UC Irvine
ICS Technical Reports

Title
Automatic view schema generation in object-oriented databases

Permalink
https://escholarship.org/uc/item/9xj8434c

Authors
Rundensteiner, Elke A.
Bic, Lubomir

Publication Date
1992

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ARCHIVES
Z
699
C3
no.92-15

Automatic View Schema Generation in Object-Oriented Databases

Elke A. Rundensteiner and Lubomir Bic

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

Technical Report 92-15

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S. Code)

Automatic View Schema Generation in Object-Oriented Databases

ELKE A. RUNDENSTEINER and LUBOMIR BIC

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

Abstract

An object-oriented data schema is a complex structure of classes interrelated via generalization and property decomposition relationships. We define an *object-oriented view* to be a virtual schema graph with possibly restructured generalization and decomposition hierarchies – rather than just one individual *virtual class* as proposed in the literature. In this paper, we propose a methodology, called *MultiView*, for supporting multiple such *view schemata*. *MultiView* is anchored on the following complementary ideas: (a) the view definer derives virtual classes and then integrates them into *one* consistent global schema graph and (b) the view definer specifies arbitrarily complex view schemata on this augmented global schema. The focus of this paper is, however, on the second, less explored, issue. This part of the view definition is performed using the following two steps: (1) view class selection and (2) view schema graph generation. For the first, we have developed a view definition language that can be used by the view definer to specify the selection of the desired view classes from the global schema. For the second, we have developed two algorithms that automatically augment the set of selected view classes to generate a *complete*, *minimal* and *consistent* view class generalization hierarchy. The first algorithm has linear complexity but it assumes that the global schema graph is a tree. The second algorithm overcomes this restricting assumption and thus allows for multiple inheritance, but it does so at the cost of a higher complexity.

Index Terms: Automatic generation of view schemata, validity criteria for the view generalization hierarchy, object-oriented views, view definition language, schema design.

Contents

1 INTRODUCTION 1

2 BASIC CONCEPTS 3

2.1 The Object Data Model 3

2.2 Multiple View Schemata Concepts 6

2.3 The Validity of the View Generalization Hierarchy 7

3 THE *MultiView* METHODOLOGY 10

3.1 The Basic Philosophy 10

3.2 The Derivation of Virtual Classes 11

3.3 The Integration of Virtual Classes into the Global Schema 13

4 VIEW SCHEMA DEFINITION 15

4.1 Using A View Definition Language 15

4.2 View Schema Creation and Deletion Commands 17

4.3 View Schema Manipulation Commands 18

4.4 Examples of View Schema Definition 20

5 AUTOMATIC GENERATION OF A VALID VIEW SCHEMA HIERARCHY 21

5.1 Problem Definition 21

5.2 View Schema Generation For Global Schemata without Multiple Inheritance 22

5.3 View Schema Generation For Global Schemata with Multiple Inheritance 28

6 RELATED WORK 40

7 CONCLUSIONS 42

References 43

List of Figures

1 Examples of Base, Global and View Schemata. 7

2 View Schema Validity Example: Minimality, Completeness and Consistency Criteria. 9

3 The *MultiView* Approach: From Base over Global to View Schematas. 11

4 Using the **Select** Operator to Create the Virtual Class **Women**. 13

5 Integrating the Virtual Class **Women** Into the Global Schema. 14

6 From Base over One Integrated Global Schema To Multiple View Schemata. 16

7 The BNF Syntax Of the View Definition Language. 17

8 The Virtual Schema Creation Algorithm A1. 23

9 Example Snapshots For Tracing of the A1 Algorithm. 24

10 Redundant and Required Edges. 27

11 An Example of Creating Redundant View *Is-A* Arcs. 29

12 Algorithm A2 for Removal of Redundant Arcs. 30

13 An Example of Removing Redundant Edges Using Algorithm A2. 31

14 View Schema Creation Algorithm A3. 34

15 View Schema Creation Algorithm A4. 36

16 Example of the Algorithm A4 for Creating A Valid Schema. 37

1 INTRODUCTION

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

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

Note that we cannot simply modify the global object schema so that it suits the requirements of one particular user. Instead, we need to support a number of *different, potentially conflicting, view schemata* of the same data model, each of which supporting a particular user's point of view. Consequently, we are concerned here with the virtual restructuring for each given view while maintaining all other view schemata; rather than with permanently changing the global database as is done in schema evolution [2].

These complete (possibly conflicting) view schemata have to be integrated with one another and with the underlying global schema into one consistent whole. This integration has to maintain the difference in the generalization and decomposition hierarchies of the view schemata. The proposed *MultiView* methodology solves this problem by breaking it into two independent steps: (1) the derivation of virtual classes via a query and their integration into *one* consistent global schema graph and (2) the definition of view schemata composed of both base and virtual classes on top of this augmented global schema. An additional requirement is that the originally specified object schema (with stored rather than derived classes) remains intact so that it can be used by other users,

if so desired. The *MultiView* methodology accomplishes this by treating the original base schema as a special non-modifiable view schema. Some of the functionalities that *MultiView* supports are the following: (1) virtual modification of the type structure and of the object membership of existing classes, (2) sharing of property functions and object instances among stored and derived classes without unnecessary duplication, (3) virtual restructuring of the generalization and the property decomposition hierarchy, (4) sharing of classes, property functions, and objects among different view schemata, (5) construction of an arbitrarily complex view schema as required by a particular user task, and (6) integration of each view schema with all other schemata into one 'consistent whole'.

The generalization hierarchy of a view schema, i.e., the *is-a* relationships among the classes of a view schema, has to be consistent with the semantics of the respective classes, i.e., with their subset and subtype relationships. Inserting arbitrary *is-a* relationships between view classes in a view schema may result in an inconsistent schema. For instance, the view definer may assert an *is-a* arc between two classes in the view schema that are not *is-a* related in the global schema. Then the view schema would imply an incorrect property inheritance and subset relationships among these two classes (and possibly also their subclasses). We define a *validity* criterion for view schemata in terms of the *completeness*, *minimality* and *consistency* properties of the view schema class hierarchy. This new concept allows for the identification of inconsistencies between the generalization hierarchies of the global and the view schemata.

Rather than requiring manual entry of view *is-a* arcs by the view definer and then checking the entered information for validity, we develop the automatic generation of the view schema hierarchy as a more desirable option. Automatic view generation will not only prevent the introduction of errors into the view schema, but also simplify the task of the view definer. *MultiView* therefore supports automatic view generation. In particular, we present two algorithms that automatically augment the set of selected view classes to generate a *valid* view schema hierarchy. The first algorithm has linear complexity but it assumes that the global schema graph is a tree. The second algorithm overcomes this restricting assumption and thus allows for multiple inheritance, but it does so at the cost of a higher complexity. We show the correctness of both algorithms.

In summary, this paper makes the following contributions. First, we extend the concept of an object-oriented view from an individual *virtual class* to a complete *view schema.* This requires the introduction of new concepts, such as, the *validity* of a view schema, which represents a step towards the development of an object-oriented database theory. Second, we present a general methodology for supporting *multiple (possible contradicting) view schemata* in OODBs, called *MultiView. MultiView* supports all of the above mentioned functionalities. Third, we present

solutions to some of the subtasks related to the proposed view paradigm. In particular, we have developed a language for view schema definition and two efficient algorithms for the automatic generation of the view schema hierarchy.

The paper is organized as follows. In Section 2, we introduce object-oriented concepts required for supporting multiple view schemata. In Section 3, we outline the *MultiView* paradigm and sketch solutions for accomplishing the first phase of the approach. The view definition language and the algorithms for generating the view class hierarchy are given in Sections 4 and 5, respectively. We compare *MultiView* to related work in Section 6 and conclude with Section 7.

