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
rundensteiner@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.

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 relationships between the virtual and the source classes, since this is required for solving MultiView's second task.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992
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 [Bare87].

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 identifiers\(^1\). Let \( T \) be the set of all types. For \( t \in T \), \( \text{properties}(t) \) corresponds to the set of property functions of \( t \) and \( \text{domain}_p(t) \) 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 \( C \in C \) 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 \( \text{type}(C) \) and to the set of property functions defined for \( C \) by \( \text{properties}(\text{type}(C)) \), or short, \( \text{properties}(C) \). If \( p \in P \) is a property function defined for \( C \), then we refer to the domain of \( p \) for \( C \) by \( \text{domain}_p(C) \). 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 \( \text{content}(C) = \{ o \mid o \in C \} \) with the predicate "in" defined based on object identities [Rund93].

**Definition 1.** For two classes \( C_1 \) and \( C_2 \in C \), \( C_1 \) is a subset of \( C_2 \), denoted by \( C_1 \subseteq C_2 \), if and only if \( (\forall o \in O) ((o \in C_1) \Rightarrow (o \in C_2)) \).

**Definition 2.** For two classes \( C_1 \) and \( C_2 \in C \), \( C_1 \) is a subtype of \( C_2 \), denoted by \( C_1 \subseteq C_2 \), if and only if \( (\forall p \in \text{properties}(C_1)) (\exists p \in \text{properties}(C_2)) (\text{domain}_p(C_1)) \subseteq \text{domain}_p(C_2) \).

**Definition 3.** For two classes \( C_1 \) and \( C_2 \in C \), \( C_1 \) is a subclass of \( C_2 \), denoted by \( C_1 \subseteq_a C_2 \), if and only if \( (C_1 \subseteq C_2) \) and \( (C_1 \subseteq C_2) \).

**Definition 4.** Let \( C_1 \) and \( C_2 \) be two classes with the types \( t_1 \) and \( t_2 \) in \( T \), respectively. Then \( t_1 \) is a function from \( T^2 \rightarrow T \) that defines a new type \( t_3 = t_1 \sqcup t_2 \). The property functions of \( t_3 \) are defined by \( \text{properties}(t_3) = \text{properties}(t_1) \cup \text{properties}(t_2) \). For each property function \( p \in \text{properties}(t_3) \), we define \( \text{domain}_p(t_3) = \text{domain}_p(t_1) \cap \text{domain}_p(t_2) \).

**Definition 5.** Let \( C_1 \) and \( C_2 \) be two classes with the types \( t_1 \) and \( t_2 \) in \( T \), respectively. Then \( \sqcap \) is a function from \( T^2 \rightarrow T \) that defines a new type \( t_3 = t_1 \sqcap t_2 \). The property functions of \( t_3 \) are defined by \( \text{properties}(t_3) = \text{properties}(t_1) \cap \text{properties}(t_2) \). For each property \( p \in \text{properties}(t_3) \), we define \( \text{domain}_p(t_3) = \text{domain}_p(t_1) \cup \text{domain}_p(t_2) \).

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

Let \( S = \{ C_i \mid i = 1, \ldots, n \} \) 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 \( (C_1 \subseteq_a C_n) \) and \( (C_n \subseteq_a C_1) \) and there are no other classes \( C_k \in S \) (with \( j=1, \ldots, m \)) for which the following is-a relationships hold: \( (C_1 \subseteq_a C_k) \) and \( (C_n \subseteq_a C_k) \) and \( (C_k \subseteq_a C_1) \) and \( (C_k \subseteq_a C_n) \). \( C_1 \) is called an indirect subclass of \( C_n \) and \( C_n \) an indirect superclass of \( C_1 \) if there are one or more classes \( C_k \in S \) for which the above is-a relationships hold. The direct subclass relationship is denoted by \( (C_1 \subseteq_a C_n) \) and the indirect one by \( (C_1 \subseteq_a^+ C_n) \).
Definition 6. An object schema is a directed acyclic graph $S = (V, E)$, where $V$ is a finite set of vertices and $E$ is a finite set of directed edges. Each element in $V$ corresponds to a class $C_i$, while $E$ corresponds to a binary relation on $V \times V$ that represents all direct $\text{is-a}$ 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 $\text{is-a}$ 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 $\text{is-a}$ relationships of a schema as the **generalization hierarchy**. A class is related to other classes via property relationships. For example, if $C_1$ has defined a property function $p$ with domain $\text{domain}_p(C_1) = C_2$, then we say that there is a property decomposition arc between $C_1$ and $C_2$ 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 $a = (C_1: C_2, l) \in A$ if and only if there is a property function defined for class $C_1$ with the property label $l$ and the domain class $C_2$.

