition transaction of the particular view schema. Within this transaction, which is marked by a `DEFINE-VIEW` command at the beginning and the `END-VIEW` command at the end, changes can be made to this one view schema only.

The MODIFY-VIEW command is similar to DEFINE-VIEW, except it is applied to an already defined view schema rather than creating a new one. It thus prepares an existing view schema VS for modification. All operators specified within this view definition transaction, i.e., after this MODIFY-VIEW command and before the terminating END-VIEW command, will modify only VS and no other view schema. Since the existing view schema VS already has a unique identifier, no new view identifier is allocated.

The view definers concludes the view definition phase by issuing the SAVE-VIEW command. This command establishes a view table for the view schema which lists all classes that are part of this view (See Section 9). In addition, the system determines the set of view *is-a* arcs that have to be inserted into this view schema and of course also into the view table [12].

Lastly, a view definer can remove a view schema with the DELETE-VIEW command. This command not only deletes the view table and view *is-a* arcs, but it also removes all virtual classes from the global schema that were created for the definition of that view schema, whenever possible. Virtual classes can no longer be removed when they are already (directly or indirectly) utilized by other view schemata.

The view schema manipulation operators (<view-schema-manipulation> in Figure 12) modify one designated view VS by either adding or deleting view classes. They assume that a view schema VS has already been created and opened for manipulation by either a DEFINE-VIEW or a MODIFY-VIEW command. The "ADD-CLASS(<*class*-name>)" command adds a class with the name <*class*-name> in GS to the view schema VS. The "ADD-CLASS-DAG(<*class*-name>)" command adds all classes to the view schema VS that are classes in the subschema of GS rooted at the class with the name <*class*-name>. Finally, the "ADD-VIEW-SCHEMA <*view*-name>" command adds all classes of the view schema with the view identifier <*view*-name> to the current view schema VS. The three commands, REMOVE-CLASS, REMOVE-CLASS-DAG, and REMOVE-VIEW-SCHEMA, do the same as the just described operators but rather than adding they are deleting the respective classes. Lastly, the "RENAME-CLASS <*old-class*-name> by <*new-class*-name>" command renames an existing view class of the view schema VS by replacing its name <*old-class*-name> by the new name <*new-class*-name>. We assume scoping here; hence this is a local change that is only visible from within the current view schema.

Below, we demonstrate the above mentioned commands of the view definition language based on the example views shown in Figure 13.

Example 12. *In this example, we discuss the definition of the view schema VS1 in Figure 13.d on top of the global schema GS depicted in Figure 13.a. Below, we give one possible view creation script for the specification of VS1.*

View Creation Script For VS1:

```
DEFINE-VIEW VS1
   VC4 = SELECT C1 where <predicate>;
   ADD-CLASS (C1);
   ADD-CLASS (C3);
   SAVE-VIEW;
END-VIEW
```

We start the view definition transaction by issuing the DEFINE-VIEW VS1 command, which creates an empty view schema with the identifier VS1. We then define and insert the virtual class

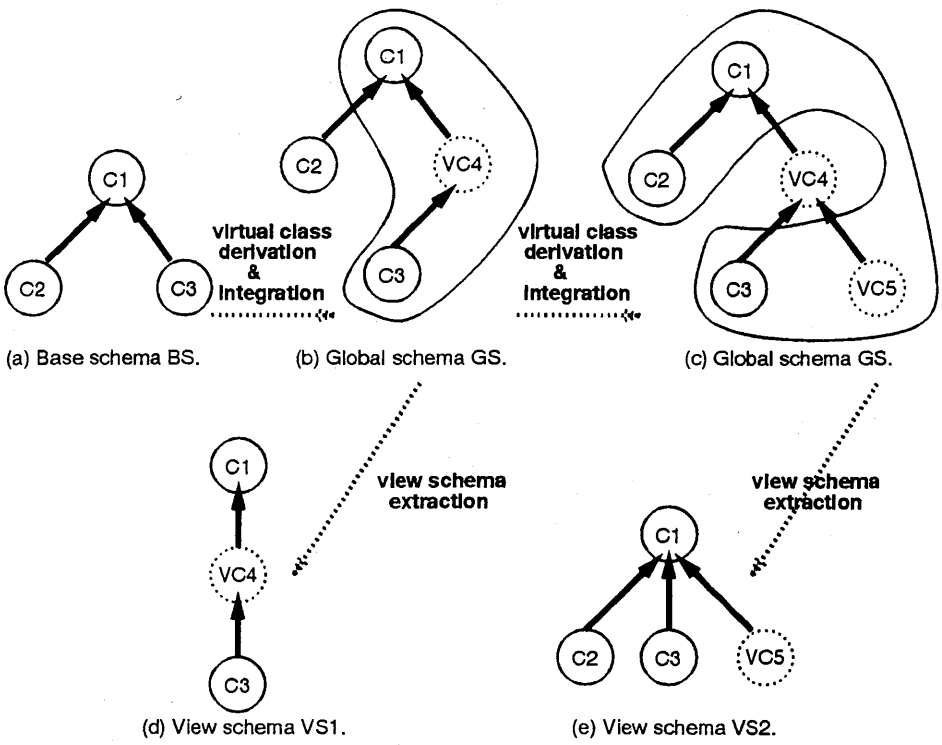


Figure 13: From Base over One Integrated Global Schema To Multiple View Schemata.

VC4 into the global schema GS. As discussed above, VC4 is also automatically added to the view schema VS1. Then the commands `ADD-CLASS(C1)` *and* `ADD-CLASS(C3)` *are issued to insert the classes C1 and C3 into VS1. VS1 now has the classes(VS1) = { VC4, C1, C3 }. Lastly, VS1 is saved with the command* `SAVE-VIEW`*. MultiView then automatically creates the view is-a arcs for VS1 [12]. The result of this view generalization hierarchy creation is shown in Figure 13.d.*

Example 13. *The second view schema VS2 in Figure 13.e is defined on top of the global schema GS depicted in Figure 13.b. A possible view creation script for VS2 is given below.*

View Creation Script For VS2:

```
DEFINE-VIEW VS2
   VC5 = SELECT VC4 where <predicate>;
   ADD-VIEW-SCHEMA (BS);
   SAVE-VIEW;
END-VIEW
```

First, the DEFINE-VIEW VS2 command creates an empty view schema with the identifier VS2. Then the virtual class VC5 is defined and integrated into the global schema GS. Then the three classes { C1, C2, C3 } are added to VS2, i.e., we now have classes(VS2) = { VC5, C1, C2, C3 }. This could be done by either issuing three `ADD-CLASS` *commands, or equivalently, we can add the base schema BS to VS2 using the command* `ADD-VIEW-SCHEMA(BS)`*, since the base schema BS is composed of exactly the three desired classes. When VS2 is saved, the is-a arcs shown in Figure 13.d are derived automatically by MultiView [12].*

Important to note here is that the restructuring of the underlying global schema GS due to the creation of VS2 did not have any effect on the existing view schema VS1. In section 8, we show that this is in general true, namely, existing view schemata remain valid after the creation of additional view schemata. We refer to this property of *MultiView* as the *view independence* property. The interested reader is referred to [12] for a more detailed discussion on the view definition language and related issues.

7 AUTOMATIC GENERATION OF A CLOSED VIEW SCHEMA

7.1 Basic Concepts

Unlike the class generalization relationships, the property decomposition relationships are not explicitly (nor independently from the actual classes) inserted into a schema. Instead, they are implicitly determined by the specification of a derived class. Recall that the definition of a virtual class automatically determines its type, i.e., all its property relationships with other classes in the view schema. For example, we create a property decomposition arc labeled p between the classes C1 and C2, a = <C1,C2,p>, by defining the class C1 to have a property function p with the $\mathbf{domain}_p$(C1):=C2. This implies that any customization of this property decomposition hierarchy by the view definer is taken care of during the class derivation phase of *MultiView*. Nonetheless, the verification of the closure criterion of a view schema can only be performed after the selection of all view classes, the third task of *MultiView*, has been completed. This is so since the closure property is a function of the complete schema (the relationships among all classes in the schema) rather than of an individual class.