2 BASIC CONCEPTS

2.1 The Object Data Model

Below, we introduce the basic concepts of OODB models needed for the remainder of the paper. Let O be an infinite set of object instances. Each element o ∈ O is an instance of an abstract data type (ADT), i.e., it can be manipulated only by means of the interface of the respective ADT. Let P be an infinite set of property functions. Each property function p ∈ P can be a value from a predefined enumeration type, an object instance from some class, or an arbitrarily complex function. Each property function p ∈ P has a name and signature (i.e., domain types). Let C be the set of all classes. A **class** $C_i \in$ C has a unique class name, a type description and a set membership. The type associated with a class corresponds to a common interface for all instances of the class, that is, the collection of applicable property functions. We refer to the name of the type associated with a class C by **type**(C) and to the set of property functions defined for C by **properties**(C). If p ∈ P is a property function defined for C, i.e., p ∈ **properties**(C), then we refer to the domain of the property function p by $\mathbf{domain}_p$(C). A **class** is also a container for a set of objects. The collection of objects that belong to a **class** C is denoted by **extent**(C) := $\{o \mid o \in C\}$ with the member-of predicate "∈" defined based on the object identities of the object instances [12]. We can now define the following relationships between classes.

Definition 1. *For two classes C1 and C2 ∈ C, C1 is called a* **subset** *of C2, denoted by C1 ⊆ C2, if and only if* $(\forall\ o \in O)\ ((o \in C1) \Longrightarrow (o \in C2))$.

Definition 2. *For two classes C1 and C2 $\in$ C, C1 is called a* **subtype** *of C2, denoted by C1 $\preceq$ C2, if and only if (***properties***(C1) $\supseteq$ **properties***(C2)) and ($\forall$ p $\in$ **properties**(C2)) (**domain**$_p$(C2) $\supseteq$ **domain**$_n$(C1)).*

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

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

Informally, we say that C1 is *is-a* related to C2 (denoted by C1 *is-a* C2) if (1) every member of C1 is a member of C2 (the subset relationship) and (2) every property defined for C2 is also defined for C1 (the subtype relationship).

The three types of class relationships are *reflexive, antisymmetric* and *transitive.* These three properties mean for instance the following for the *is-a* relationship. By reflexivity, the *is-a* relationship (C_i *is-a* C_i) holds for all C_i. By antisymmetry, the *is-a* relationships (C_i *is-a* C_j) and (C_j *is-a* C_i) imply (C_i=C_j). By transitivity, the *is-a* relationships (C_i *is-a* C_j) and (C_j *is-a* C_k) imply (C_i *is-a* C_k).

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

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

more classes $C_{k_j} \in S$ (with j= 1,2, ..., m) for which the above *is-a* relationships hold. This *indirect subclass* relationship between C_1 and C_n is denoted by (C_1 *is-a** C_n) for (j≥0) and by (C_1 *is-a+* C_n) for (j≥1). A graph-theoretic representation of a set of classes S that explicitly represents all *direct subclass* relationships among the classes in terms of edges is defined below.

Definition 4. *A* **(generalization hierarchy) schema** *is a directed acyclic graph*[1] *S=(V,E), where V is a finite set of vertices and E is a finite set of directed edges. Each element in V corresponds to a class C_i, while E corresponds to a binary relation on $V \times V$ 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).*

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

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

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

---

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

2.2 Multiple View Schemata Concepts

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

Definition 5. *The* **base schema** *(BS) is a (generalization hierarchy) schema S=(V,E), where all nodes in V correspond to base classes with stored rather than derived object instances.*

Definition 6. *Let BS be a* **base schema**. *The* **global schema** *(GS) is an extension of the base schema that is augmented by the collection of all virtual classes defined during the lifetime of the database as well as is-a relationships among this extended set of classes.*

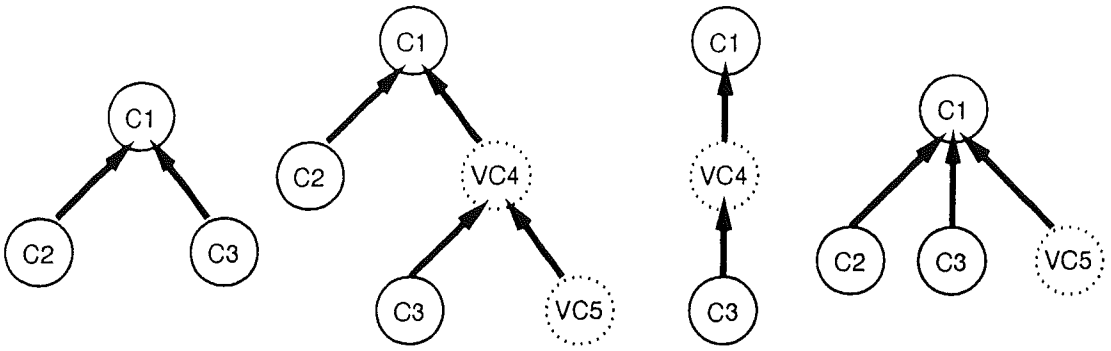
A subgraph of the global schema which contains only virtual classes and their *is-a* relationships is commonly called a *virtual schema* [17].

Definition 7. *Given a global schema GS=(V,E), then a* **view schema** *(VS), or short, a* **view**, *is defined to be a schema VS= (VV, VE) with the following properties:*

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

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

is a special case of a schema. Therefore all properties of a general schema defined in Section 2.1. must also hold. We call the classes in a view schema (both the base and the virtual ones) *view classes* and the *is-a* relationships among these view classes *view is-a relationships.*



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

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

Example 1. *Figure 1 shows the relationship between (a) the base schema BS, (b) the global schema GS, and (c) and (d) two different view schemata VS1 and VS2. We depict base and virtual classes by circles and dotted circles, respectively. The global schema GS in Figure 1.b is derived from the base schema BS in Figure 1.a by deriving the virtual classes VC4 and VC5 and by interconnecting them with the remaining classes in GS to create a valid schema. The view schemata in Figure 1.c and 1.d are derived from GS by selecting a subset of its classes and interconnected them into a valid schema using view is-a arcs.*

Note that the base schema is a special case of a view schema that consists exclusively of all base classes and no virtual classes. We will maintain the base schema as a view schema, i.e., there will be a view object table (or base table) that lists all base classes and their *is-a* relationships (See [12]). This is important so that users of the data model can see the original data model of the application domain without having to consider derived information. This base table is a special view table in as much as it is predefined and not modifiable.

2.3 The Validity of the View Generalization Hierarchy

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

Definition 8. *Given a view schema VS=(VV,VE) defined on the global schema GS=(V,E). For all classes C_1,C_2 in VV, an is-a arc from source C_1 to sink C_2 is* **required** *in VS, if (C_1 is-a* C_2) in GS and there is no C_x in VV such that (C_1 is-a* C_x) in GS and (C_x is-a* C_2) in GS. The view VS is* **complete**, *if the set VE of all its view is-a relationships contains all required arcs in VS.*

This defines the completeness criterion of *is-a* arcs as follows: if two classes C_1 and C_2 in VS are *is-a* related in GS then they also have to be *is-a* related in VS. If there is no indirect path of length greater than one between C_1 and C_2 in VS (such that, C_1 *is-a+* C_2 in VS), then the edge (C_1 *is-a* C_2) is *required* in VS.

Definition 9. *Given a view schema VS=(VV,VE) defined on the global schema GS=(V,E). For all classes C_1,C_2 in VV, an is-a arc from source C_1 to sink C_2 is* **redundant** *in VS, if there is a class C_x in VV such that (C_1 is-a* C_x) in GS and (C_x is-a* C_2) in GS. The view VS is* **minimal**, *if none of the view is-a arcs in the set VE is redundant in VS.*

This defines the minimality criterion of *is-a* arcs as follows: if there is an indirect *is-a* path (i.e., a path of length greater or equal to two) between two classes then there should not also be a direct *is-a* arc between them. Stated differently, an *is-a* arc from source C_1 to sink C_2 is *redundant* in VS if there also is an *is-a* arc path of length greater or equal to two from C_1 to C_2 in VS (i.e., C_1 *is-a+* C_2).

