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

Elke A. Rundensteiner†
Department of Information and Computer Science
University of California, Irvine, CA 92717-3425
rundenst@ics.uci.edu

Abstract

A view in object-oriented databases (OODB) corresponds to virtual schema graph with possibly restructured generalization and decomposition hierarchies. We propose a methodology, called *MultiView*, for supporting multiple such *view schemata*. *MultiView* represents a simple yet powerful approach achieved by breaking view specification into independent tasks: class derivation, global schema integration, view class selection, and view schema generation. Novel features of *MultiView* include an object algebra for class customization; an algorithm for the integration of virtual classes into the global schema; a view definition language for view class selection, and the automatic generation of a view class hierarchy. 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. Lastly, we introduce the fundamental concept of *view independence* and show *MultiView* to be *view independent*.

1 Introduction

Relational views have been of limited use, because in many systems they cannot be updated. Views in OODBs are more likely to play an important role for defining customized interfaces for advanced applications, since updates can be handled better due to:

1. object identity; maintaining the unique identity of an object even if its external characteristics are modified and/or hidden (in a view), and
2. abstract data types; associating type-specific (update) operations with the encapsulated object.

† The author's current address is University of Michigan, Ann Arbor, Dept. of EECS, Ann Arbor, MI 48109-2122.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

While the concept of views has been studied extensively in the context of the relational model, it is largely unexplored for OODBs. Initial proposals of views on OODBs have emerged that define a view to be a *virtual class* derived by an object-oriented query [Heil90, Scho91, Kim89]. An object-oriented schema is a complex structure of classes interrelated via various relationships, such as, the generalization and decomposition hierarchies [Kim89, Bane87]. An object-oriented view should thus be defined to be a *virtual, possibly restructured, subschema graph* of the global schema [Tana88]. This raises a number of challenging research issues in terms of how to restructure such view schema graphs and how to relate them with the global schema.

In this paper, we propose a methodology, called *MultiView*, for supporting multiple *view schemata* that successfully solves these problems. *MultiView* breaks view specification into three tasks: (1) customization of virtual classes, (2) integration of virtual classes into *one* consistent global schema and (3) the specification of arbitrarily complex view schemata on this global schema. *MultiView*'s division of view specification into a number of well-defined tasks, some of which have been successfully automated, makes it a powerful tool for supporting the specification of views by non-database experts while enforcing view consistency. In this paper, we outline the overall approach and present a solution to the first task of *MultiView*, while solutions to the second and third task are given in [Rund92d] and [Rund92c], respectively.

Though *MultiView* is independent of particulars of the class derivation operators, we define a set of object algebra operators for the purpose of this work [Rund92b]. We study in particular the class relationships between the virtual and the source classes, since this is required for solving *MultiView*'s second task.

Class integration, the second task of *MultiView*, tackles the problem of how a virtual class relates to, and can be integrated with, the remaining classes in the global schema [Rund92d]. 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. 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 via a property decomposition hierarchy (for forming complex objects). Class integration needs to guarantee the consistency of these class relationships when adding new classes [Rund92d].

We cannot modify the existing schema so that it suits the requirements of one user. Instead, we need to support a number of *different, potentially conflicting, view schemata* of the same schema. We thus are concerned with the *virtual restructuring* of the global schema for each view; rather than with permanently changing the global database as is done in schema evolution [Bane87].

We solve the third task of *MultiView* by dividing it into 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 generalization hierarchy [Rund92c].

We have developed criteria for the *closure* of the property decomposition and for the *consistency* of the generalization hierarchies of a view. In this paper, we present an algorithm for checking the closure property of a view schema. Given a non-closed view, this algorithm is guaranteed to transform the non-closed view into a closed, yet minimal, view schema (Section 7). We present proofs of correctness and a complexity analysis for the closed-view generation algorithm.

Lastly, we introduce the concept of *view independence*, which we argue to be a fundamental requirement for any OODB view mechanism – similar to the well-known concept of data independence. In Section 8, we show *MultiView* to be *view independent.*

In Sections 2 and 3, we introduce object-oriented concepts and describe *MultiView*, respectively. The object algebra is presented in Section 4, while class integration is discussed in Section 5. We introduce the view specification language and the closed-view-generation algorithm in Sections 6 and 7, respectively. *MultiView* is shown to be view independent in Section 8. We present related work and conclusions in Sections 9 and 10, respectively.

2 Object-Oriented Concepts

2.1 The Object Data Model

Let P be an infinite set of property functions. Each p $\in$ P can be a value from a simple enumeration type, an object instance from some class, an arbitrarily complex function, or an object method. Each p $\in$ P has a name and signature (i.e., domain types). For simplicity, we assume that all p $\in$ P have unique property identifier[1]. Let T be the set of all types. For $t \in T$, $\mathbf{properties}_t$ corresponds to the set of property functions of t and $\mathbf{domain}_p(t)$ denotes the domain of p in t.

[1] To determine whether two property functions are identical is equally hard to proving that two programs are equivalent. We therefore ensure uniqueness of properties by associating a unique property identifier with each newly defined property [Rund92b].

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. 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, then we refer to the domain of p for C by $\mathbf{domain}_p$(C). A **class** is also a container for a set of objects. Let O be an infinite set of object instances. The collection of objects that belong to a **class** C is denoted by **content**(C) = $\{o \mid o \in C\}$ with the predicate "$\in$" defined based on object identities [Rund93].

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**$(C2)$) *(*$\mathbf{domain}_p(C1) \subseteq \mathbf{domain}_p(C2)$).

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).*

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 = t1 $\sqcup$ t2. The property functions of t3 are defined by* **properties***(t3) =* **properties***(t1) $\cup$* **properties***(t2.). For each property function p $\in$* **properties***(t3), we define* $\mathbf{domain}_p(t3) = \mathbf{domain}_p(t1) \cap \mathbf{domain}_p(t2)$.

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 = t1 $\sqcap$ t2. The property functions of t3 are defined by* **properties***(t3) =* **properties***(t1) $\cap$* **properties***(t2). For each property p $\in$* **properties***(t3), we define* $\mathbf{domain}_p(t3) = \mathbf{domain}_p(t1) \cup \mathbf{domain}_p(t2)$.