As indicated in Section 2.4, instead of just checking whether a given view is closed or not, it is more useful to transform a view that is found to be not closed into a *type-closed* view schema. In this section we present an algorithm, called Closed-View Generation algorithm, that solves this problem. In particular, the algorithm automatically determines the minimal [2] set of classes by which the view schema VS has to be extended in order for the view to be *type-closed*. Before describing the algorithm, we present a theorem that describes what this minimal set is. This theorem also shows that this set is a necessary and sufficient addition to a view to assure the closure of the resulting view.

Theorem 1. (Correctness) *Given a view schema VS=(VV, VE) defined on the global schema GS=(V,E). Then* $MIN := (\bigcup_{C_i \in VV}(Uses^*(C_i))) - VV$ *is the minimal subset of classes from V that have to be added to the view VS to make it closed.*

Proof: We prove Theorem 1 in two parts. In part I, we show the sufficiency of the set MIN = $\bigcup_{C_i \in VV}(Uses^*(C_i))$ - VV for closure, namely, we show that a view VS becomes *closed* if we add the set MIN to its view classes VV. In part II, we show the necessity of MIN for closure, namely, we show that MIN is the minimal set required to make VS closed. These two facts together imply the correctness of the theorem.

[2] We assume that all classes initially selected for the view are indeed required, i.e., none of the view classes can be dropped in order to make the view *type-closed*.

Part I: Adding the set MIN = $(\bigcup_{C_i \in VV}(Uses^*(C_i))) - VV$ to the view VS will make the view *closed.*

Case I.a: Let VS=(VV,VE) be a view that is already closed. By Definition 13, VV = VV $\cup$ $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$. By subtracting the set VV from both sides of the equation, we derive that $\bigcup_{C_i \in VV}(Uses^*(C_i)) - VV = \emptyset$. This implies MIN = $(\bigcup_{C_i \in VV}(Uses^*(C_i))) - VV = \emptyset$. Since the view VS is assumed to be closed, no classes need to be added to the view, i.e., adding the set MIN = $\emptyset$ trivially makes the view closed.

Case I.b: Let VS=(VV,VE) be a view that is not closed. Then create a new view VS'=(VV',VE') with VV' the set of classes created by adding MIN to VV, i.e., VV' := VV $\cup$ MIN. Then VV' = VV $\cup$ MIN = VV $\cup$ $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$ - VV) = VV $\cup$ $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$. We now need to show that VS' is closed. By Definition 13, VS' is closed if and only if VV' = VV' $\cup$ $(\bigcup_{C_i \in VV'}(Uses^*(C_i)))$. We prove this as follows:

$\bigcup_{C_i \in VV'}(Uses^*(C_i))$

$= \bigcup_{C_i \in (VV \cup \bigcup_{C_k \in VV}(Uses^*(C_k)))} (Uses^*(C_i))$

$= \bigcup_{(C_i \in VV) \vee (C_i \in \bigcup_{C_k \in VV}(Uses^*(C_k)))} (Uses^*(C_i))$

$= \bigcup_{C_i \in VV}(Uses^*(C_i)) \cup \bigcup_{C_i \in (\bigcup_{C_k \in VV}(Uses^*(C_k)))}(Uses^*(C_i))$

$= \bigcup_{C_i \in VV}(Uses^*(C_i))$

$\subseteq$ VV $\cup$ $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$

= VV'.

Finally, $\bigcup_{C_i \in VV'}(Uses^*(C_i)) \subseteq$ VV' implies VV' = VV' $\cup$ $(\bigcup_{C_i \in VV'}(Uses^*(C_i)))$. We thus have shown that the addition of the set MIN to the non-closed view VS creates the closed view VS'. **q.e.d.**

Part II: The set MIN = $(\bigcup_{C_i \in VV}(Uses^*(C_i))) - VV$ is the *minimal* set of classes that has to be added to a view VS to make it closed.

Case II.a: Let VS=(VV,VE) be a view that is already closed. Then by part I.a, MIN = $\emptyset$. The empty set is obviously the equal to the smallest possible set of classes that has to be added to make the view closed.

Case II.b: Part II follows directly from Definition 13 for a view VS=(VV,VE) that is not closed. By Definition 13, all classes that are in the transitive closure of the *Uses** relationship of VS, $\bigcup_{C_i \in VV}(Uses^*(C_i))$, must also be part of VS in order for VS to be closed. On the other hand, classes that are already part of VS do not have to be added again. Therefore, all classes in $\bigcup_{C_i \in VV}(Uses^*(C_i)) - VV$ must be added to VV. Note that $\bigcup_{C_i \in VV}(Uses^*(C_i)) - VV$ is equal to MIN. **q.e.d.**

7.2 Closed-View Generation: Algorithm and Examples

The Closed-View Generation algorithm (CVG) is given in Figure 14. CVG determines whether a given view is closed or not. If the view is not closed then the algorithm automatically determines the minimal set of classes by which the view schema VS has to be extended in order for the view to be *type-closed*. This is done by recursively exploring the *Uses* relationships of classes. Note that the *uses* relationships of a class are independent from which other class of the schema we have reached the current class. Therefore, we only need to calculate the simple form of a transitive closure of the *uses* relationship. This observation reduces the complexity of the algorithm considerably, namely, from cube to linear complexity. Once we process a class C_i by checking its *Uses* relationships, it need not be processed anymore. In order to avoid unnecessary repetitive processing, the algorithm maintains a list of all classes that do not have to be checked anymore, called CVG-done. In addition, it maintains a list of all classes reached via the *Uses* relationship that still have to be processed, called CVG-tmp.

The algorithm proceeds as follows. While there are any classes left to be processed in the CVG-tmp set, the algorithm picks one of them, say Ci. The processing of the class Ci entails the following. If Ci is not in the view, then the view is not closed and the flag *Closed* is set to false. The algorithm also adds Ci to the CVG-done set which serves the following two purposes: first, it assures that Ci will not be processed again, and second, it collects all classes that need to be added to the view to make it closed. Next, the algorithm checks for all classes Ck in the *Uses(Ci)* set, whether they have to be processed for closure. They do not have to be processed for closure, if either they already have been processed (i.e., are in CVG-done) or if they are guaranteed to be processed at some later time (i.e., are in VV or in CVG-tmp). If they still have to be processed then they are added to CVG-tmp. The algorithm terminates when all classes reachable from the view classes of the view VS have been processed, i.e., when CVG-tmp is empty. If the view is *closed*, then the algorithm returns the flag "Closed=true" and the set "CVG-done=$\emptyset$". If the view is not *closed*, then the algorithm returns the flag "Closed=false" and the set "CVG-done$\neq \emptyset$". The latter contains all classes that have to be added to the view schema in order to make it *closed*, i.e., CVG-done = MIN with MIN defined in Theorem 1. Below, we discuss examples of applying the CVG algorithm to the view schemata shown in Figure 15.

Example 14. *The view VS1 in Figure 15.b is defined on top of the global schema GS depicted in Figure 15.a. The CVG algorithm first initializes CVG-done := $\emptyset$, CVG-tmp := $\{C1, C3\}$, and Closed := true. For the first iteration of the while-loop, the iteration variable Ci is equal to C1. The first if-statement evaluates to false, since (C1 $\in$ VV), and thus is skipped. Uses(C1) := $\{C3\}$ due the 'state-transition' property defined for C1. Therefore, the for-loop is executed but once with the iteration variable Ck := C3. The if-statement in the body of the for-loop evaluates to false, since (C3 $\in$ VV). For the second iteration of the while-loop, the iteration variable Ci is set to C3. The first if-statement is again skipped. Uses(C3) := $\{C1\}$ due to the two properties 'prv-state' and 'nxt-states' defined for C3. The for-loop is executed with Ck := C1. The if-statement in the loop body again evaluates to false, since (C1 $\in$ VV). CVG-tmp is now empty, and therefore the algorithm terminates. CVG returns the parameters (Closed=true) and (CVG-done=$\emptyset$). VS1 has been shown to be closed.*

Data Structures and Variables:

Set of classes: CVG-tmp, CVG-done;
Classes: Ci, Ck;
Boolean flag: Closed;