Definition 10. *Given a view schema VS=(VV,VE) defined on the global schema GS=(V,E). For all classes C_1,C_2 in VV, an is-a arc from source C_1 to sink C_2 in VS is* **incompatible** *if the edge <C_1,C_2> is in VE and not(C_1 is-a* C_2) in GS. The view VS is* **consistent**, *if none of its view is-a arcs in the set VE is incompatible.*

This defines the consistency criterion of *is-a* arcs as follows: an *is-a* arc from source C_1 to sink C_2 can exist in *VS* if and only if the two classes are *is-a* related in GS. In other words, it indicates the following equivalence relationship for all classes C_1,C_2 in VV: (C_1 *is-a** C_2) in VS $\Longleftrightarrow$ (C_1 *is-a** C_2) in GS. Note that this consistency criterion is a direct consequence of the basic definition of a view schema (Definition 7). The third requirement 'VE $\subseteq$ transitive-closure(E)' implies the consistency of the view schema, i.e., it implies that all *is-a* relationships in *VS* must be directly derivable from the *is-a* relationships in *GS*.

Definition 11. *A view schema VS=(VV, VE) for a given global schema GS=(V, E) is* **is-a valid** *(or* **valid***) if the set of all view is-a relationships VE among its view classes VV is complete and minimal and consistent.*

This definition states that a valid view schema *VS=(VV, VE)* contains *all* required *is-a* relationships and *no* redundant or incompatible *is-a* relationships. We demonstrate the concepts introduced in this section with the example given in Figure 2.

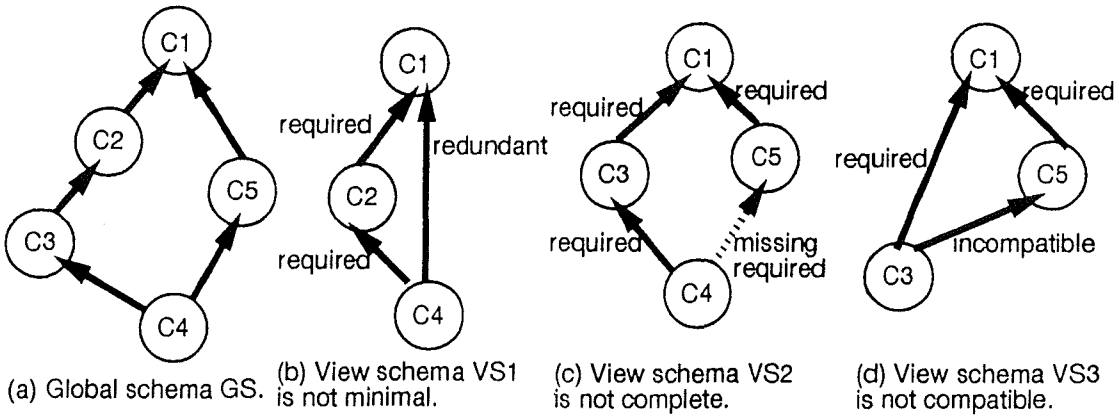


Figure 2: View Schema Validity Example: Minimality, Completeness and Consistency Criteria.

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

Other requirements for the validity of a view schema, e.g., type closure, are not directly relevant to our work and therefore are omitted in this paper [12, 5, 17].

3 THE *MultiView* METHODOLOGY

3.1 The Basic Philosophy

In this section, we outline our approach for supporting *multiple view schemata* in OODBs, called the *MultiView* paradigm. *MultiView* is anchored on the following complementary ideas: (1) the derivation of virtual derived classes and their integration into *one* consistent global schema graph and (2) the definition of view schemata composed of both base and virtual classes in terms of the augmented global schema. The first phase supports the virtual customization of existing classes by deriving new virtual classes with a possibly modified type description and membership extent. For this we assume that virtual classes are derived from the global schema using some object-oriented queries (e.g., see [7, 5, 14]). This fulfills the first requirement for a view support system listed in Section 1. The integration of the virtual classes into the underlying global schema takes care of the second requirement, namely, the maintenance of explicit relationships between stored and derived classes in terms of type inheritance and subset relationships. This is useful for sharing property functions and object instances consistently among classes without unnecessary duplication. It also is a necessary basis for the second phase of *MultiView*, namely, for the formation of arbitrarily complex view schema graphs. If the virtual classes are not integrated with the classes in the global schema, then a view schema would correspond to a collection of possibly 'unrelated' classes rather than a generalization schema graph as defined in Definition 7.

The second phase of *MultiView* utilizes this augmented global schema graph for selection of both base and virtual classes and for arranging these view classes in a consistent class hierarchy, called a *view schema*. This phase handles all remaining requirements for a view support system listed in Section 1. It supports for instance the virtual restructuring of the *is-a* hierarchy by allowing to hide from and to expose classes within a view schema. For the explicit selection of view classes from the global schema, we have developed a view schema definition language that can be used by the view definer to specify the classes required for a particular view schema (see Section 4). Note that the *is-a* relationships among the set of selected view classes of a view schema are dictated by their subset and subtype relationships as defined in Section 2. Inserting arbitrary *is-a* relationships between classes in a view schema may result in an incorrect schema in terms of property inheritance and subset relationships. Therefore, rather than requiring the manual insertion of *view is-a* arcs by the view definer, we have developed algorithms that automatically augment the set of selected view classes to generate a *valid* view schema class hierarchy. The first algorithm has linear complexity but it assumes that the global schema graph is a tree. The second algorithm overcomes this restricting assumption and thus allows for multiple inheritance, but it does so at the cost of a higher complexity. These algorithms are presented in Section 5.

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

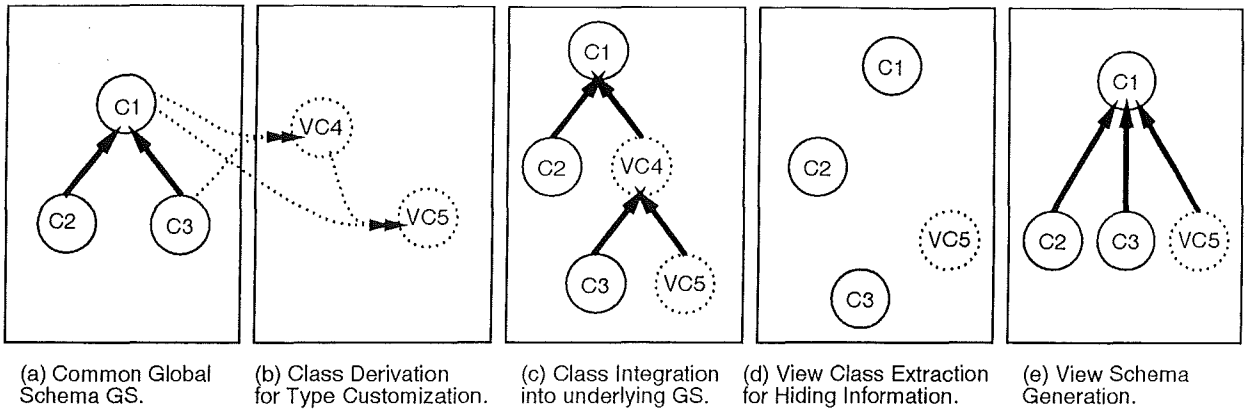


Figure 3: The *MultiView* Approach: From Base over Global to View Schematas.

Example 3. *This example of the view schema construction process is based on Figure 3. In this figure we depict base and virtual classes by circles and dotted circles, respectively. Given the global schema GS in Figure 3.a, the view definer first specifies one or more virtual classes using some object-oriented query, e.g., the two classes VC4 and VC5 shown in Figure 3.b. Class VC4, for instance, is derived based on the two source classes C1 and C3 as depicted by the dotted arrows pointing from Figure 3.a to Figure 3.b. The integration of the virtual classes VC4 and VC5 into GS is given in Figure 3.c. View schema definition now proceeds by selecting a subset of classes from the augmented schema GS. As depicted in Figure 3.d, the selected view classes can be both base and virtual classes. Lastly, the chosen view classes are interconnected into one schema graph. The resulting virtual schema graph, called a view schema, is given in Figure 3.e.*

Below we outline the basic ideas underlying the first part of the *MultiView* paradigm. The goal here is not to present a complete treatment of this subject (which is beyond the scope of this paper), but rather to explain the basic concepts. A detailed treatment of these issues can be found in [12].