Definitions 4 and 5 define the *greatest common subtype* and the *lowest common supertype* of two classes, respectively.

Let S = $\{ C_i \mid i = 1,...,n\}$ 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 ... (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 for which the above *is-a* relationships hold. The *direct subclass* relationship is denoted by (C_1 *is-a*^{d} C_n) and the *indirect* one by (C_1 *is-a** C_n).

Definition 6. *An* **object schema** *is a directed acyclic graph*[2] *S=(V,E), where V is a finite set of vertices and E is a finite set of directed edges. Each element in V corresponds to a class C_i, while E corresponds to a binary relation on $V \times V$ 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). There is one designated root node, called* **Object**, *which contains all object instances of the database and its type description is empty*[3].

We refer to the set of *is-a* relationships of a schema as the **generalization hierarchy**. A class is related to other classes via property relationships. For example, if C1 has defined a property function p with $\textbf{domain}_p$(C1)=C2, then we say that there is a property decomposition arc between C1 and C2 labeled 'p'.

Definition 7. *Let S=(V,E) be an object schema. Let L be a set of labels that correspond to the names of the property functions in P. Then the* **property decomposition hierarchy** *of 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 $V \times V \times L$, called the 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.*

A property decomposition hierarchy consists of one or more disconnected subgraphs with possibly loops, self-loops, and multi-edges.

2.2 Object-Oriented Views

We distinguish between **base** and **virtual** classes. **Base** classes are defined during the initial schema definition and their object instances 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 membership based on the state of the database. The content 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 classes in V correspond to base classes with stored rather than derived instances.*

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

[3] The schema root class provides a unique entry point into the database. Note that this definition is not limiting, since in reality the database schema may correspond to a set of DAGs (some of which may of course be isolated classes) – with their interconnection to the **Object** root possibly hidden to the user.

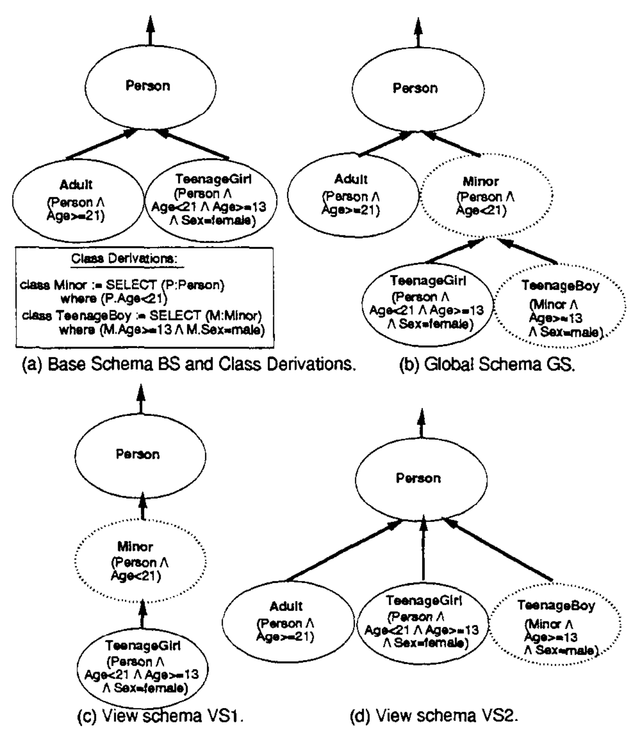
Definition 9. *The* **global schema** *(GS) is an extension of the base schema BS augmented by the collection of all virtual classes defined during the lifetime of the database as well as their is-a relationships.*

A subgraph of the global schema which contains only virtual classes is commonly called a *virtual schema* [Tana88, Abit91].

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:*

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

The first condition states that each view schema is uniquely identifiable. The second states that all classes of VS also have to be classes of *GS*. The third states that the view schema maintains only *is-a* relationships among its view classes that are directly derivable from *GS*. We call the classes in a view schema (both the base and the virtual ones) *view classes* and the *is-a* relationships *view is-a relationships*.



(a) Base Schema BS and Class Derivations. (b) Global Schema GS.
(c) View schema VS1. (d) View schema VS2.

Figure 1: Base, Global & View Schemata Examples.

Example 1. *Figure 1 shows (a) the base schema BS, (b) the global schema GS, and (c) and (d) two view schemata. We depict base and virtual classes by circles and dotted circles, respectively. GS in Figure 1.b* *is derived from BS in Figure 1.a by adding the virtual classes* **Minor** *and* **TeenageBoy** *and by interconnecting them with the remaining classes. The view schemata in Figure 1.c and 1.d are derived from GS by selecting a subset of its classes and interconnecting them into a valid schema using view is-a arcs.*

2.3 The Closure of the View Property Decomposition Hierarchy

This section addresses the consistency of the property decomposition hierarchy [Tana88, Heil90], while the consistency of the generalization hierarchy is handled in [Rund92c]. Let the function *Uses(C)* represent the set of classes that are used by *C*'s type interface. For example, if p corresponds to an object pointer defined by $\mathbf{domain}_p$(C)=C2, then *Uses(C)* contains C2.

Definition 11. *Let C be a finite set of classes, L a finite set of property labels, PD=(C,A,L) a property decomposition hierarchy. Then Uses:C* $\longrightarrow 2^C$ *is a function defined by: For* $Ci, Cj \in C$*, for* $pk \in L$*,* $Uses(Ci)=\{Cj \in C | a_{ij} =< Ci, Cj, pk > \in A\}$. *For* $S \subseteq C$*,* $Uses(S) = \cup_{Ci \in S} Uses(Ci)$. *We define the closure operator * by* $Uses^*(Ci) = \cup_{j=1}^{|V|} Uses^j(Ci)$ *with* $Uses^1(Ci) = Uses(Ci)$ *and* $Uses^j(Ci) = Uses(Uses^{j-1}(Ci))$ *for* $j > 1$.

Uses(Ci) (*Uses*(Ci)*) corresponds to the classes that are directly (directly or indirectly via transitive closure) used by the class *Ci*.

Definition 12. *A view schema VS=(VV,VE) is defined to be a* **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 being *used* in a view (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).