Procedures and Functions:

get-next(set-of-classes) → class;
not-element(class,set-of-classes) → boolean;
add-to-set(class,set-of-classes);

Input:

Global Schema *GS = (V,E)* and View Schema *VS=(VV, VE)*

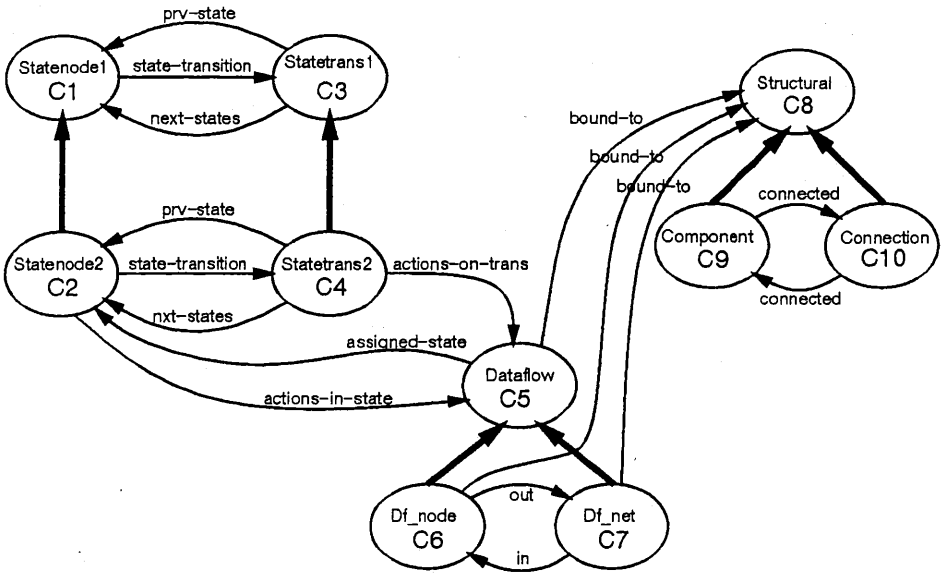
Output:

The flag Closed indicates whether the view is closed.
The set of classes CVG-done contains all missing classes required to make the view closed.

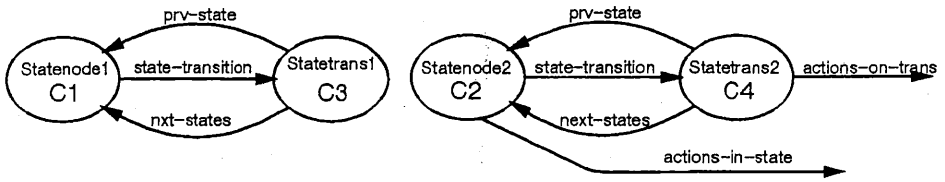
Algorithm CVG: The Closed-View Generation Algorithm.

```
algorithm CVG(GS, VS) return (CVG-done: set-of-classes,Closed:boolean-flag) is
    CVG-done := ∅;
    CVG-tmp := VV;
    Closed := true;
    while (Ci := get-next(CVG-tmp)) do
        if (not-element(Ci,VV)) then
            Closed := false;
            add-to-set(Ci,CVG-done);
        endif;
        for all Ck in Uses(Ci) do
            if (not-element(Ck,CVG-done) and not-element(Ck,CVG-tmp) and not-element(Ck,VV)) then
                add-to-set(Ck,CVG-tmp);
            endif;
        endfor;
    endwhile
    return (CVG-done,Closed);
end algorithm;
```

Figure 14: The Closed-View Generation Algorithm.

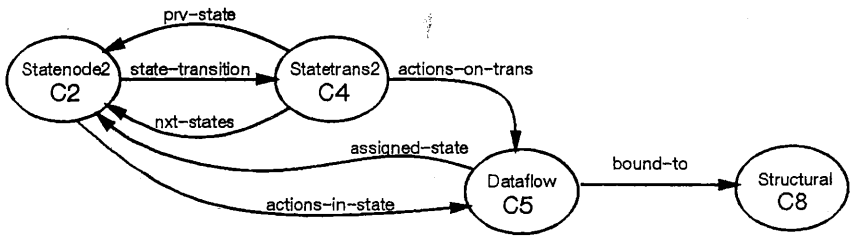


a. The global schema GS.



b. The view VS1 is closed.

c. The view VS2 is not closed.



d. The view VS2' is closed (derived from VS2 by the CVG algorithm).

Figure 15: Examples of Applying the Closed-View Generation Algorithm.

Example 15. *In this example, we describe how CVG is applied to the view VS2 depicted in Figure 15.c with VS2 defined on GS shown in Figure 15.a. CVG initializes the variables as follows: CVG-tmp := $\emptyset$, CVG-tmp := $\{C2, C4\}$, and Closed := true. For the first iteration of the while-loop, the iteration variable Ci is set equal to C2. The first if-statement evaluates to false and is skipped. Since Uses(C2) := $\{C4, C5\}$, the for-loop has two iterations. For the iteration with Ck := C4, the if-statement is skipped. For the second iteration with Ck := C5, the if-statement evaluates to true and C5 is added to CVG-tmp for further processing. For the second iteration of the while-loop with the iteration variable Ci:=C4, the first if-statement is again skipped. The for-loop has two iterations since Uses(C4) := $\{C2, C5\}$. For both iterations, the if-statement is skipped. For the third iteration of the while-loop with the iteration variable Ci:=C5, the first if-statement evaluates to true since C5 $\notin$ VV. Therefore, C5 is added to CVG-done and the flag Closed is set to false. Since Uses(C5) := $\{C2, C8\}$, the for-loop has two iterations. For the second iteration of the for-loop with Ck:=C8, the if-statement evaluates to true and C8 is added to CVG-tmp. For the fourth and last iteration of the while-loop with the iteration variable Ci:=C8, the first if-statement evaluates to true and C8 is added to CVG-done. Since Uses(C8) := {}, the for-loop is not executed. CVG-tmp is now empty and the CVG algorithm terminates with (Closed=false) and (CVG-done=$\{C5, C8\}$). CVG thus has shown that the view VS2 is not closed. In addition, the algorithm has determined the set of classes that have to be added make a complete view out of VS2, namely, the set CVG-done. The resulting augmented and thus closed view VS2' is shown in Figure 15.d.*

7.3 Correctness and Complexity of the Closed-View Generation Algorithm

Theorem 2. (Correctness) *Given a view schema VS=(VV, VE) defined on the global schema GS=(V,E), then the closed-view generation algorithm CVG in Figure 14 correctly generates a closed view VS'. In particular, CVG returns the value Closed=true if the view schema VS is closed, and the value Closed=false, otherwise. If the view VS is not closed, then CVG also generates the minimal set of classes that have to be added to VS to make it closed, namely, CVG-done =* $(\bigcup_{C_i \in VV}(Uses^*(C_i))) - VV$.

Proof: We prove Theorem 2 in two parts. In the first part, we show that the algorithm returns the correct value for the *Closed* flag. In the second part, we show that the CVG-done set returned by the algorithm is equal to $(\bigcup_{C_i \in VV}(Uses^*(C_i))) - VV$.

Part I: (Closed=true) $\Longleftrightarrow$ (VS is closed).

In part I.a, we show "(Closed=true) $\Longrightarrow$ (VS is closed)". In part I.b, we show "(VS is closed) $\Longrightarrow$ (Closed=true)". These together imply the desired equivalence of part I.

Part I.a: (Closed=true) $\Longrightarrow$ (VS is closed).

Assume that the algorithm CVG returned the flag Closed=true. This means that the first if-statement with the condition ($Ci \notin VV$) never evaluated to true. This implies that CVG did *not* find a single class Ci in the transitive closure of the *uses* relationship of VV (or $Ci \in VV \cup \bigcup_{C_i \in VV}(Uses^*(C_i))$) for which the following holds: $Ci \notin VV$. We can therefore conclude that for all classes Ci, $Ci \in VV \cup \bigcup_{C_i \in VV}(Uses^*(C_i))$ also implies $Ci \in$ VV. Hence, $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$

$\cup$ $VV \subseteq VV$. This implies the relationship $VV = (\bigcup_{C_i \in VV}(Uses^*(C_i))) \cup VV$. By Definition 13, the view VS is closed.

q.e.d.