3.2 The Derivation of Virtual Classes

MultiView uses class derivation mechanisms for a number of different purposes, such as, to customize type descriptions, to limit the access to property functions, to collect object instances into groups meaningful for the task at hand, and so on. For this we assume an object-oriented query language that can be used by the view definer to derive arbitrarily complex virtual classes. This part of the

MultiView paradigm is in sync with the work presented in the literature [7, 5, 14]. Examples of typical operators proposed in the literature are **selection**, **projection**, **set operations**, etc. The result of such a class derivation is a virtual class VC that has a derived type description and a derived membership extent (See Section 2 for definitions of these concepts). The **Select** operator, for instance, similar to the selection operator defined for relational algebra [3], has the following 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. More formally stated,

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

The extent of the virtual class derived by selection is a subset of the extent of the source class; this subset relationship is denoted by <*virtual*-class> ⊆ <*source*-class> (Definition 1). The type description of the derived selection class is equal to the type of the source class, i.e.,

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

By default, this implies the subtype relationship <*virtual*-class> ≼ <*source*-class> (Definition 2). From these two class relationships we can deduce the *is-a* relationship <*virtual*-class> *is-a* <*source*-class> (Definition 3). Next we present an example of applying the **Select** operator for class derivation.

Example 4. *In Figure 4, the virtual class* **Women** *is derived by the query "***Women** := **select from** *(***People***)* **where** *(Sex="female")". Women contains a subset of the object instances that are members of the source class* **People**, *i.e.,* **Women** ⊆ **People**. *Furthermore,* **Women** *inherits the type description from its source class* **People**, *while constraining the domain of one of its property functions, namely, the Sex property. Hence the subtype relationship* **Women** ≼ **People** *holds. From these two class relationships we can conclude that (***Women** *is-a* **People***).*

While **selection** is an example of a set-manipulating operator, there are also operators that work on the type description of a class. The **hide** operator, for instance, modifies the type description of a class by hiding (projecting) some of its property functions; it is similar to the **project** operator in the classical relational algebra. **Refine** is another type-manipulating operator that

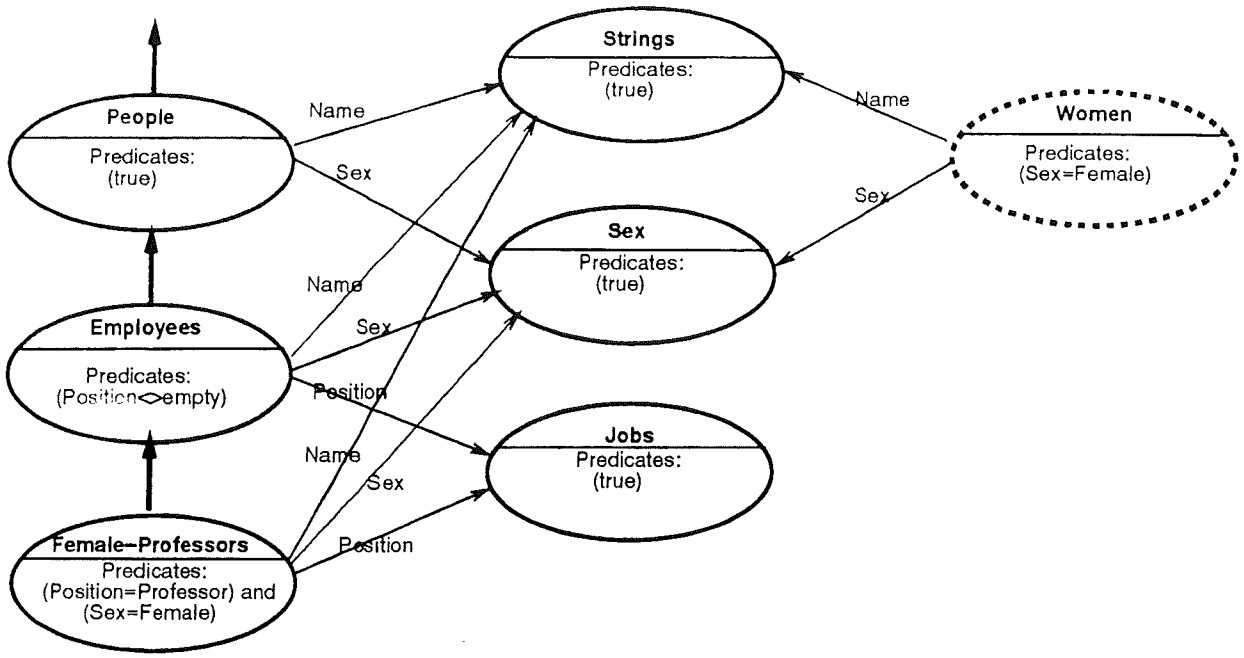


Figure 4: Using the **Select** Operator to Create the Virtual Class **Women**.

adds additional property functions to a type. It is similar in flavor to calculating a derived value for each tuple of a relation and then adding (joining) this derived value to the relation as an additional column. Other examples are set operators, such as, **union**, **difference** and **intersect**, which modify both the type description and the set membership of their source classes. A detailed analysis of these set operators for OODBs can be found in [11].

3.3 The Integration of Virtual Classes into the Global Schema

For reasons explained in the beginning of this section, the *MultiView* paradigm will integrate the virtual classes into the global schema. Algorithms for special classification subproblems have been proposed in the literature. For instance, Schmolze and Lipkis [13] describe a classifier for 'concepts' in the KL-ONE Knowledge Representation System. Scholl et al. [14] sketch the class integration process for a selected subset of the operators of the query language COOL. In [11], we describe the integration of virtual classes derived using set operations into the underlying schema graph. In this paper we sketch an overall approach for the class integration problem. A detailed treatment of this problem is, however, beyond the scope of this paper and can be found in [12].

Class integration is concerned with finding the most appropriate location in the schema graph for a given virtual class. We exploit the *subtype*, *subset* and *is-a* relationships between the virtual

class and the classes in the global schema to solve this classification problem. The classifier determines the *is-a* relationships between the virtual class VC and all other classes in the global schema by comparing both their type descriptions and their membership predicates. This comparison then deduces the correct location of VC by placing VC between its most direct sub- and superclasses. The basic idea for finding the correct position for the class VC in the schema G=(V,E) can be summarized as follows. First, we find all classes in G that subsume VC, i.e., they are the direct superclasses of VC defined by parents(VC) := $\{C_i \mid (VC \textit{ is-a } C_i) \wedge (\nexists C_j \in V)(j \neq i)((VC \textit{ is-a } C_j) \wedge (C_j \textit{ is-a } C_i))\}$. Similarly, we find all classes in G that VC subsumes, i.e., they are the direct subclasses of VC defined by children(VC) := $\{C_i \mid (C_i \textit{ is-a } VC) \wedge (\nexists C_j \in V)(j \neq i)((C_i \textit{ is-a } C_j) \wedge (C_j \textit{ is-a } VC))\}$. VC then is placed directly below all classes in the parents set and directly above all classes in the children set. In general, the classification problem is not decidable since it may involve the comparison of arbitrary functions and predicates. In the worst case, if some *is-a* relationship is not discovered, then this means that the virtual class is placed higher in the class hierarchy than would theoretically be possible. This would still be a correct but possibly not the most informative class arrangement.

This process can be fine-tuned for each query operator. This would allow us to limit the search to a small portion of the global schema based on the semantics of the operator and the position of the respective source classes [12]. Rather than presenting detailed classification algorithms here, we demonstrate this process on an example.