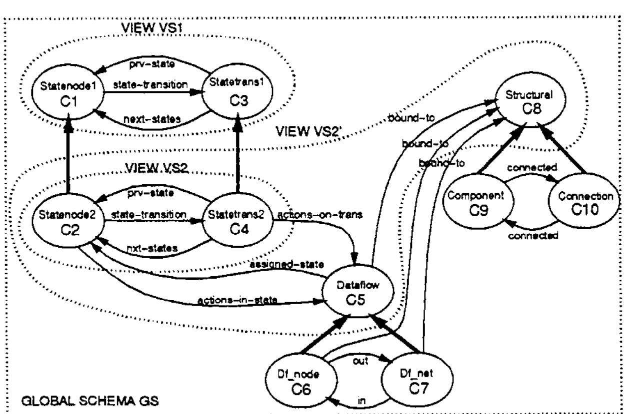


Figure 2: Examples of Closed and Non-Closed Views.

Example 2. *Figure 2 depicts is-a and property decomposition relationships by bold arrows without and by regular arrows with labels, respectively. A (view) schema is denoted by encircling its (view) classes by a dotted line. The views VS1 and VS2' are closed. The view VS2 =* {**Statenode2**, **Statetrans2**} *is not closed, since the 'actions-in-state' property defined for the view class* **Statenode2** *has the domain class* **Dataflow**, *which is not contained in VS2.*

3 The *MultiView* Methodology

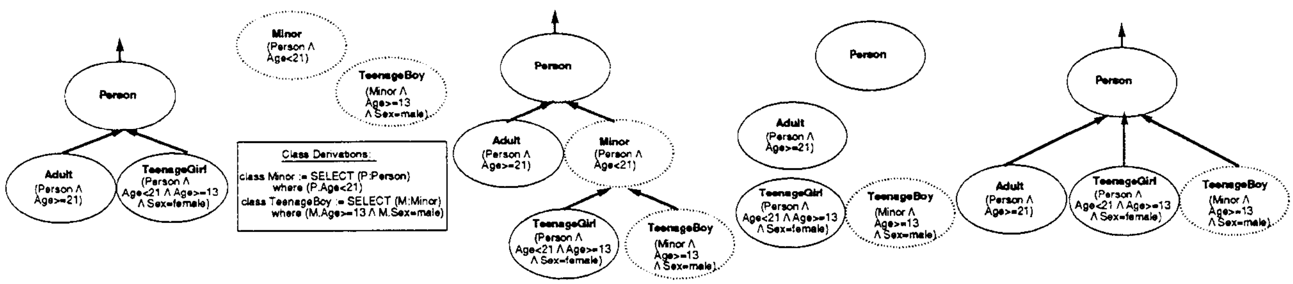
MultiView is a methodology for supporting *multiple view schemata* in OODBs. *MultiView* breaks view specification into three independent tasks:

1. the customization of types and object sets by deriving virtual classes via object-oriented queries,
2. the 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 the augmented global schema.

The separation of the view design process into a number of well-defined tasks has several advantages. First, it simplifies view specification, since each of the tasks can be solved independently from the others. Second, it increases the level of support by allowing for the automation of some of the tasks. We present algorithms for automating the second task and the third task in [Rund92d] and in [Rund92c], respectively.

The first task of *MultiView* supports the virtual customization of existing classes by deriving new classes with a modified type description and/or membership content. *MultiView* uses these class derivation mechanisms for 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. Since there is no generally agreed-upon object algebra, we define our own object algebra for this work (and for the first prototype of *MultiView*) in Section 4. It is similar in flavor to the ones proposed in the literature [Kim89, Heil90, Scho91].

MultiView supports the integration of virtual classes into one comprehensive global schema [Rund92d]. This integration takes care of the maintenance of explicit class relationships between stored and derived classes. This is useful for sharing property functions and object instances consistently among classes without unnecessary duplication. Class integration also assures the consistency of all views with the global schema and with one another. Last but not least, it is a necessary basis for the third task 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



(a) Base Schema BS. (b) Class Customization. (c) Class Integration into the Global Schema GS. (d) View Class Selection. (e) View Schema Generation.

Figure 3: The *MultiView* Approach: From Base over Global to View Schemata.

the classes in the global schema, then a view would correspond to a collection of possibly 'unrelated' classes rather than a schema graph (Definition 6).

The third task of *MultiView* utilizes the augmented global schema for the selection of both base and virtual classes and for arranging these view classes into a consistent class hierarchy. This supports the virtual restructuring of the generalization and the property decomposition hierarchies by allowing us to hide from and to expose classes within a view schema. For the explicit selection of view classes, we have developed a view definition language that can be used by the view definer to specify the classes required for a particular view (see Section 6).

We also present an algorithm for checking the closure property of a view schema. Given a non-closed view, 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). We now give an example of the tasks involved in constructing a *view schema* in *MultiView.*

Example 3. *Given the global schema GS in Figure 3.a, the view definer specifies the two virtual classes* **Minor** *and* **TeenageBoy** *using object-oriented queries (Figure 3.b). The integration of the two virtual classes into GS is given in Figure 3.c. View schema definition now proceeds by selecting a subset of classes from the augmented GS (Figure 3.d). Lastly, the chosen view classes are interconnected into one view schema (Figure 3.e).*

4 Class Customization Using Object Algebra

The *MultiView* methodology is independent from the particular object algebra chosen for the class derivation task. However, since there is no agreed-upon standard, we present a representative set of algebra operators. We have shown the distinction between the type and the set aspect of a class to be a valuable tool for characterizing the semantics of query operators on object-based data models [Rund92b]. In this vein, we define the semantics of the operators by characterizing their manipulation of the type and the set aspect of the source class. We also focus on the *subset, subtype* and *subclass* relationships among the source and result classes, since this is a necessary foundation for successfully addressing the, generally ignored, class integration problem. Table 1 summarizes the object algebra operators, in particular, it gives their syntax, semantics and the resulting class relationships.