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.

![Diagram of a class hierarchy](image)

**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 [Tana@Heilo], while the consistency of the generalization hierarchy is handled in [Rund92c]. Let the function $\text{Uses}(C)$ represent the set of classes that are used by $C$'s type interface. For example, if $p$ corresponds to an object pointer defined by domain(C)=C2, then $\text{Uses}(C)$ 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 $\text{Uses}: C \rightarrow 2^C$ is a function defined by: For $C_i, C_j \in C$, for $pk \in L$, $\text{Uses}(C_i) = \{C_j \in C | a_{ij} = \langle C_i, C_j, pk \rangle \in A \}$. For $S \subseteq C$, $\text{Uses}(S) = \bigcup_{C_i \in S} \text{Uses}(C_i)$. We define the closure operator $\ast$ by $\text{Uses}^\ast(C_i) = \bigcup_{j=1}^{\infty} \text{Uses}^j(C_i)$ with $\text{Uses}^0(C_i) = \text{Uses}^1(C_i) = \text{Uses}(C_i)$ and $\text{Uses}^j(C_i) = \text{Uses}(\text{Uses}^{j-1}(C_i))$ for $j > 1$.

$\text{Uses}(C_i)$ ($\text{Uses}^\ast(C_i)$) corresponds to the classes that are directly (directly or indirectly via transitive closure) used by the class $C_i$.

Definition 12. A view schema $VS=(VV, VE)$ is defined to be a closed view if the following holds: $VV = (\bigcup_{C_i \in VV} (\text{Uses}^\ast(C_i))) \cup VV$.

The closure criterion assures that all classes that are being used in a view (i.e., whose class names are visible in the $\text{Uses}^\ast$ set of a view class) are also defined within the view (i.e., they themselves are view classes).

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 $VS_1$ and $VS_2'$ are closed. The view $VS_2 = \{\text{Statenode2, Statetrans2}\}$ 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 $VS_2$.

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
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 “\(<virtual-class> = hide \langle prop-functions\> \text{ from } \langle source-class\>\)” 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 “\(<virtual-class> = refine \langle prop-function-defs\> \text{ for } \langle source-class\>\)” with \(<prop-function-defs\>\) 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 “\(<virtual-class> = select \text{ from } \langle source-class\> \text{ where } \langle predicate\>\)” 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

Figure 3: The MultiView Approach: From Base over Global to View Schemata.
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 source 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 Area, hence Comps2 ≤ 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 ⊆ 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)” derives FexLayout. Then extent(FexLayout) = extent(DataPathUnits) ∩ extent(RandomLogicUnits).
extent(RandomLogicUnits) = \{01,02\}. And type(FexLayout) = type(DataPathUnits).

In Figure 4, 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) = \{03,04,05\}. And type(AllOtherComps) = type(Components) = \{Get-Name, Comp-Type\}.
The relationship (AllOtherComps is-a Components) has been added to Figure 4.

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 | (VC is-a C_i) \land (\exists C_j \in V)(j \neq i)((VC is-a C_j) \land (C_j is-a C_i))\}. Similarly, we find all classes in G that are the direct subclasses of VC defined by direct-children(VC) = \{C_j | (C_j is-a VC) \land (\exists C_k \in V)(j \neq i)((C_k is-a C_j) \land (C_j is-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.

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 \subset 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 \subset 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 command, will modify only the 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 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 = (Uc EVV(U Sec(C;)) - VV is the minimal subset of classes from V that have to be added to the view VS to make it closed.