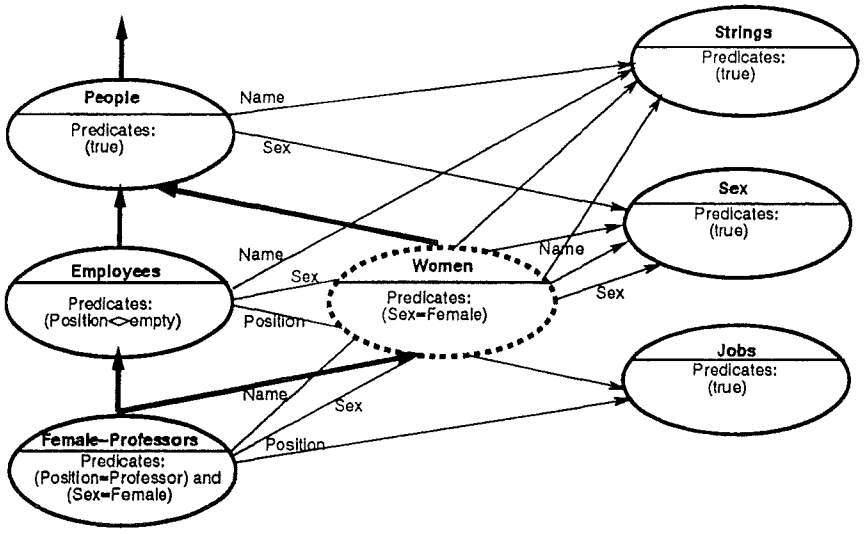


Figure 5: Integrating the Virtual Class **Women** Into the Global Schema.

Example 5. *In this example we explain how the virtual class* **Women** *derived via the* **select** *operator is integrated into the global schema in Figure 5. As discussed in Example 4, the class relationship*

*(***Women** *is-a* **People***) can be derived directly from the semantics of the* **Select** *operator. We therefore insert the edge (***Women** *is-a* **People***) into the global schema as done in Figure 5. The classification process is however not complete. Instead, we now traverse the schema graph downwards from the source class to find the most specialized classes - lowest in the is-a hierarchy - that are still is-a related with the* **Women** *class. Since the* **Employees** *class has a more refined type definition than the* **Women** *class, the type relationship (***Employees** $\preceq$ **Women***) holds. We can however not determine any subset relationship between these two classes. Hence, neither (***Women** *is-a* **Employees***) nor (***Employees** *is-a* **Women***) is true. The type relationship (***Female-Professor** $\preceq$ **Women***) holds, because the* **Female-Professor** *class inherits the additional property function 'Position' from the* **Employees** *class. We can also establish a subset relationship between these two classes based on their associated predicates. Namely, the predicate "(Sex=Female)" of the* **Women** *class clearly subsumes the predicate "(Sex=Female) and (Position=Professor)" of the* **Female-Professor** *class. Therefore we can infer the subset relationship (***Female-Professor** $\subseteq$ **Women***). By Definition 3, (***Female-Professor** $\preceq$ **Women***) and (***Female-Professor** $\subseteq$ **Women***) imply (***Female-Professor** *is-a* **Women***). Therefore we have added an is-a edge between the two classes as depicted in Figure 5.*

4 VIEW SCHEMA DEFINITION

4.1 Using A View Definition Language

As explained in Section 3, the first phase of *MultiView* works on the global schema, i.e., view class derivation moves from the base schema to a more and more complex global schema. An example of this can be seen in the transition from Figure 6.a to Figure 6.b and from Figure 6.b to Figure 6.c. The second phase of *MultiView* extracts a view schema from the global schema, i.e., view schemata subsetting moves from the global schema to one view schema. An example of this view extraction can be seen from Figure 6.b to Figure 6.d and from Figure 6.c to Figure 6.e. In this section, we assume that the first phase of *MultiView*, namely, the definition of virtual classes and their integration into one underlying global schema, has been taken care of. For the following, we are concerned with the second phase of *MultiView*, namely, the definition of a view schema on top of the augmented global schema. For this we define a view definition language that can be utilized by the view definer for the specification of view schemata.

In Figure 7, we present the BNF syntax of this view definition language. Note that the syntax for particular class derivation operators, which are used during the first phase of *MultiView*, is not further specified. Recall that *MultiView*, and of course also the view definition language, is

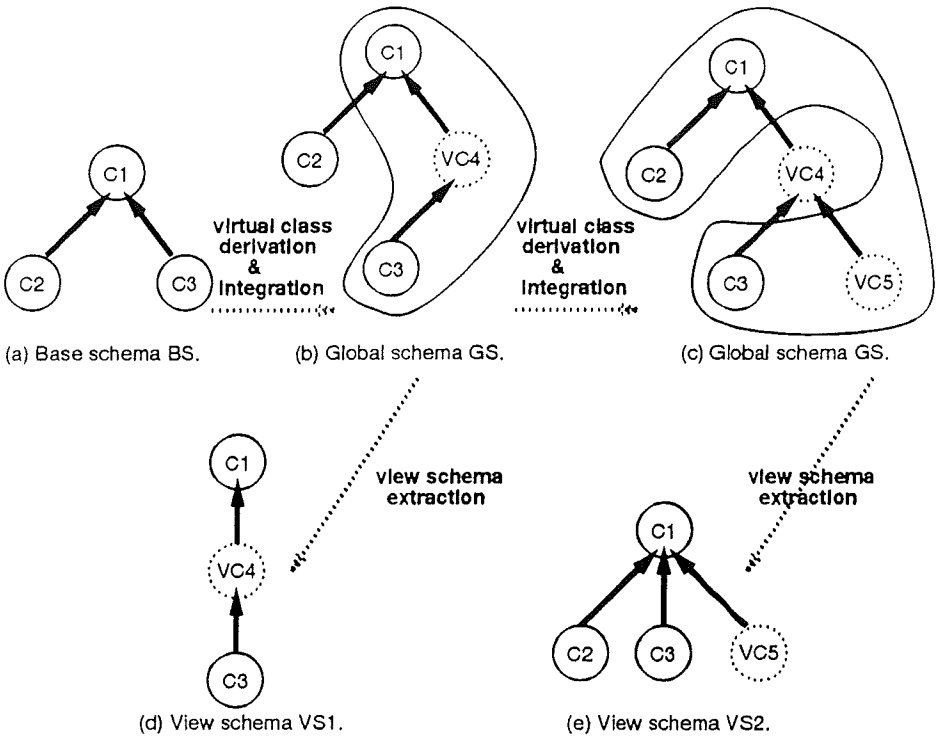


Figure 6: From Base over One Integrated Global Schema To Multiple View Schemata.

independent of the choice of particular query operators. We also want to stress that phases one and two of *MultiView* do not have to be executed sequentially, i.e., they can be intermixed. For instance, during the process of creating a particular view schema, the view definers may decide that they need additional virtual classes that are not yet available in the global schema. They then can use a class derivation operator from the first phase of *MultiView* to augment the global schema with the necessary virtual class before continuing the definition of the view schema. Note that the view definition language is concerned only with the manipulation of view classes and not with view *is-a* relationships. Rather than specifying *is-a* arcs manually, *MultiView* will automatically generate the set of view *is-a* arcs that has to be inserted in order to make the view schema *valid.* This is the topic of Section 5.

We divide the discussion of the view definition language into two parts. In Section 4.2, we describe the more general operators that manipulate complete view schemata, while in Section 4.3 we describe the operators used for the definition of a particular view schema. The second operator type is concerned with modifying a particular, possibly empty, view schema by adding and/or deleting view classes.

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

Figure 7: The BNF Syntax Of the View Definition Language.

4.2 View Schema Creation and Deletion Commands

The following operators work on a complete view schema by either initiating or terminating a transaction on a particular view schema:

1. the **DEFINE-VIEW** command creates a new view schema;
2. the **MODIFY-VIEW** command modifies an existing view schema;
3. the **SAVE-VIEW** command saves a view schema;
4. the **DELETE-VIEW** command deletes an existing view schema; and
5. the **END-VIEW** command terminates a transaction on a view schema.