The **hide** operator modifies the type description of a class by hiding some of its property functions - similar to the project operator in relational algebra. It has the syntax "<*virtual*-class> = **hide** [<prop-functions>] **from** (<*source*-class>)" with <prop-functions> being one or more property functions defined for <*source*-class>. It removes the property functions listed in the set <prop-functions> from the source class while preserving all others. The set content of the virtual class is equal to the set content of the source class.

The **refine** operator is a type-manipulating operator that refines an existing type description by adding additional property functions. It has the syntax "<*virtual*-class> = **refine** [<prop-function-defs>] **for** (<*source*-class>)" with <prop-function-def> being the definition of a new property function in the form of a new property name and a function body with the latter a legal arithmetic, boolean or set expression. The property functions in <prop-function-defs> are assumed to be distinct from all others in the global schema and therefore get assigned a unique property identifier. The set content of the virtual class is equal to the set content of the source class.

The **select** operator is a set-manipulating operator that selects a subset of object instances from a given set of objects - similar to the select operator of relational algebra [Date90]. It has the syntax "<*virtual*-class> = **select from** (<*source*-class>) **where** (<predicate>)" with <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. The type stays the same.

Set operators manipulate both the type description and the set membership of their two source classes. A detailed analysis of these set operators for

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> $\preceq$ <*virtual*-class> <*source*-class> $\subseteq$ <*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> $\preceq$ <*source*-class> <*virtual*-class> $\subseteq$ <*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> $\preceq$ <*source*-class> <*virtual*-class> $\subseteq$ <*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> $\preceq$ <*virtual*-class> ∧ <*source*-class2> $\preceq$ <*virtual*-class> <*source*-class1> $\subseteq$ <*virtual*-class> ∧ <*source*-class2> $\subseteq$ <*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> $\preceq$ <*source*-class1> ∧ <*virtual*-class> $\preceq$ <*source*-class2> <*virtual*-class> $\subseteq$ <*source*-class1> ∧ <*virtual*-class> $\subseteq$ <*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> $\preceq$ <*source*-class1> <*virtual*-class> $\subseteq$ <*source*-class1> <*virtual*-class> *is-a* <*source*-class1>

Table 1: The Object Algebra Operators: Syntax, Semantics and Class Relationships.

OODBs can be found in [Rund92b]. The semantics of the **union** operator are to return a set of object instances composed of the members of either or both of the source classes. The resulting type description is equal to the lowest common supertype of the two sources classes (Definition 5). The **intersect** operator returns a set of object instances that are members of both source classes. Furthermore, the type description of the resulting virtual class is equal to the greatest common subtype of the two sources classes (Definition 4). Lastly, the **difference** operator returns a set of object instances that are members of the first but not of the second source class. The resulting type description is equal to the description of the first source class.

Example 4. *In Figure 4, the is-a relationships between the virtual and the sources classes are indicated by bold arrows. Figure 4.a depicts the query "***BehaviorGraph = hide** *[SetState, GetState]* **from** *(***StateGraph***)". Then* **extent***(***BehaviorGraph***) =* **extent***(***StateGraph***) and* **type***(***BehaviorGraph***) = [Domain, NodeOp].*

*In Figure 4.b, the query "***Comps2 = refine** *[Area = Height * Width]* **for** *(***Comps***)" derives* **Comps2**. *We have* **extent***(***Comps2***) =* **extent***(***Comps***). The type of* **Comps2** *has been extended by the new method Aera, hence* **Comps2** $\preceq$ **Comps**. **Comps2** *is integrated into GS by placing* **Comps2** *below* **Comps** *as direct subclass.*

*In Figure 4.c, the query "***Adders = select from** *(***Comps***)* **where** *(Plus in* **Comps***.Ops)" derives* **Adders** *from* **Comps**. *The* **Adders** *class consists of all object members of* **Comps** *that implement the Plus operator, thus* **Adders** $\subseteq$ **Comps**. **Type***(***Adders***) =* **type***(***Comps***).*

*In Figure 4.d, the query "***GraphConstructs = union***(***DataFlow,ControlFlow***)" derives* **GraphConstructs**. *Then* **extent**(**GraphConstructs**) = **extent**(**DataFlow**) ∪**extent**(**ControlFlow**) = $\{D1, D2, D3, C1, C2\}$.*Also***type**(**GraphConstructs**) = **type***(***DataFlow***)* ⊓ **type***(***ControlFlow***) = [Domain]. The is-a relationships are indicated by the edges (***DataFlow** *is-a* **GraphConstructs***) and (***ControlFlow** *is-a* **GraphConstructs***).*

In Figure 4.e, the **intersect** *operator is used in the query* **FexLayout** = **intersect***(***DataPathUnits,RandomLogicUnits***). Then* **extent***(***FexLayout***) =* **extent***(***DataPathUnits***)* ∩

extent(**RandomLogicUnits**) = $\{O1, O2\}$. *And* **type**(**FexLayout**) = **type**(**DataPathUnits**) $\sqcup$ **type**(**RandomLogicUnits**)= *[Comp-Type, DF-Construct, CF-Construct, get-DF-Graph].*

In Figure 4.f, the **diff** *operator is used in "***AllOtherComps = diff**(**Components,ALUs**)*" to derive* **AllOtherComps** *from* **Components** *that are not in* **ALUs**. *We have* **extent**(**AllOtherComps**) = **extent**(**Components**) − **extent**(**ALUs**) = $\{O3, O4, O5\}$. *And* **type**(**AllOtherComps**) = **type**(**Components**) = *[Get-Name, Comp-Type]. The relationship (***AllOtherComps** *is-a* **Components**) *has been added to Figure 4.f.*

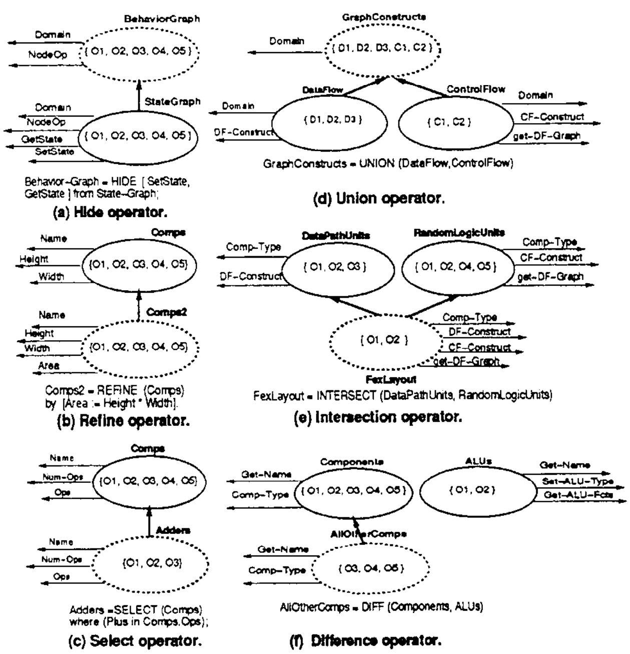


Figure 4: Examples of Class Derivation.