Part I.b: (VS is closed) $\Longrightarrow$ (Closed=true).

Assume that the view VS is closed. Then by Definition 13, the relationship $\bigcup_{C_i \in VV}(Uses^*(C_i)) \subseteq VV$ holds. This means that there is *not* a single class Ci in the transitive closure of the *uses* relationship of VV that is not also in VV, i.e., ($\nexists\, Ci$ in V) $((Ci \notin VV) \wedge (Ci \in \bigcup_{C_i \in VV}(Uses^*(C_i)))$. Therefore, for all classes Ci processed by the algorithm CVG the condition "$Ci \notin VV$" of the first if-statement will always evaluate to false. Since the body of this first if-statement is never executed, the variable *Closed* is never modified and the initial value *Closed=true* remains.

q.e.d.

Part II: The algorithm generates the set CVG-done = $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$ - VV.

The while-loop recursively traverses the transitive closure of the *uses* relationship of all view classes of VS. Initially it starts with all classes in VS, since CVG-tmp is initialized to VV. Later the for-loop recursively adds all classes that can be reached from a class in CVG-tmp via the *uses* relationship. Hence, over the duration of the CVG execution, the while-loop will process all classes in (VV $\cup$ $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$) at least once. The algorithm adds all classes in the above set to the CVG-done set for which the condition "$Ci \notin VV$" of the first if-statement evaluates to true. Therefore, CVG-done will be equal to the set $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$ - VV. By Theorem 1, the set CVG-done thus corresponds to the minimal set of classes that has to be added to the non-closed view VS to create the closed view VS'.

q.e.d.

Theorem 3. (Complexity) *Given a view schema VS=(VV, VE) defined on the global schema GS=(V,E) with PGS=(V,A,L) the matching property decomposition hierarchy of GS as defined in Definition 7. The complexity of the closed-view generation algorithm CVG for the view VS is equal to O(|A|) with |A| the number of property decomposition arcs in PGS.*

Proof: We prove Theorem 3 in three parts.

Part I: First, we show that all functions used by CVG have constant complexity.

We assume that each class in GS has a unique index in the range from 1 to $|GS|$. Then we can implement the set-of-classes data structure by a bit-vector of length $|GS|$ as follows: the i-th bit is true if the class C_i is in the set, and false, otherwise. To check whether a class C_i is or is not an element in a set using the *not-element()* function is done by checking the value of the corresponding bit. This takes constant time. To add or to remove an element using the functions *add-to-set*() and *get-next*(), respectively, corresponds to flipping a bit. Both functions thus take constant time. CVG-done and VV are assumed to be implemented using this scheme.

Assume the variable CVG-tmp is implemented by both a linked list and a bit-vector representation. The linked list representation is used to get the next element in the list in constant time by the *get-next*() function, while the bit-vector representation is used by the *not-element*()

function to assure that the element is not already present in the list in constant time. The *add-to-set(Ck,CVG-tmp)* function flips the corresponding bit for Ck in the vector representation to true and it also appends Ck to the matching linked list representation of CVG-tmp. Similarly, the *get-next()* function first removes the next element Ci from the linked list representation of CVG-tmp and then updates the CVG-tmp's bit-vector representation of Ci by setting the appropriate bit to false. These functions can each be done in constant time. **q.e.d.**

Part II: Next, we show that each class of GS will be placed at most once into the CVG-tmp set.

When a class Ci is processed using the while-loop, it is first removed from CVG-tmp using the function *get-next()*. It is then placed into CVG-done using the first if-statement (assuming that it is not an element of VV). Once a class Ci is in the CVG-done set or in VV, Ci will never be placed back into CVG-tmp for the following reason. The second if-statement is the only statement that adds elements to the CVG-tmp set, and the condition of this if-statement, which is "(*not-element*(Ck,CVG-done)" and *not-element*(Ck,VV)) and ... ", assures that a class Ci with the above characteristics is not placed back into CVG-tmp. **q.e.d.**

Part III: Lastly, an analysis of the overall algorithm is conducted using part I and II.

- By part I shown above, all functions used by the CVG algorithm have constant complexity. Therefore, the two if-statements can be executed in constant time each.
- By part II shown above, we know that each node of the global schema GS is placed at most once in the CVG-tmp set. This implies that the while-loop is executed at most once for each node in GS, i.e., there are in the worst case $|GS|$ iterations.
- For each class Ci in the CVG-tmp list (i.e., for each iteration of the while-loop), the for-loop has exactly one iteration for each class Ck in the $Uses(Ci)$ set. $|Uses(Ci)|$, which denotes the number of distinct classes with which the class Ci in GS maintains direct property decomposition relationships, is smaller than or equal to the number of outgoing property decomposition arcs for the class Ci, denoted by $\#arcs(Ci)$. This is true since there may be two properties arcs labeled by the property names p1 and p2 defined for Ci with the same domain class Cj, e.g., a1 = <Ci,Cj,p1> and a2 = <Ci,Cj,p2>. And, for each class $Cj \in |Uses(Ci)|$, there must be at least one arc ak with the domain class Cj, i.e., ak = <Ci,Cj,pk> for some label pk.

The overall complexity of the closed-view generation algorithm CVG can now be computed as follows: complexity(CVG) $\leq O(\sum_{Ci \in V}(|Uses(Ci)|)) \leq O(\sum_{Ci \in V}(\#\text{arcs(Ci)})) = O(|A|)$. **q.e.d.**

Theorem 4. (Complexity) *Given a view schema VS=(VV,VE) defined on the global schema GS=(V,E) with PVS=(VV,VA,VL) the matching property decomposition hierarchy of VS as defined in Definition 7. If the view schema VS is closed, then the closed-view generation algorithm CVG has the complexity of O(|VA|) with |VA| the number of property decomposition arcs in PVS.*

Proof: Assume that the view schema *VS* is *closed.* By Definition 13, a *closed* view defines all classes that it uses. Therefore, the test of the second if-statement "*not-element*(C,VV)" will always evaluate to false. Therefore, no classes are added to the CVG-tmp set besides the initial set VV, and the while-loop has exactly $|VV|$ iterations. Similarly, by Definition 13, the *Uses* set of a class in *VS* consists only of classes that are defined in *VS.* Therefore, the size of a *Uses* set for a view class of a closed view is equal to all outgoing arcs of the class; with these arcs being contained in the property decomposition hierarchy of VS. The for-loop for a class Ci has at most $\#arcs(Ci)$ iterations. We thus have complexity(CVG) $\leq$ O($\sum_{Ci \in VV}$(#arcs(Ci))) = $O(|VA|)$. **q.e.d.**

Since the chosen set representation used in the algorithm is a bit-vector of length $|GS|$, the initialization of these vectors has a complexity of $|GS|$. Therefore, the complexity of the CVG algorithm including initialization would be complexity(CVG) = O(min($|GS|$, $|VA|$)).

8 THE VIEW INDEPENDENCE OF *MultiView*

8.1 The View Independence Concept

The concept of *data independence* developed for the relational model is defined as the "immunity of applications to change in storage structure and access technique" [3]. This is achieved by separating the interface to the database (the conceptual data model) from the actual implementation (the physical data model). *Physical data independence* addresses the independence of users and user programs from the physical structure of the stored data, while *logical data independence* addresses the independence of users and user programs from the logical structure of the data. A system provides *physical data independence* by supporting an implementation-independent interface (a logical data schema) that the users can utilize in place of operating directly on the underlying storage structure and access paths. A system provides *logical data independence* by supporting a view definition mechanism that lets the users define their own view schema on top of the common logical schema. The concept of *data independence* thus addresses the immunity of users from changes of the underlying data model. It does, however, not preclude from having to update the specification of possibly all existing view schemata when the underlying data model is extended and/or reorganized.

The *MultiView* methodology is based on the actual restructuring of the global schema for generating new view schemata. Therefore, we introduce the concept of *view independence* to be the immunity of view schema definition and semantics to changes of the underlying global schema.