The first operation, the **DEFINE-VIEW** command, initializes a new empty view schema and assigns a unique view identifier to it. This operation is executed prior to any other operators. Hence, the creation of virtual classes or the modification of a view schema VS can be done only in the context of a view definition transaction of the particular view schema. Within this transaction,

which is marked by a DEFINE-VIEW command at the beginning and the END-VIEW command at the end, changes can be made to this one view schema only. Furthermore, the system keeps track of all virtual classes created by this view schema.

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

Once the view definers want to conclude the view definition phase, they issue the SAVE-VIEW command. This command establishes a view table for the view schema which lists all classes that are part of this view [12]. In addition, the system determines the set of view *is-a* arcs that have to be inserted into this view schema and of course also into the view table. This is the topic of Section 5.

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

4.3 View Schema Manipulation Commands

Once an appropriate command (which is either the DEFINE-VIEW or the MODIFY-VIEW command) has been issued to allow for transactions on a particular view schema VS to take place, then the commands specified in this section can be used to manipulate VS. This definition of a particular view schema proceeds using the following two types of steps: (1) the creation of virtual classes and their integration into the global schema and (2) the insertion of view classes into and deletion of view classes from the view schema VS. Note that the former assumes that the virtual class is also automatically added to the current view. The operators for the former are discussed in Section 3, while the commands for the latter are given next:

1. the "ADD-CLASS<*class*-name>" command;

2. the "ADD-CLASS-DAG<*class*-name>" command;

3. the "ADD-VIEW-SCHEMA<*view*-name>" command;

4. the "REMOVE-CLASS <*class*-name>" command;

5. the "REMOVE-CLASS-DAG <*class*-name>" command;

6. the "REMOVE-VIEW-SCHEMA <*view*-name>" command; and

7. the "RENAME-CLASS <*old-class*-name> by <*new-class*-name>" command:

The semantics of these view definition language commands are straightforward. They assume that a view schema VS has already been created and opened for manipulation by either a DEFINE-VIEW or a MODIFY-VIEW command. They then modify this designated view VS by either adding to or deleting from the schema. The "ADD-CLASS(<*class*-name>)" command adds a class with the name <*class*-name> in GS to the view schema VS. The "ADD-CLASS-DAG(<*class*-name>)" command adds all classes to the view schema VS that are classes in the subschema of GS rooted at the class with the name <*class*-name>. Finally, the "ADD-VIEW-SCHEMA <*view*-name>" command adds all classes of the view schema with the view identifier <*view*-name> to the current view schema VS.

The next three commands do the same as the ones above but rather than adding they are deleting the respective classes. The "REMOVE-CLASS <*class*-name>" command removes the class with the name <*class*-name> from VS. Recall that if a virtual class is created during a transaction of modifying the view schema VS, then it is automatically inserted into the view schema VS. If during the view creation process this virtual class is removed from the view schema VS, then the class is also deleted from the global schema. The "REMOVE-CLASS-DAG <*class*-name>" command removes the subschema of GS rooted at the class with the name <*class*-name> from VS. The "REMOVE-VIEW-SCHEMA <*view*-name>" command removes the existing view schema with the identifier <*view*-name> from VS.

Lastly, the "RENAME-CLASS <*old-class*-name> by <*new-class*-name>" command renames an existing view class of the view schema VS by replacing its name <*old-class*-name> by the new name <*new-class*-name>. We assume scoping here; hence this is a local change that is only visible from within the current view schema.

4.4 Examples of View Schema Definition

Below, we demonstrate the above mentioned commands of the view definition language based on the example views shown in Figure 6.

Example 6. *In this example, we discuss the definition of the view schema VS1 in Figure 6.d on top of the global schema GS depicted in Figure 6.a. Note that the global schema GS is equal to the base schema BS of this application domain, since no virtual classes have been added yet to this schema. There are a number of different view creation commands that could be used to accomplish the definition of this view schema. Below, we give one possible view creation script for VS1.*

View Creation Script For VS1:

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

We start the view definition transaction by issuing the DEFINE-VIEW VS1 command, which creates an empty view schema with the identifier VS1. We then define and insert the virtual class VC4 into the global schema GS. As discussed above, VC4 is also automatically added to the view schema VS1, i.e., we now have classes(VS1) = { VC4 }. Then the commands ADD-CLASS*(C1) and* ADD-CLASS*(C3) are issued to insert the classes C1 and C3 into the current view schema VS1. VS1 now has the classes(VS1) = { VC4, C1, C3 }. Lastly, VS1 is saved with the command* SAVE-VIEW*. MultiView then automatically creates the view is-a arcs for VS1 as described in Section 5. The result of this view generalization hierarchy creation is shown in Figure 6.d.*

Example 7. *The second view schema VS2 in Figure 6.e is defined on top of the global schema GS depicted in Figure 6.b. A possible view creation script for VS2 is given below.*

View Creation Script For VS2:

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

Again, we start the view definition transaction by issuing the DEFINE-VIEW VS2 command, which creates an empty view schema with the identifier VS2. Then the virtual class VC5 is defined

and added to the global schema GS. Then the three classes { C1, C2, C3 } are added to VS2, i.e., we now have classes(VS2) = { VC5, C1, C2, C3 }. This could be done by either issuing three ADD-CLASS *commands, or equivalently, we can add the base schema BS to VS2 using the command* ADD-VIEW-SCHEMA(BS), *since the base schema BS is composed of exactly the three desired classes. When VS2 is saved, the is-a arcs shown in Figure 6.d are derived automatically by MultiView using algorithms described in Section 5.*

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

5 AUTOMATIC GENERATION OF A VALID VIEW SCHEMA HIERARCHY

5.1 Problem Definition

As explained in Section 3, we automate the specification of the view schema class hierarchy rather than requiring manual entry of the *view is-a* arcs by the view definer. Automatic view generation not only simplifies the task of the view definer, but it also prevents the introduction of errors into the view schema. We identify three types of errors. First, the view definer may omit *is-a* arcs that are *required* to describe the complete semantics of the view schema (Definition 8). This would leave some of the type inheritance that is actually taking place in the view schema unexposed. Second, the view definer may insert *redundant is-a* arcs. This violates the minimality requirement of a view schema 9, which would for instance obscure the property inheritance among subclasses. Third, the view definer may assert incorrect *is-a* arcs (Definition 10). An example of the third type is the insertion of an *is-a* arc between two classes in the view schema that are not *is-a* related in the global schema; and thus the view schema would imply an incorrect type inheritance. The manual specification approach requires the development of a view correctness checker that verifies the correctness of the manually inserted arcs. This view correctness checker would have to determine which type of error has occurred and then would have to take appropriate actions to correct it. Instead, we propose the use of an automatic view generator which is guaranteed to generate a valid view schema. Note that the proposed automatic view generator is similar in flavor (and in its power) to the view correctness checker.

In the remainder of this section, we present two algorithms that automate the view schema creation process. The first algorithm has linear complexity but it assumes that the global schema has a tree structure. The second algorithm overcomes the restricting assumption of a tree class hierarchy, but it does so at the cost of a larger complexity.

The problem we address in this section can be formally stated as follows. Let $GS = (V,E)$ be a global schema. Assume that a subset of classes VV $\subseteq$ V of GS has been marked by the view identifier $< VS >$, i.e., these marked classes have been selected to belong to the view schema VS via the operations given in Section 4.2. We wish to develop an algorithm that automatically determines a set VE of *is-a* edges among classes in VV, such that $VS = (VV, VE)$ is a valid view schema. For a definition of valid view schema see Definition 11.