5 Class Integration

MultiView integrates all virtual classes derived for different views into one global schema in order to explicitly represent the generalization relationships between virtual and base classes. In this section we sketch an overall approach for the class integration problem. A detailed treatment of this topic is beyond the scope of this paper and can be found elsewhere [Rund92d].

Class integration is concerned with finding the most 'appropriate' location in the schema graph G for a virtual class VC in terms of property inheritance and subset relationships between classes. For this, the classifier determines the *is-a* relationships between the virtual class VC and all other classes in GS by comparing their type descriptions and their membership predicates. The algorithm for finding the correct position for VC in G=(V,E) can be summarized as follows. First, we find all classes in G that are the direct superclasses of VC defined by direct-parents(VC) = $\{C_i \mid (VC\ is\text{-}a\ C_i) \wedge (\nexists C_j \in V)(j \neq i)((VC is\text{-}a\ C_j) \wedge (C_j is\text{-}a\ C_i))\}$. Similarly, we find all classes in G that are the direct subclasses of VC defined by direct-children(VC) = $\{C_i \mid (C_i is\text{-}a\ VC) \wedge (\nexists C_j \in V)(j \neq i)((C_i is\text{-}a\ C_j) \wedge (C_j is\text{-}a\ VC))\}$. VC 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 for OODB models since it may involve the comparison of arbitrary functions and predicates. In the worst case, if some *is-a* relationship is not discovered, then the virtual class is placed higher in the class hierarchy than would theoretically be possible. This would be a correct but not the most informative class arrangement.

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 object algebra operator [Rund92a]. For instance, for the **refine** operator, which produces a virtual class with a new property function p, this algorithm can be reduced to a simple O(1) algorithm requiring no search. The reader is referred to [Rund92d] for more details. We complete this section by demonstrating the classification process on an example.

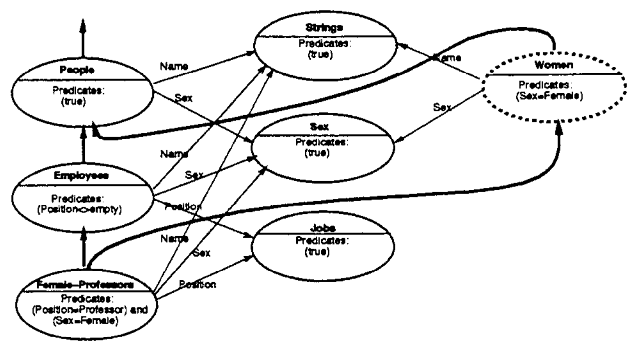


Figure 5: Integrating the Class **Women** Into GS.

Example 5. *In Figure 5, the virtual class* **Women** *is derived by the query "***Women = select from** (**People**) **where** *Sex=female)". From Section 4, we can deduce the following class relationships:* (**Women** $\subseteq$ **People**), (**Women** $\preceq$ **People**), *and* (**Women** *is-a* **People**). *We therefore insert the edge* (**Women** *is-a* **People**) *into GS. Next, we search for the most specialized classes that are still is-a related with the* **Women** *class. 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 the subset relationship* (**Female-Professor** $\subseteq$ **Women**) *can thus add the is-a relationship (***Female-Professor** *is-a* **Women***) in form of an edge to the graph.*

6 View Schema Specification

Next, we discuss the third task of *MultiView*, namely, the definition of a view schema on top of the global schema. We divide view specification into two subtasks:

1. the selection of view classes, and
2. the generation of view relationships between the view classes.

This separation into two subtasks reduces view specification to a simple activity. For the first subtask, we define a view definition language that can be utilized by the view definer for the specification of view schemata. For the second subtask, we have developed algorithms that will automatically generate a generalization hierarchy from a given set of view classes. This automatic generation of view *is-a* arcs is preferable over their manual entry since it simplifies the task of the view designer and guarantees the consistency of the resulting view schema. Details about the view definition language and the automatic view generation can be found in [Rund92c], while below we introduce the underlying ideas.

The view definition language consists of two groups of operators: the first group either initiates or terminates a transaction on a view schema while the second group discussed in the next paragraph modifies a given view schema. The `DEFINE-VIEW` command for instance initializes a new view schema and assigns a unique view identifier to it, while the `MODIFY-VIEW` command prepares an already defined view schema for modification. All operators specified within a view definition transaction, i.e., after a `DEFINE-VIEW` or a `MODIFY-VIEW` command and before the terminating `END-VIEW` command, will modify only the one designated view schema VS. The view definers conclude the view definition phase by issuing the `SAVE-VIEW` command. *MultiView* then automatically augments the set of classes by the necessary view *is-a* arcs [Rund92c].

The second group of commands modifies the view VS by either adding or deleting view classes. The "`ADD-CLASS(`<*class*-`name`>`)`" command adds a class <*class*-`name`> in GS to VS. The "`ADD-CLASS-DAG(`<*class*-`name`>`)`" command adds all classes to 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 <*view*-`name`> to VS. The commands `REMOVE-CLASS`, `REMOVE-CLASS-DAG`, and `REMOVE-VIEW-SCHEMA` do the same as the just described operators but rather than adding they delete the respective classes. Lastly, the "`RENAME-CLASS` command renames a view class of VS by replacing its name <*old-class*-`name`> by the new name <*new-class*-`name`>.