Definition 14. *A database system provides* **view independence** *if the specification and the semantics of existing view schemata are not affected by the definition of new view schemata.*

This concept of view independence is a necessary and important requirement for object-oriented database systems, since the underlying base schema is restructured with possibly each new view schema definition. A redefinition of all existing view schemata for whenever a new view schema is introduced would be an unacceptable overhead. The concept of *view independence* does not have any significance in relational database theory where the definition of new views has no affect on the underlying base schema. In fact, the relational model is by default *view independent.*

For the *MultiView* methodology to be view independent would mean that the integration of new virtual classes into the global schema does not require the redefinition of the existing view schemata nor does it modify their semantics. The latter means that in spite of the restructuring of the global schema the following must hold: (1) the view classes of a view defined on the global schema do not change their type description nor their set membership and (2) the *is-a* relationships between the view classes of a given view are preserved. A more precise definition of the view independence concept for *MultiView* is given below.

Definition 15. *Let G* be the set of all schemata, C* the set of all classes, O the set of all object instances, and P the set of all properties. Let GS=(V,E) be a global schema and VS=(VV,VE) a view schema defined on GS. Let VS* be the set of all view schemata defined on GS. Let* Π*: G* $\rightarrow$ G* be a function that applies a class derivation operator to GS and then restructures GS by integrating the resulting virtual class into GS*[3]*. Let GS' = (V',E') be the global schema GS and VS'=(VV',VE') the view schema VS after the the integration of virtual classes into GS using the function* Π*, i.e., GS' :=* Π*(GS) and VS' :=* Π*(VS).*

(a) *The view classes VV of the view schema VS are defined to be* **preserved** *through the application of the function* Π *to GS iff the following holds:*

- $\exists$ *a one-to-one mapping m:* $C^* \rightarrow C^*$*, such that,*
  $(\forall C_i \in C^*)((C_i \in VV) \Longrightarrow (\exists! C_i' \in VV')(C_i'=m(C_i))$*, and vice versa,*
  $(\forall C_i' \in C^*)((C_i' \in VV') \Longrightarrow (\exists! C_i \in VV)(C_i=m^{-1}(C_i'))$[4]*.*
- $(\forall C_i \in VV)\ (\forall o \in O)\ ((o \in C_i)$ *in* $VV \Longleftrightarrow (o \in m(C_i))$ *in* $VV')$*.*
- $(\forall p \in P)(\forall C_i \in VV)\ ((p \in$ **properties**(C_i) *in VS)* $\Longleftrightarrow (p \in$ **properties**$(m(C_i))$ *in VS')).*

(b) *The view is-a relationships VE among the view classes VV are defined to be* **preserved** *through the application of the function* Π *to GS iff the following holds:*

- $(\forall C_i, C_j \in VV)\ (((C_i$ *is-a* $* C_j) \in VE) \Longleftrightarrow ((m(C_i)$ *is-a* $* m(C_j)) \in VE'))$*.*

with the mapping m as defined in **(a)**.

(c) *The view schema VS is defined to be* **preserved** *through the restructuring of GS using the function* Π *iff the type description and set membership of the view classes VV are* **preserved** *as defined in* **(a)** *and the view is-a relationships VE among the view classes VV are* **preserved** *as defined in* **(b)**.

(d) *The MultiView methodology is defined to be* **view independent** *if all view schemata in VS* are* **preserved** *as defined in* **(c)**.

---

[3]For this report, we assume that the function Π corresponds to the relational algebra operators and the integration algorithm presented earlier in this paper. Without loss of generality, other operators or integration algorithms could be substituted.

[4]This mapping m is simply the equality operator on the class identifiers, since each class has a unique identifier.

8.2 Proving *MultiView* View Independent

Below, we prove the *view independence* property of *MultiView* in two steps. First, we show that view classes (Definition 15.a) and then that the *is-a* relationships among view classes (Definition 15.b) are not affected by the restructuring of the global schema[5].

8.2.1 Preservation of View Classes

As specified in Definition 15.a, a view class needs to preserve both its type description and its set membership through possible restructuring of the underlying global schema in order for *MultiView* to be view independent. There are two design choices for determining the type description of a view class:

- First, we can determine the type of a view class based on the type descriptions of classes visible in the view schema.
- Second, we can determine the type of a view class based on the complete underlying global schema, which may be partially invisible in the view.

The former offers the advantage that a property is only visible in a view schema, if the class that defines this property is also visible. This would thus guarantee a unique location (class) in the view hierarchy for the definition of any property that is visible in the view schema. Unfortunately, this approach violates the view independence property as we will show below.

Example 16. *In this example we demonstrate the two approaches for type determination of a view class based on Figure 16. In this figure we use the following graphical convention: For a given class, we depict attributes that are directly defined for that class (for that view schema) by an arrow. Attributes that are inherited (for that view schema) are depicted by a dotted arrow.*

Figures 16.a and 16.b show the global schema before and after the derivation and integration of the two virtual classes C0' and C1'. Figures 16.c and 16.d demonstrate the type determination of a view class using approach 1. Approach 1 determines types based on the classes visible in the view schema. For instance, attribute a1 is defined in class C1 in the global schema, and since class C1 is not visible in the view schema VS1, the attribute a1 is also not visible in VS1. Therefore, the class C2 in Figure 16.c does not have the attribute a1 defined. As shown in Figure 16.d, both classes C0 and C2 change their types due to the restructuring of the base schema. Class C0, for instance, has the type **properties***(C0)= { a0,ax } before and the type* **properties***(C0)= { a0 } after the global schema restructuring. Consequently, approach 1 does not guarantee view independence.*

---

[5]Note that we define the **type** of a class to be the union of its defined and inherited property functions. Turning a defined property into an inherited property (as done for instance in Figure 16.e for property a1) is not considered to be a change of the class type. We define the set membership of a class, denoted by *extent*(C) = $\{o \mid o \in C\}$, to be the union of its direct and indirect instances; and it is this combined membership of direct and indirect members that we require to stay invariant with view creation. The *direct* membership content of a class C, defined by *direct-extent*(C) = *extent*(C) - $\bigcup_{i=1}^{k}$ *extent*(C_i) with C_i (with i = 1, ..., k) the direct subclasses of C, or the indirect membership of C, defined by *indirect-extent*(C) = *extent*(C) - *direct-extent*(C), may of course be modified by the creation of a new view. For instance, the creation of additional subclasses of a class C may diminish C's direct membership and increase C's indirect membership.