5.2 View Schema Generation For Global Schemata without Multiple Inheritance

First we describe the algorithm A1 for the automatic generation of a view schema, which assumes that the global schema class structure is a tree. The algorithm traverses the GS schema tree in a breadth-first manner from the root down to the leaves. For each node C_i in GS (starting with the root node) that is marked by $<VS>$ it searches all branches in the subtree rooted at C_i. An *is-a* edge is inserted into VS between the parent C_i of a subtree and all subclasses of C_i that (1) are also marked by $<VS>$ and (2) are the closest to C_i in the tree. More formally, if $(C_1, C_2 \in VS)$ and $(C_1$ *is-a** $C_2)$ in GS and $(\nexists\ C_i$ *in* $VS)((C_1$ *is-a** $C_i)$ and $(C_i$ *is-a** $C_2))$, then the algorithm A1 inserts the edge $(C_1$ *is-a* $C_2)$ into VS. By Definition 11, this newly inserted edge is a *required* edge. This edge is guaranteed not to become *redundant*, since in a tree-structured graph there is at most one path between any two nodes.

For a given parent node C_i, this search process stops when the algorithm either finds a marked child or a leaf node of GS on all branches of C_i's subtree. The algorithm terminates when all subtrees rooted in nodes C_i marked by $<VS>$ have been searched. Hence, each marked node serves as direct parent for subclasses in its class hierarchy once. A detailed description of the algorithm A1 is presented in Figure 8.

Next we present a step-by-step example of applying the A1 algorithm to generate a view schema.

Example 8. *Figure 9 presents an example of applying the A1 algorithm to generate a view schema. The figures with numbers ending in "1" show how the search process proceeds through the global*

Assumption:

Global Schema *GS* is tree-structured (No multiple inheritance).

Input:

Global Schema *GS = (V,E)* and View Schema *VS=(VV, VE)*
with VV $\subseteq$ V marked by the view identifier <VS> and *VE*=$\emptyset$.

Output:

Determine a set of *is-a* edges *VE* on *VV*,
such that *VS = (VV, VE)* is a *valid* view schema on *GS*.

Data Structures:

ParentQueue is a queue to hold nodes of *GS*.
ChildrenQueue is a queue to hold nodes of *GS*.
Parent and Child are variables that hold one class each.

Algorithm A1: Creation of *Is-A* Arcs for A View Schema.

```
algorithm Edge-Creation(GS, VS) is
    Push the root of GS onto ParentQueue[2].
    while (Parent = pop(ParentQueue)) do
        Put all children of Parent in GS onto the ChildrenQueue.
        while (Child = pop(ChildrenQueue)) do
            if Child is in VS then
                insert isa(Child,Parent) into edges(VS);
                put Child onto ParentQueue;
            else
                Put all children of Child in GS onto the ChildrenQueue;
            endif
        endwhile
    endwhile
end algorithm;
```

Figure 8: The Virtual Schema Creation Algorithm A1.

[2]We assume here that each view schema includes the root object class of the global schema. This assures that each view schema is a DAG rather than a forest.

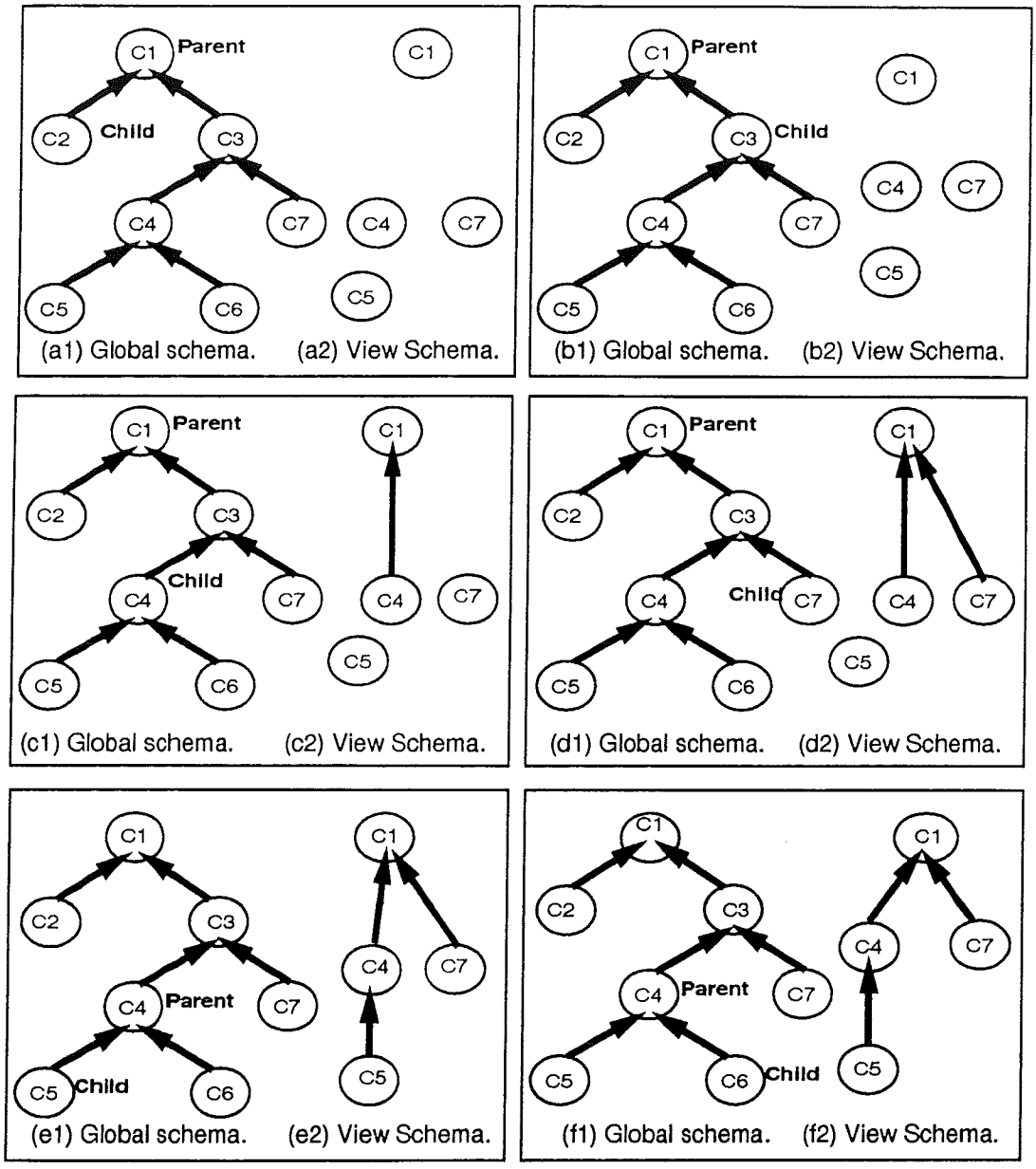


Figure 9: Example Snapshots For Tracing of the A1 Algorithm.

schema GS, and those ending in "2" describe the incremental insertion of edges into the view schema VS.

Figures 9.a1 and 9.a2 show GS and the selected nodes of VS, respectively. First the root class C_1 of GS is put onto the ParentQueue. The Parent C_1 is popped from the queue. Then all children of Parent are inserted into the ChildrenQueue, and the Child C_2 is popped from the queue. This results in the state listed below (and shown in Figure 9.a1).

ParentQueue = $< >$
Parent = C_1
ChildrenQueue = $< C_3 >$
Child = C_2

The if-statement then finds that the Child C_2 is in not VS and inserts all children of C_2 into the ChildrenQueue. VS stays unchanged as shown in Figure 9.a2. A new iteration of the inner while loop results in the following search state (corresponding to Figure 9.b1).

ParentQueue = $< >$
Parent = C_1
ChildrenQueue = $< >$
Child = C_3

Again the Child C_3 is not in VS, therefore we insert the children $\{ C_4 , C_7 \}$ of C_3 into the ChildrenQueue. VS still stays unchanged as shown in Figure 9.b2. The state generated by the next iteration of the inner while loop is shown below (and in Figure 9.c1):

ParentQueue = $< >$
Parent = C_1
ChildrenQueue = $< C_7 >$
Child = C_4

This time the Child C_4 is in VS. Hence we insert an edge from the Child to the Parent into VS, namely, the edge (C_4 is-a C_1). The resulting VS is depicted in Figure 9.c2. We also add the Child C_4 to the ParentQueue. The next iteration results in the state given below (and in Figure 9.d1).