Example 6. *A* **view creation script** *for the view VS depicted in Figure 3.e is given below.*

```
DEFINE-VIEW VS
   class Minor = select (P:Person)
      where (P.Age<21);
   class TeenageBoy = select (M:Minor)
      where (M.Age>=13) and (M.Sex=male);
   ADD-CLASS (TeenageBoy);
   ADD-VIEW-SCHEMA (BS);
   SAVE-VIEW;
END-VIEW
```

First, the DEFINE-VIEW VS command creates an empty view schema with the identifier VS. We then define the virtual classes **Minor** *and* **TeenageBoy** *(Figure 3.b) and integrate them into GS (Figure 3.c).* **TeenageBoy** *is added to the view with the command* `ADD-CLASS(`**TeenageBoy**`)`. *Then the three classes of the base schema are added to VS using the command* `ADD-VIEW-SCHEMA(BS)`. *The selected view classes are shown in Figure 3.d. When VS is saved, the is-a arcs shown in Figure 3.e are derived automatically by MultiView [Rund92c].*

7 Automatic Generation of a Closed View Schema

7.1 The Minimality Criterion

The closure criterion of a view schema can be verified only after the selection of all view classes, since it is a function of (the relationships among all classes in) the complete schema. As indicated in Section 2.3, instead of checking whether a given view is closed or not, it is more useful to also transform a view that is found to be not closed into a *closed* view schema. The Closed-View-Generation algorithm presented in this section solves this problem. In particular, it determines the minimal[4] set of classes by which the view VS has to be extended in order for VS to be *closed*. We describe this minimal set below.

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.*

[4] 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 *closed*.

Proof: We prove Theorem 1 in two parts. Part I show the sufficiency and part II the necessity of MIN for closure. These two facts together imply the correctness of Theorem 1.

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

Case I.a: Let VS=(VV,VE) be a view that is already closed. By Definition 12, VV = VV $\cup$ $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$. By subtracting the set VV from both sides of the equation, we derive $\bigcup_{C_i \in VV}(Uses^*(C_i)) - VV = \emptyset$. This implies MIN = $(\bigcup_{C_i \in VV}(Uses^*(C_i))) - VV = \emptyset$. Since VS is assumed to be closed, no classes need to be added to VS.

Case I.b: Let VS=(VV,VE) be a view that is not closed. Then create a new view VS'=(VV',VE') with VV' = VV $\cup$ MIN. Then VV' = VV $\cup$ MIN = VV $\cup$ $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$ - VV) = VV $\cup$ $(\bigcup_{C_i \in VV}(Uses^*(C_i)))$.

$\bigcup_{C_i \in VV'}(Uses^*(C_i))$
$= \bigcup_{C_i \in (VV \cup \bigcup_{C_k \in VV}(Uses^*(C_k)))} (Uses^*(C_i))$
$= \bigcup_{(C_i \in VV) \vee (C_i \in \bigcup_{C_k \in VV}(Uses^*(C_k))))} (Uses^*(C_i))$
$= \bigcup_{C_i \in VV}(Uses^*(C_i)) \cup$
$\quad \bigcup_{C_i \in (\bigcup_{C_k \in VV}(Uses^*(C_k)))}(Uses^*(C_i))$
$= \bigcup_{C_i \in VV}(Uses^*(C_i))$
$\subseteq$ 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' $\cup$ $(\bigcup_{C_i \in VV'}(Uses^*(C_i)))$. By Definition 12, we thus have shown that VS' is closed. ■

Part II: MIN is the *minimal* set of classes required to make the view VS closed.

Case II.a: Let VS=(VV,VE) be a view that is closed. Then, by part I.a, MIN = $\emptyset$. By default, the empty set is 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 12 for a view VS that is not closed. Namely, 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 $\bigcup_{C_i \in VV}(Uses^*(C_i)) - VV$, which is equal to MIN, must be added to VV. ■

7.2 CVG Algorithm and Examples

An algorithm for Closed-View-Generation (CVG) is given in Figure 6. CVG determines whether a view is closed or not. If the view VS is not closed then the algorithm automatically determines the minimal set of classes by which VS has to be extended in order to be *closed.* This is done by recursively exploring the *Uses* relationships of classes. Note that the *Uses* relationships of a class C are independent from the class of the schema by which C has been reached. This observation reduces the complexity of the transitive closure portion of the algorithm from cube to linear complexity. Once we have processed a class C_i by checking its *Uses* relationships, it need not be checked anymore (it then is placed into CVG-done).