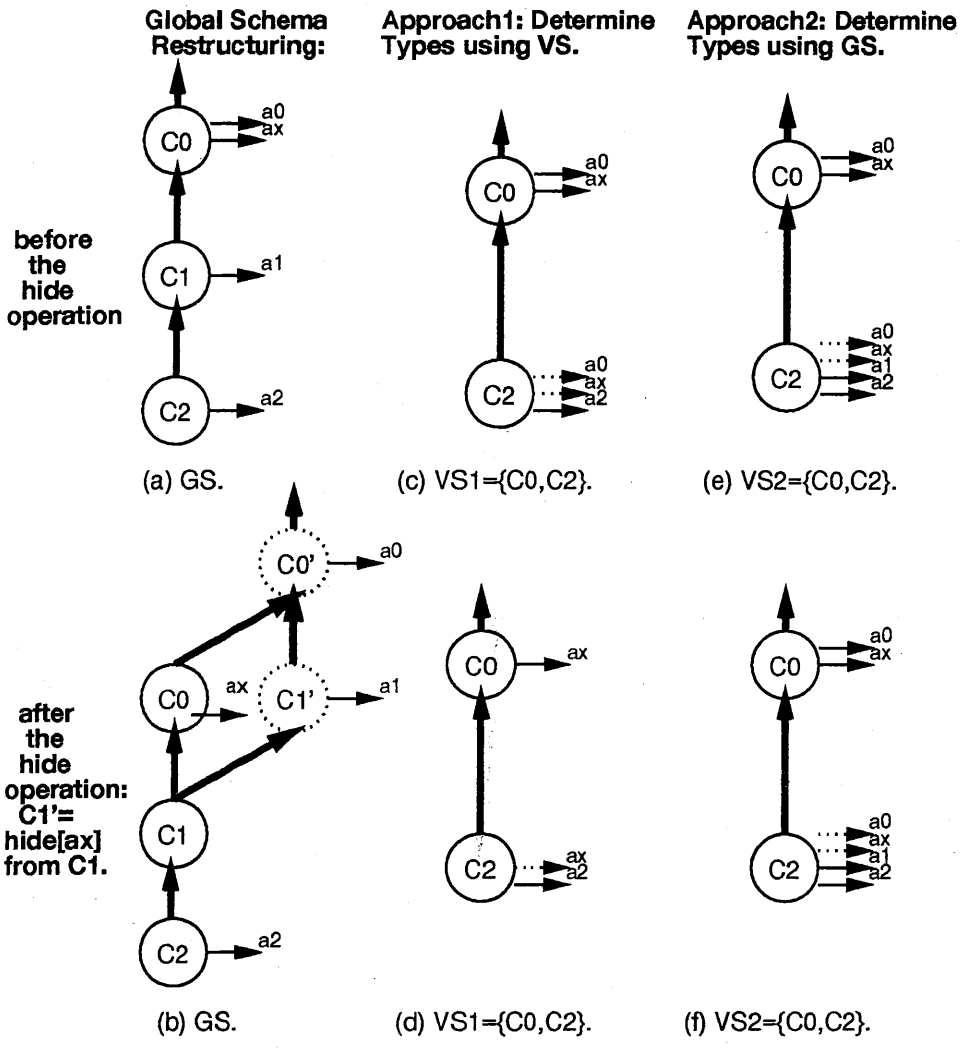


Figure 16: Two Approaches for Type Determination of a View Class

Figures 16.e and 16.f demonstrate the second approach that determines types based on the class hierarchy in the global schema. Hence, the class C2 in Figure 16.e has defined the attribute a1, even though in the global schema attribute a1 is defined in a class that is not visible in the view schema VS2. Note that methods inherited from classes invisible in the view schema are displayed as defined methods for the first class that inherits them. For example, attribute a1 is displayed as being defined rather than inherited for class C2 in VS2. Figure 16.f shows that the view schema is not affected by the restructuring of the global schema, i.e., all classes maintain their original types. Approach 2 thus guarantees view independence for this example.

Theorem 5. *Approach 1 for type determination of view classes does not preserve the view independence property.*

Proof (by Counterexample): Theorem 5 can be shown by giving one example for when the view independence property is indeed violated by approach 1. Example 16, in particular, Figures 16.c and 16.d, are such a counterexample. Namely, the integration of the virtual class *C1'* into GS affected the type description of the existing view classes in the view VS1. **q.e.d.**

We have thus shown that approach 1 may lead to type description changes of view classes when *is-a* restructuring the global schema for a new view. This violates the view independence property and thus is clearly unacceptable. This is one reason for why we have adopted the second approach in *MultiView*. Approach 2, namely, the determination of the type description of view classes based on the global schema rather than on the view schema, poses the obvious constraint on the global schema to maintain the type description of all classes during schema restructuring. We have thus reduced the problem of type preservation from the view schemata to the global schema.

Theorem 6. *Let VS* be the set of all view schemata defined on the global schema GS. The MultiView methodology* **preserves** *the view classes of all view schemata in VS* through the restructuring of GS using the function* Π *with the term* **preserves** *defined in Definition 15.a.*

Proof: (Intuitive)

We now give the intuitive reasoning for Theorem 6. As explained earlier, mapping m given in Definition 15 corresponds to the equality operator on the class identifiers. Class identifiers are unique and the set of view classes is always a subset of the set of global classes. Hence, the mapping m is a one-to-one function (Part I of Definition 15.a).

The class derivation of a virtual class creates a new class with possibly a new type description and a new content; it does obviously not modify the semantics of existing classes. Hence, we are only concerned with the integration part and not the class derivation part of the function Π. As discussed above, *MultiView* determines the type description and the set membership of a view class directly from the global schema. Therefore, we can reduce the problem of view class preservation from the view schemata to the global schema. We thus only need to show that all classes *Ci* of GS are preserved when integrating new virtual classes into GS.

Recall that the integration algorithm of virtual classes (even if fine-tuned for particular query operators) follows the principle explained in Section 5. Namely, the virtual class VC is inserted below its direct superclasses called direct-parents(VC) and above its direct subclasses in

GS called direct-children(VC) with ($\forall$ $C_i \in$ direct-parents(VC)) (VC *is-a* C_i) and ($\forall$ $C_j \in$ direct-children(VC)) (C_j *is-a* VC). Due to (1) VC being *is-a* related to both sets of classes and (2) the transitivity of the *is-a* relationship, we can deduce that classes in these sets were *is-a* related to one another before the insertion of VC. In particular, ($\forall C_i \in$ direct-parents(VC)) ($\forall C_j \in$ direct-children(VC)) (C_j *is-a* * C_i). Clearly, the insertion of VC does not modify the content of any of the existing classes, i.e., part II of Definition 15.a holds. The insertion of VC also does not modify the type description of any of the existing classes. All classes that are made subclasses of VC in the modified GS are also subtypes of VC; i.e., they will not inherit any new property functions and their types will be preserved. This shows part III of Definition 15.a. **q.e.d.**

8.2.2 Preservation of View *is-a* Relationships

Theorem 7. *Let GS be a global schema and VS* be the set of all view schemata defined on GS. The MultiView methodology* **preserves** *the view is-a relationships among the view classes of each view in VS* through the restructuring of GS using the function* Π *with the term* **preserves** *as defined in Definition 15.b.*

Proof: (Intuitive) As specified in Definition 11, we derive the *is-a* relationships of view classes directly from the *is-a* relationship found in the underlying global schema, i.e., ($\forall$ C_i,$C_j \in$ VV) ((C_i *is-a* * $C_j \in$ GS) $\Longleftrightarrow$ (C_i *is-a* * $C_j \in$ VS)). Consequently, if we can show that the relative *is-a* relationships are maintained for all pairs of classes in GS, then we have also shown that they are maintained for all pairs of classes in VS.

The integration algorithm of virtual classes (even if fine-tuned for the particular query operators) follows the general approach explained in Section 5. Namely, the virtual class VC is inserted below its direct superclasses and above its direct subclasses in GS called direct-parents(VC) and direct-children(VC), respectively. As shown in Theorem 6, we can deduce that these sets of classes had to be *is-a* related before the insertion of VC. In particular, ($\forall C_i \in$ direct-parents(VC)) ($\forall C_j \in$ direct-children(VC)) (C_j *is-a* * C_i). Therefore, the insertion of VC does not add any new *is-a* relationships. It is obvious that the insertion of the virtual class VC does not remove any *is-a* relationships. We have thus shown the preservation of all *is-a* relationships in GS. **q.e.d.**

Theorem 8. *The MultiView methodology is* **view independent.**

Proof: Theorems 6 and 7 show respectively that the *MultiView* approach **preserves** the view classes and the view *is-a* relationships of all view schemata defined on a global schema GS through the restructuring of GS. By Definition 15 these two theorems together prove the view independence of *MultiView.* **q.e.d.**

9 REALIZATION OF *MultiView*

While steps one and two of MultiView (Figures 4.b and 4.c) are real in as much as they actually modify the underlying global schema, steps three and four (Figures 4.d and 4.e) are virtual since they leave the underlying global schema intact (Section 3). This is supported by maintaining information on view schemata, such as, their view classes and their view *is-a* relationships, in separate view object tables as described in this section. There are two equivalent ways in which to maintain the information about multiple view schemata: a centralized or a distributed approach. An example of these two approaches is given in Figure 17.b and 17.c, respectively, while Figure 17.a depicts the schemata in the graphical form used throughout the paper.