4We 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 = \( \bigcup_{C \in E} (U \setminus \{V\}) \) 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, \( VV = \bigcup_{C \in E} (U \setminus \{V\}) \). By subtracting the set VV from both sides of the equation, we derive \( \bigcup_{C \in E} (U \times V) = \emptyset \). This implies MIN = \( \bigcup_{C \in E} (U \setminus \{V\}) \). 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 U MIN. Then VV’ = VV U MIN = \( \bigcup_{C \in E} (U \setminus \{V\}) \). By subtracting the set VV from both sides of the equation, we derive \( \bigcup_{C \in E} (U \setminus \{V\}) = VV \). This implies MIN = \( \bigcup_{C \in E} (U \setminus \{V\}) \). Since VS is assumed to be closed, no classes need to be added to VS.

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, \( \bigcup_{C \in E} (U \times V) \), 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 \( \bigcup_{C \in E} (U \setminus \{V\}) \) must be added to VS.

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 Ci 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) \rightarrow class:
not-element(class.set-of-classes) \rightarrow 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.

return (set-of-classes,boolean-flag) is

CVG-done = 0; CVG-tmp = VV; Closed = true;
while (Ci=get-and-remove-next(CVG-tmp)) do
if (not-element(Ci, CVG-done)) 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) 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 VS 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 VS in Figure 2. CVG first initializes CVG-tmp=(Cl, 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 Ci. Uses(C1) = (C3). Therefore, the for-loop is executed but once with Ck = C3. The second if-statement evaluates to false, since (C3 ∈ VS). For the second while-loop iteration with Ci=C3, the first if-statement is again skipped. Uses(C3) = {C1}. The second if-statement evaluates to true, since (C1 ∈ VS). CVG terminates with (Closed=true) and (CVG-done=0); VS thus is closed.

Example 8. CVG is applied to the view VS 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 ∉ VS. Therefore, C5 is added to CVG-done Closed is set to false. Since Uses(C5) = (C2, C5), the for-loop has two iterations. For the second iteration with Ck=C5, the if-statement evaluates to true and C5 is added to CVG-tmp. For the fourth and last while loop iteration with Ci=C5, the first if-statement evaluates to true and C8 is added to CVG-tmp. For the fifth while-loop iteration with Ci=C8, the first if-statement is true and C8 is added to CVG-tmp. 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 = |VCevV(Uses*(Ci))| − VS.

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) ⇐⇒ (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. Thus, the two if-statements) used by CVG have constant complexity. Finally, we can show that all functions (and thus the two if-statements) used by CVG have constant complexity.

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 II: G* → G* be a function that applies a class derivation operator to GS and then restructures GS by integrating the resulting virtual class
into $GS$. 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 $\Pi$. i.e., $GS' = \Pi(GS)$ and $VS' = \Pi(VS)$.

(a) The view classes $VV$ of $VS$ are preserved through the application of the function $\Pi$ to $GS$ iff the following holds:

- $\exists$ a one-to-one mapping $m$: $C \rightarrow C'$ such that $(\forall C_1 \in C)((C_1 \in VV) \implies (\exists C_1' \in C') \in VV')$ and vice versa, $(\forall C_1 \in C')((C_1 \in VV') \implies (\exists C_1' \in VV) C_1 \equiv m^{-1}(C_1'))$.

- $(\forall C_i \in VV')((\forall \theta \in O) ((\theta \in C_i) \in VV \equiv (\theta \in m(C_i)) in VV')$.

(b) The view is-a relationships $VE$ for $VV$ are preserved through the application of the function $\Pi$ to $GS$ iff the following holds: $(\forall C_i, C_j \in VV) ((C_i \text{ is-a } * C_j) \in VE \iff (m(C_i) \text{ is-a } * m(C_j)) \in VE')$.

(c) The view $VS$ is preserved through the restructuring of $GS$ using the function $\Pi$ 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 $\Pi$ (Definition 14.a).

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 $C_i$ 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, $(\forall C_i \in \text{ direct-parents}(VC)) (\forall C_j \in \text{ direct-children}(VC)) (C_j \equiv * C_i)$. 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. 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 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 $\Pi$ (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., $(\forall C_i, C_j \in VV) ((C_i \equiv \text{ is-a } * C_j) \in VE \iff (m(C_i) \equiv \text{ is-a } * m(C_j)) \in VE)$.

**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], [Kan90], [Sch91], and [Ahit91]. 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 [Sch91] 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...
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