Data Structures and Variables:
Set of classes: CVG-tmp, CVG-done;
Classes: Ci, Ck;
Boolean flag: Closed;
Procedures and Functions:
get-and-remove-next(set-of-classes) → class;
not-element(class,set-of-classes) → boolean;
add-to-set(class,set-of-classes);
Input:
Global and View schemata *GS = (V,E), VS=(VV,VE)*
Output:
Closed: flag to indicate whether the view is closed.
CVG-done: set of classes required for closure of VS.
Algorithm CVG: Closed-View-Generation Algorithm.

```
algorithm CVG(GS,VS)
return (set-of-classes,boolean-flag) is
    CVG-done = ∅; CVG-tmp = VV; Closed = true;
    while (Ci=get-and-remove-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 6: The Closed-View-Generation Algorithm.

CVG maintains all classes reached via the *Uses* relationship that still have to be processed in CVG-tmp. While there are any classes left to be processed in CVG-tmp, the algorithm picks one of them, say *Ci*. 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 CVG-done: this 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 *Uses(Ci)*, whether they have to be processed for closure. They do not have to be processed for closure, if either they have already 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 VS have been processed, i.e., CVG-tmp is empty. If the view is *closed* (not *closed*), then the algorithm returns "Closed=true" and "CVG-done=$\emptyset$" ("Closed=false" and "CVG-done$\neq \emptyset$"). CVG-done contains all classes that have to be added to VS to make it *closed*, i.e., CVG-done = MIN (Theorem 1).

Example 7. *CVG is applied to the view VS1 in Figure 2. CVG first initializes CVG-tmp=*$\{C1, C3\}$*. For the first while-loop iteration with Ci=C1, the first if-statement evaluates to false and is skipped. Due the 'state-transition' property defined for C1, Uses(C1) =* $\{C3\}$ *Therefore, the for-loop is executed but once with Ck = C3. The second if-statement evaluates to false, since (C3* $\in$ *VV). For the second while-loop iteration with Ci=C3, the first if-statement is again skipped. Uses(C3) =* $\{C1\}$*. The second if-statement is false, since (C1* $\in$ *VV). CVG terminates with (Closed=true) and (CVG-done=*$\emptyset$*). VS1 thus is closed.*

Example 8. *CVG is applied to the view VS2 in Figure 2. CVG first initializes CVG-tmp=*$\{C2, C4\}$*. For the first while-loop iteration with Ci=C2, the if-statement evaluates to false and is skipped. Since Uses(C2) =* $\{C4, C5\}$*, the for-loop has two iterations. For Ck=C4, the if-statement is skipped. For Ck=C5, the if-statement evaluates to true and C5 is added to CVG-tmp for further processing. For the second while-loop iteration with Ci=C4, the first if-statement is skipped. The two for-loop iterations with Uses(C4) =* $\{C2, C5\}$ *both are skipped. For the third while-loop iteration with Ci=C5, the first if-statement evaluates to true since C5* $\notin$ *VV. Therefore, C5 is added to CVG-done Closed is set to false. Since Uses(C5) =* $\{C2, C8\}$*, the for-loop has two iterations. For the second iteration with Ck=C8, the if-statement evaluates to true and C8 is added to CVG-tmp. For the fourth and last while-loop iteration with 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 terminates with (Closed=false) and (CVG-done=*$\{C5, C8\}$*). Adding CVG-done to VS2 results in the closed view VS2'.*

7.3 The Correctness and Complexity of Closed-View-Generation

Theorem 2. (Correctness) *Given a view schema VS=(VV,VE) defined on GS=(V,E), then the closed-view generation algorithm CVG in Figure 6 correctly generates a closed view VS'. In particular, CVG returns Closed=true if VS is closed, and Closed=false, otherwise. If 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 the correctness of CVG in two parts. In part I, we show that the algorithm correctly determines whether a view is closed or not, i.e., (Closed=true) $\Longleftrightarrow$ (VS is closed). In part II, we show that the algorithm actually generates the set of additional classes needed to make VS closed, i.e., CVG-done = MIN. Proofs for part I and part II are beyond the scope of this paper and can be found in [Rund92a]. Finally, part I and II together prove Theorem 2. ■

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

Proof: The detailed proof for Theorem 3 can be found in [Rund92a], while below we outline the key observations. First, we can show that all functions (and thus the two if-statements) used by CVG have constant complexity. Next, we can show that each class *Ci* of GS is placed at most once into CVG-tmp, and hence the while-loop is executed at most once for each *Ci*. Third, the for-loop has exactly one iteration for each class Ck in the *Uses(Ci)* set of *Ci*. $|Uses(Ci)| \leq \#arcs(Ci)$ with $\#arcs(Ci)$ equal to the number of outgoing property decomposition arcs of *Ci*. Complexity(CVG) $\leq O(\sum_{Ci \in V}(|Uses(Ci)|)) \leq O(\sum_{Ci \in V}(\#\text{arcs}(Ci))) = O(\min(|V|,|A|))$. ■

8 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" [Date90]. This is achieved by separating the interface to the database (the conceptual data model) from the actual implementation (the physical data model). A system provides *logical data independence* by supporting a view mechanism that lets the users define their own view schema on top of the common logical schema. *Data independence* does not protect the user from having to update the specification of possibly all existing views when the underlying data model is extended and/or reorganized.

Definition 13. *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.*

The concept of *view independence* is an important requirement for OODB systems, since the underlying base schema is restructured with the definition of possibly each new view schema. A redefinition of all existing views for whenever a new view schema is introduced would be unacceptable. *View independence* does not have any significance in relational databases where the definition of new views has no affect on the underlying base schema.

Definition 14. *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*[5]*. Let GS' = (V',E') be the global schema and VS'=(VV',VE') the view schema derived from VS after 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 VS are* **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,* $(\forall C_i' \in C)((C_i' \in VV') \Longrightarrow (\exists! C_i \in VV)(C_i = m^{-1}(C_i'))$[6].
- $(\forall C_i \in VV)\ (\forall o \in O)\ ((o \in C_i)$ *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 for VV are* **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* $* m(C_j)) \in VE'))$.

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

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

For *MultiView* to be view independent means that view generation does not affect the types and contents of view classes of existing views nor their view *is-a* relationships.

Theorem 4. *Let VS* be the set of all view schemata defined on GS. MultiView* **preserves** *the view classes of all view schemata in VS* through the restructuring of GS using the function* Π *(Definition 14.a)*[7]

The proof for Theorem 4 can be found in [Rund92a], while below we give the intuitive reasoning. *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 to the global schema. We thus need to show that all *Ci* of GS are preserved when integrating new classes into GS. Recall that the integration algorithm follows the principle that VC is inserted directly below its direct superclasses and directly above its direct subclasses in GS (Section 5). 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, more precisely, ($\forall C_i \in$ 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 existing classes, i.e., part II of Definition 14.a holds. The insertion of VC also does not modify their types. All classes that are made subclasses of VC in the modified GS are also subtypes of VC; i.e., their types will be preserved. This shows part III of Definition 14.a. ■