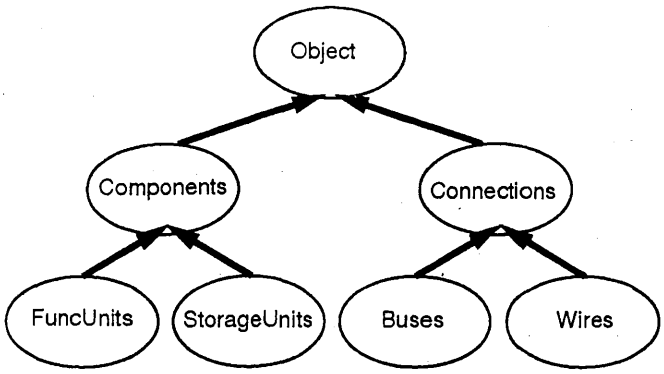
The centralized approach (Figure 17.b) maintains one *view schema table* for each view schema (similar to a data dictionary in the relational model). The *base schema table* and the *global schema table* are two special schema tables kept for the base and the global schema, respectively. Such a view schema table contains the following information: a list of class names of classes that belong to the schema, their internal class identifiers, and their direct sub- and superclasses. Figure 17.b, for instance, shows how the schemata of Figure 17.a are captured by the centralized approach. There are three view (schema) tables, one for the global schema and one for each view schema. The view table for View1, for instance, has been assigned the view identifier $< VS1 >$. The table enumerates all classes that belong to View1, which are Objects, Components, FuncUnits, and StorageUnits, as well as their direct sub- and superclasses in the view.

The second approach distributes the above described schema information across the classes of the global schema. More precisely, each class of the global schema would be extended with a list of view identifiers of the view schemata to which that class belongs. For each such view identifier, it would enumerate the direct sub- and superclasses visible within the respective view. Figure 17.c, for instance, shows how the schemata of Figure 17.a are captured by the distributed approach. In Figure 17.c, for instance, the Components class belongs to both views View1 and View2. Hence, the class definition of the Components class lists their view identifiers $<VS1>$ and $<VS2>$. It also lists sub- and superclasses of the Component class for each view. For instance, the Components class has two *is-a* related subclasses for View1 and none for View2.

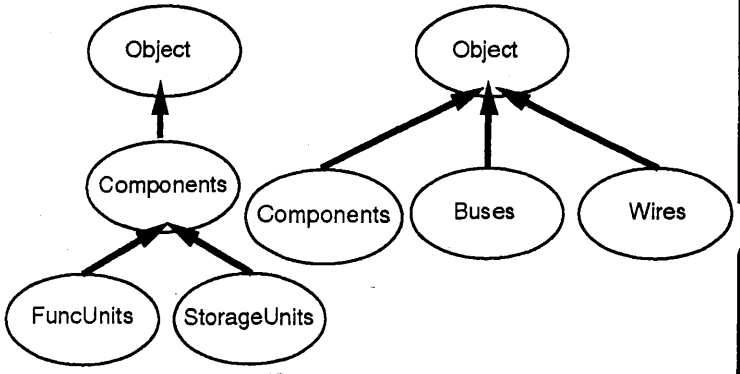
We have chosen the centralized approach over the distributed one for the following reasons. First, this does not require any extension of the class concept. Hence, we can directly use existing object-oriented database technology for an implementation of MultiView. Also, no modification of a class description is required due to the insertion of the class into and deletion of the class from of a view schema. In addition, an operation to remove an obsolete view schemata or to copy one view schemata into another view schemata could easily be accomplished by manipulating the class dictionaries used in the centralized approach (without requiring any search through all the classes distributed throughout the global schema).

10 RELATED WORK

Most initial efforts of defining views for OODBs suggest the use of the query language defined for their respective object model to derive a virtual class. For instance, the work by Heiler and Zdonik [5] and the work by Scholl et al. [14] fall in this category. *MultiView* can use any of these



a.1. Global Schema GS.



a.2. View Schema <VS1>. a.3. View Schema <VS2>.
a. Graphical Representation of Global and View Schemata.

Global	**<GS>**	
Classes	Superclasses	Subclasses
Object	—	Components Connections
Components	Object	FuncUnits StorageUnits
FuncUnits	Components	—
StorageUnits	Components	—
Connections	Object	Buses Wires
Buses	Connections	—
Wires	Connections	—

View1	**<VS1>**	
Classes	Superclasses	Subclasses
Object	—	Components
Components	Object	FuncUnits StorageUnits
FuncUnits	Components	—
StorageUnits	Components	—

View2	**<VS2>**	
Classes	Superclasses	Subclasses
Object	—	Components Buses Wires
Components	Object	—
Buses	Object	—
Wires	Object	—

b. Centralized Approach: Schema Representation using a separate View Table for each Schema.

Class: Object
GlobalSchema (<GS>):
Superclasses: —
Subclasses: Components, Connections
View1 (<VS1>):
Superclasses: —
Subclasses: Components
View2 (<VS2>):
Superclasses: —
Subclasses:Components,Buses,Wires

Class: Components
GlobalSchema (<GS>):
Superclasses: Object
Subclasses: FuncUnits StorageUnits.
View1 (<VS1>):
Superclasses: Object
Subclasses: FuncUnits StorageUnits
View2 (<VS2>):
Superclasses: Object
Subclasses: none

Class: FuncUnits
GlobalSchema (<GS>):
Superclasses: Components
Subclasses: none
View1 (<VS1>):
Superclasses: Components
Subclasses: none

Class: StorageUnits
GlobalSchema (<GS>):
Superclasses: Components
Subclasses: none
View1 (<VS1>):
Superclasses: Components
Subclasses: none

Class: Connections
GlobalSchema (<GS>):
Superclasses: Object
Subclasses: Buses, Wires

Class: Buses
GlobalSchema (<GS>):
Superclasses: Connections
Subclasses: none
View2 (<VS2>):
Superclasses: Object
Subclasses: none

Class: Wires
GlobalSchema (<GS>):
Superclasses: Connections
Subclasses: none
View2 (<VS2>):
Superclasses: Object
Subclasses: none

c. Distributed Approach: Distributing Schema Information across Classes.

Figure 17: Centralized versus Distributed Realization

proposed class derivation mechanisms to implement the first phase of view schema generation, i.e., the customization of individual classes. It thus is a superset of these approaches.

Most of these approaches do not discuss the integration of derived classes into the global schema. Instead, the derived classes are treated as "stand-alone" objects [5], or they are attached directly as subclasses of the schema root class [7]. Scholl et al.'s recent work [14] is an exception; they discuss the classification of virtual classes derived by a selected subset of the operators of the query language COOL into one schema. They do however not consider the problem of generating multiple view schemata, and hence *MultiView* can be considered to be a compatible extension of their work.

Tanaka et al. present an early work on schema virtualization [17]. Their work does not distinguish between the task of integrating derived classes into a common schema and the task of generating view schemata. The interplay between these tasks is not well-defined in their approach. Also, they allow for the arbitrary addition of *is-a* edges in a virtual schema, which in many cases will lead to an inconsistent schema, rather than supporting the automatic generation of the class hierarchy of a view schema as done in *MultiView*. Their approach thus does not assure the validity of a view 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. In fact, by breaking the view schemata definition process into a number of distinct phases, we were able to reduce the view definition language to an extremely simple language. In summary, *MultiView* is a more systematic solution approach compared to their rather ad-hoc proposal.

Shilling and Sweeney [15] present an alternative approach for supporting views for object-oriented systems. Namely, they extend the conventional concept of a class object from having one type (one ADT interface) to having multiple interfaces. The purpose is to limit the access rights to property functions and to control the visibility of instance variables. We accomplish the same goal by using the type refinement capability of the generalization hierarchy to differentiate between different combinations of property functions defined for a collection of objects. Our work is simpler, since it does not require the extension of the traditional class concept. Furthermore, Shilling and Sweeney approach the problem from the programming language point of view, and thus they are not concerned with the sets of objects attached to a class, i.e., the class extent. Consequently, they do not address the derivation of new classes by restricting the membership of a class via a select-like query. Lastly, their approach focuses on one class only, and the effects of multiple interfaces on the class generalization hierarchy are not addressed.

Gilbert's proposal [4], similar to [15], is also based on the idea of defining multiple interfaces for a class object. *MultiView* does not require the extension of the traditional class concept, and thus can be implemented directly with the existing object-oriented database technology, while Gilbert's approach could not. Nonetheless, *MultiView* is as powerful as the multi-interface approach; any view schema that can be defined using the multi-interface approach can also be defined using our strategy. In addition, our work allows for the direct application of the class derivation mechanisms proposed in the literature. The use of general query operators is currently not handled by [4].

11 CONCLUSIONS

In this paper, we have defined an object-oriented view to be a *virtual, possibly restructured, subschema graph* of the global schema rather than just one individual *virtual class.* We have presented a novel approach for supporting these *multiple view schemata* in OODBs, called *MultiView.* This approach is simple yet powerful; it allows for instance for the customization of a view schema by virtually restructuring both the generalization and the property decomposition hierarchies of the underlying global schema.

In this paper, we have also presented solutions to specific subtasks related to the proposed view methodology. First, we have formally defined a set of object algebra operators that can be used to customize the type structure and object membership of virtual classes. Second, we have proposed an algorithm that solves the integration of these newly derived *virtual* classes into the global schema. Third, we have developed a view definition language that can be used by the view definer to select the desired view classes from the global schema. *MultiView* provides support for schema design in terms of automating some parts of the specification process (see subtask two mentioned above) and in terms of enforcing the consistency of a view schema. For instance, in this paper we have presented an algorithm that does only verify the closure property of a view schema but, if the view is found to be incomplete, will transform the view schema into a minimal yet closed view. In this paper, we have also introduced the concept of *view independence*, which we argue to be a fundamental requirement for any view mechanism developed for object-oriented databases. We prove *MultiView* to be *view independent.* Finally, we have outlined some implementation techniques for realizing *MultiView* with existing OODB technology.

Note that the *MultiView* methodology is not specific to a particular OODB model. This generality allows the *MultiView* approach to be incorporated into most existing OODBs. *MultiView* would then enrich these systems by allowing them to support a more powerful notion of views. Our paradigm builds on existing work in as much as it is independent of the class derivation operators chosen from the set of proposed operators in the literature [5, 7, 14, 10]. A major contribution of the proposed approach lies in its simplicity compared to alternative proposals [4], and hence the potential ease in adapting it for existing database systems and in implementing it with existing OODB technology.

We are currently implementing a first prototype of *MultiView.* Based on this prototype, we want to explore alternative implementation strategies for *MultiView.* In particular, the development of efficient query processing techniques for queries issued to view schemata needs to be further researched. Furthermore, the design of a graphical interface for the incremental view definition phase would be a useful feature for application domains. It would open the avenue for non-database experts to utilize *MultiView* to define their desired application-specific views. Indeed, the development of *MultiView* has been driven by our need to provide multiple design views for CAD tools working on a central database, and we are planning to apply *MultiView* to address this problem.

Acknowledgements. I would like to thank Professor Lubomir Bic and Professor Daniel Gajski for providing me with support and encouragement. Without their help, his work would not have come about.

References

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

[2] Banerjee, J., Kim, W., Kim, H. J., and Korth, F., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", *Proc. of ACM SIMOD'87,* May 1987, pp. 311- 322.

[3] Date, C. J., *An Introduction to Database Systems,* Vol. I, Fifth Edition, Addison-Wesley Publishing Company, Inc., 1990.

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

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

[6] Khoshafian, S. N. and Copeland, G. P., "Object Identity," in *Proc. OOPSLA'86,* ACM, Sep. 1986, pp. 406–416.

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

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

[9] Mylopoulos, J., Bernstein, P. A., and Wong, H.K.T., "A Language Facility for Designing Database-Intensive Applications," in *ACM Trans. on Database Systems,* vol. 5, issue 2, pp. 185–207, June 1980.

[10] Rundensteiner, E. A., and Bic, L., "Set Operations in Object-Based Data Models", in *IEEE Transaction on Data and Knowledge Engineering,* to appear in June 1992.

[11] Rundensteiner, E. A., Bic, L., Gilbert, J., and Yin, M. L. "Set-Restricted Semantic Groupings," in *IEEE Trans. on Data and Knowledge Engineering,* to appear in April 1993.

[12] Rundensteiner, E. A. and Bic, L., "Automatic View Schema Generation in Object-Oriented Databases", Univ. of Cal., Irvine, Technical Report #92-15, Jan. 1992.

[13] Schmolze, J. G., and Lipkis, T. A., Classification in the KL-ONE Knowledge Representation System, *The Eigth Int. Joint Conf. on Artificial Intelligence, (IJCAI'83),* Aug. 1983, vol.1, pg. 330 – 332.

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

[15] Shilling, J. J., and Sweeney, P. F., Three Steps to Views: Extending the Object-Oriented Paradigm, in *Proc. of the Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89),* New Orleans , Sep. 1989, 353 – 361.

[16] Shipman, D. W., "The Functional Data Model and the Data Language DAPLEX," in *ACM Trans. on Database Systems,* vol. 6, issue 1, pp. 140–173, Mar. 1981.

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

A Object Algebra Derivation Operators: Syntax, Semantics and Class Relationships

hide		
	syntax	<*virtual*-class> := **hide** [<prop-functions>] **from** (<*source*-class>)
	semantics	**type**(<*virtual*-class>) := {p ∈ P | p ∈ **properties**(<*source*-class>) ∧ p ∉ <prop-functions>} **extent**(<*virtual*-class>) := **extent**(<*source*-class>)
	class rels	<*source*-class> ≼ <*virtual*-class> <*source*-class> ⊆ <*virtual*-class> <*source*-class> *is-a* <*virtual*-class>
refine		
	syntax	<*virtual*-class> := **refine** [<prop-function-defs>] **for** (<*source*-class>)
	semantics	**type**(<*virtual*-class>) := {p ∈ P | p ∈ **properties**(<*source*-class>) ∨ p ∈ <prop-function-def>} **extent**(<*virtual*-class>) := **extent**(<*source*-class>)
	class rels	<*virtual*-class> ≼ <*source*-class> <*virtual*-class> ⊆ <*source*-class> <*virtual*-class> *is-a* <*source*-class>
select		
	syntax	<*virtual*-class> := **select from** (<*source*-class>) **where** (<predicate>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class>) **extent**(<*virtual*-class>) := {o ∈ O | o ∈ <*source*-class> ∧ <predicate>(o) = true}
	class rels	<*virtual*-class> ≼ <*source*-class> <*virtual*-class> ⊆ <*source*-class> <*virtual*-class> *is-a* <*source*-class>
union		
	syntax	<*virtual*-class> := **union**(<*source*-class1>,<*source*-class2>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) ⊓ **type**(<*source*-class2>) **extent**(<*virtual*-class>) := {o ∈ O | o ∈ <*source*-class1> ∨ o ∈ <*source*-class2>}
	class rels	<*source*-class1> ≼ <*virtual*-class> ∧ <*source*-class2> ≼ <*virtual*-class> <*source*-class1> ⊆ <*virtual*-class> ∧ <*source*-class2> ⊆ <*virtual*-class> <*source*-class1> *is-a* <*virtual*-class> ∧ <*source*-class2> *is-a* <*virtual*-class>
intersect		
	syntax	<*virtual*-class> := **intersect**(<*source*-class1>,<*source*-class2>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) ⊔ **type**(<*source*-class2>) **extent**(<*virtual*-class>) := {o ∈ O | o ∈ <*source*-class1> ∧ o ∈ <*source*-class2>}
	class rels	<*virtual*-class> ≼ <*source*-class1> ∧ <*virtual*-class> ≼ <*source*-class2> <*virtual*-class> ⊆ <*source*-class1> ∧ <*virtual*-class> ⊆ <*source*-class2> <*virtual*-class> *is-a* <*source*-class1> ∧ <*virtual*-class> *is-a* <*source*-class2>
diff		
	syntax	<*virtual*-class> := **diff**(<*source*-class1>,<*source*-class2>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) **extent**(<*virtual*-class>) := {o ∈ O | o ∈ <*source*-class1> ∧ o ∉ <*source*-class2>}
	class rels	<*virtual*-class> ≼ <*source*-class1> <*virtual*-class> ⊆ <*source*-class1> <*virtual*-class> *is-a* <*source*-class1>