Theorem 5. *Let GS be a global schema and VS* be the set of all view schemata defined on GS. MultiView* **preserves** *the view is-a relationships among the view classes of each view in VS* through the restructuring of GS using the function* Π *(Definition 14.b).*

A proof for Theorem 5 can be found in [Rund92a]. *MultiView* derives the *is-a* relationships of view classes directly from their *is-a* relationships in GS, 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. As shown in Theorem 4, we can deduce that the classes in the direct-parents(VC) and the direct-children(VC) set were *is-a* related before the insertion of VC. Therefore, the insertion of VC does not add any new *is-a* relationships. Obviously, it does not remove any either. We have thus shown the preservation of *is-a* relationships in GS. ■

Theorem 6. *MultiView is* **view independent**.

Proof: Theorems 4 and 5 show respectively that *MultiView* **preserves** the view classes and the view *is-a* relationships of all view schemata defined on GS through the restructuring of GS. By Definition 14, this proves the view independence of *MultiView*. ■

9 Related Work

Most initial proposals for defining views for OODBs suggest the use of the query language defined for their respective object model to derive a virtual class, e.g., [Kim89], [Heil90], [Kaul90], [Scho91], and [Abit91]. Most of them do not discuss the integration of derived classes into the global schema. Instead, the derived classes are treated as 'stand-alone' objects [Heil90] or they are attached directly as subclasses of the schema root [Kim89]. Scholl et al.'s recent work [Scho91] is one of the exceptions; they discuss the classification of virtual classes derived by the query language COOL into one schema. They do however not consider the

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

[6] This one-to-one mapping m is simply the equality operator on the class identifiers, since each class has a unique class identifier and VV $\subseteq$ V.

[7] We define the **type** of a class to be the union of its defined and its inherited property functions. Turning a defined property into an inherited property is not considered to be a change of the class type. Similarly, we define the set membership of a class, denoted by *content*(C) = $\{o \mid o \in C\}$, to be the union of its direct and indirect instances.

problem of generating multiple view schemata or of enforcing the consistency of the view schema.

Tanaka et al.'s work [Tana88] on schema virtualization does not distinguish between the task of integrating derived classes into a common schema and the task of generating view schemata. Also, they allow for the manual addition of *is-a* edges in a virtual schema, which may lead to an inconsistent schema, rather than supporting automatic view generation as done in *MultiView*. 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 summary, *MultiView* is a more systematic solution approach compared to their rather ad-hoc proposal.

Shilling and Sweeney [Shil89] extend the conventional concept of a class from having one type to having multiple interfaces. We accomplish the same goal by using the type refinement capability of the generalization hierarchy. Our work is simpler, since it does not require the extension of the traditional class concept. Furthermore, they approach the problem from the programming language point of view, and thus they do not handle the object instances associated with a class. 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 [Gilb90], similar to [Shil89], is also based on the idea of defining multiple interfaces for a class object. However, while our approach 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 [Gilb90].

10 Conclusions

In this paper, we have presented a simple yet powerful approach for supporting multiple view schemata in OODBs, called *MultiView*. *MultiView* allows for the customization of a view schema by virtually restructuring both the generalization and the property decomposition hierarchies of the global schema. In addition, we have defined an object algebra that can be used to customize the type structure and object membership of classes. We have also proposed an algorithm for integrating these derived classes into the global schema. *MultiView* provides support for view design by automating some tasks of the view specification process and by supplying automatic tools for enforcing the consistency of a view schema. For instance, we have presented an algorithm that not only verifies the closure property of a view schema but, if found incomplete, will transform the view schema into a minimal, yet closed, view. 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*.

Acknowledgements. I want to thank Lubomir Bic and Daniel D. Gajski for providing me with advice, encouragement, and financial support.

References

[Abit91] Abiteboul, S., and Bonner, A., "Objects and Views," in *Proc. SIGMOD*, May 1991, pp. 238 - 247.

[Bane87] Banerjee, J., Kim, W., Kim, H. J., and Korth, F., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," *Proc. of SIMOD*, May 1987, pp. 311- 322.

[Date90] Date, C. J., *An Introduction to Database Systems*, Vol. I, Fifth Ed., Addison-Wesley, 1990.

[Gilb90] Gilbert, J. P., "Supporting User Views", *Proc. OODB Task Group Workshop*, Canada, Oct. 1990.

[Heil90] Heiler, S., and Zdonik, S. B., "Object views: Extending the vision", in *Proc. IEEE Data Eng. Conf.*, Feb. 1990, pp. 86 - 93.

[Kaul90] Kaul, M., Drosten, K., and Neuhold, E.J., "ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views", in *Proc. IEEE Data Eng. Conf.*, Feb. 1990, pp. 2 - 10.

[Kim89] Kim, W., "A model of queries in object-oriented databases," in *Proc. Int. Conf. on Very Large Databases*, Aug. 1989, pp. 423 – 432.

[Rund92a] Rundensteiner, E. A., "*MultiView:* A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases", Univ. of Cal., Irvine, Tech. Rep. #92-07, Jan. 1992.

[Rund92b] Rundensteiner, E. A., and Bic, L., "Set Operations in Object-Based Data Models", in *IEEE Transaction on Data and Knowledge Eng.*, vol. 4, issue 3, June 1992.

[Rund92c] Rundensteiner, E. A. and Bic, L., "Automatic View Generation in Object-Oriented Databases", Univ. of Cal., Irvine, Tech. Rep. #92-15, Feb. 1992.

[Rund92d] Rundensteiner, E. A., "A Class Integration Algorithm and its Application For Supporting Consistent Object Views," Univ. of Cal., Irvine, Tech. Rep. #92-50, May 1992.

[Rund93] Rundensteiner, E. A., Bic, L., Gilbert, J., and Yin, M.-Y., "Set-Restricted Semantic Groupings," in *IEEE Trans. on Data and Knowledge Eng.*, to appear in April 1993.

[Schm83] Schmolze, J. G., and Lipkis, T. A., "Classification in the KL-ONE Knowledge Representation System," in *The Int. Joint Conf. on Artificial Intelligence*, Aug. 1983, vol.1, pp. 330 – 332.

[Scho91] Scholl, M. H., Laasch, C. and Tresch, M., "Updatable Views in Object-Oriented Databases," in *Proc. 2nd DOOD Conf.*, Germany, Dec. 1991.

[Shil89] Shilling, J. J., and Sweeney, P. F., "Three Steps to Views: Extending the Object-Oriented Paradigm," in *Proc. OOPSLA*, Sep. 1989, pp. 353 - 361.

[Tana88] Tanaka, K., Yoshikawa, M., and Ishihara, K., "Schema Virtualization in Object-Oriented Databases," in *Proc. IEEE Data Eng. Conf.*, Feb. 1988, pp. 23 – 30.

[Yu91] Yu and Osborn, "An Evaluation Framework for Algebraic Object-Oriented Query Models," in *Proc. IEEE Data Eng. Conf.*, Feb. 1991.