ParentQueue = $< C_4 >$
Parent = C_1
ChildrenQueue = $< >$
Child = C_7

The Child C_7 is again in VS, hence we insert the edge (C_7 is-a C_1) into VS (Figure 9.d2). We also add the Child C_7 to the ParentQueue. Since the ChildrenQueue is empty, we start with the outer while loop by popping a new Parent C_4 off the ParentQueue. The children $\{ C_5 , C_6 \}$ of C_4 are added to the ChildrenQueue, and the new state is given below (and in Figure 9.e1).

ParentQueue = < C_7 >
Parent = C_4
ChildrenQueue = < C_6 >
Child = C_5

Since the current Child C_5 is in VS, we insert an edge between C_5 and the new Parent C_4 into VS (as shown in Figure 9.e2). We also add the Child C_5 to the ParentQueue. The new state after the pop of the ChildrenQueue is given below (and in Figure 9.f1).

ParentQueue = < C_7, C_5 >
Parent = C_4
ChildrenQueue = < >
Child = C_6

C_6 is not in VS. However, C_6 is a leaf of GS and therefore no new children are added to the ChildrenQueue. The ChildrenQueue is again empty. We start with the outer while loop and pop a new Parent C_7 from the ParentQueue. No more edges are created since both C_7 and C_5 are leaf nodes. The view schema shown in Figure 9.f2 is the final result.

Theorem 1. (Correctness) *The algorithm A1 generates a valid view schema VS=(VV, VE) assuming the underlying global schema GS=(V,E) does not have multiple inheritance.*

Proof (By Contradiction): For the resulting view schema *VS* to be valid, we need to show that the algorithm A1 creates *all required* and *no redundant* and *no incompatible* *is-a* edges for *VS*.

Part I: The algorithm A1 creates *no redundant* *is-a* edges.

Assume that A1 inserted an edge (C_2 *is-a* C_1) into *VS* that is *redundant* in *VS* (See Figure 10.a). By Definition 9, this means that: (1) the classes C_1 and C_2 are in *VS*, and (2) there exists another class C_3 in *VS* with $C_3 \neq C_1 \neq C_2$, such that (C_1 *is-a** C_3) in *GS* and (C_3 *is-a** C_2) in *GS*. For A1 to insert the edge (C_2 *is-a* C_1) into *VS*, the state must have been Parent=C_1 and Child=C_2. Since the schema is a tree, there is exactly one path from C_1 to C_2. Therefore, A1 must have traversed this path from the Parent C_1 down to Child C_2 without finding any marked node C_k on the path. If A1 would have found a marked node C_k, then it would have stopped the search for Parent=C_1 along this branch. This is a contradiction to the assumption that the inserted edge (C_2 *is-a* C_1) is redundant, i.e., that there is a marked node C_3 between C_2 and C_1. Hence, the assumption is incorrect.

q.e.d.

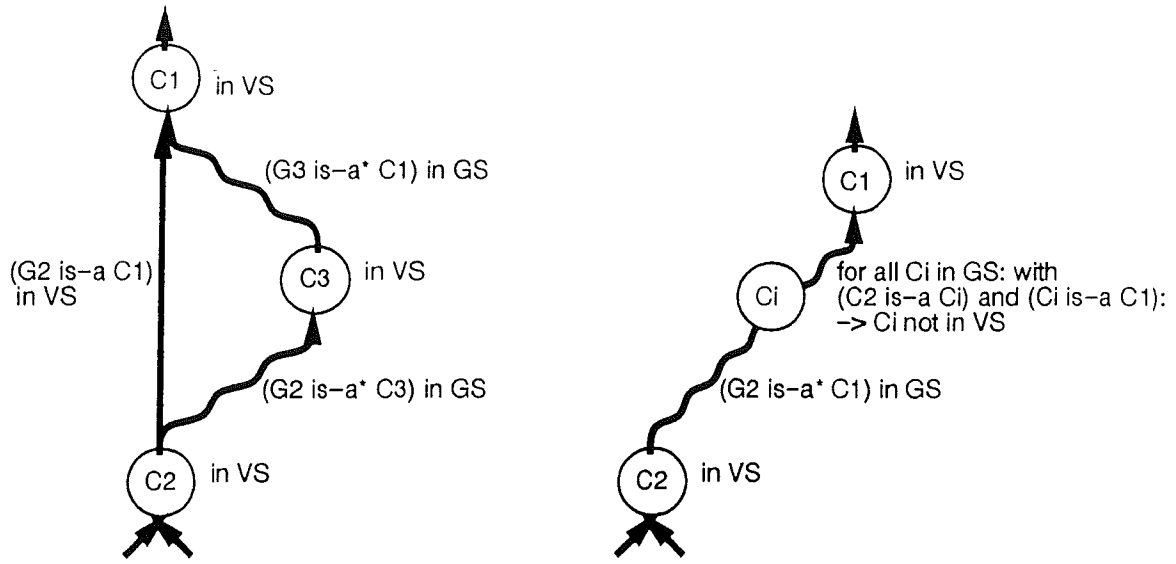


Figure 10: Redundant and Required Edges.

Part II: The algorithm A1 creates *all required is-a* edges.

Assume that A1 did not insert an edge (C_2 *is-a* C_1) into *VS* even though the edge (C_2 *is-a* C_1) was *required* in *VS* (See Figure 10.b). By Definition 8, for the edge (C_2 *is-a* C_1) to be *required* in *VS* means that: (1) the classes C_1 and C_2 are in *VS*, and (2) C_1 and C_2 are *is-a* related in *GS* by (C_1 *is-a** C_2), and (3) $\forall$ C_i *in GS with* (C_2 *is-a** C_i) and (C_i *is-a** C_2), the class $C_i \notin$ *VS*. $C_1 \in$ *VS* implies that at some point during the algorithm, A1 will make this marked node a parent, i.e., Parent=C_1. Thereafter, A1 would search all branches of the subtree rooted at C_1 for marked children. For A1 not to discover the node C_2, it must find some other marked node C_i before finding C_2 (i.e., above C_2 and below C_1). But by Definition 11, there is no such node C_i between C_2 and C_1 that is marked. Hence, the assumption leads to a contradiction.

q.e.d.

Part III: The algorithm A1 creates *no incompatible is-a* edges.

Assume that A1 inserted an edge (C_2 *is-a* C_1) into *VS* that is *incompatible* in *VS*. By Definition 10, this means that C_2 and C_1 are not *is-a* related in GS, i.e., there is no path in GS between C_2 and C_1. For A1 to insert the edge (C_2 *is-a* C_1) into *VS*, the state must have been Parent=C_1 and Child=C_2. Since the algorithm A1 is traversing *GS* rather than *VS*, this situation can only occur if the node C_2 was found by traversing *GS* downwards from C_1 on some path in *GS*. Hence, the node C_2 is a *subclass* (*child*) of C_1 in *GS*, denoted by (C_2 *is-a* * C_1). This is a contradiction to the assumption that this relationship does not hold in *GS*.

q.e.d.

Theorem 2. (Complexity) *The complexity of the algorithm A1 is linear in the number of nodes in GS, i.e., $O(|GS|)$, assuming the class hierarchy of the global schema is a tree.*

Proof: *GS* being a tree structure implies that there is exactly one unique path to reach each node from the root. Hence, a node can never be reached through a second path, and therefore each node (except the root node) is placed into the ChildrenQueue exactly once. It is inspected in constant time by checking its annotation to determine whether it belongs to the view schema, and then it is removed from the ChildrenQueue. Now, it is either completely discarded, or, it is placed in the ParentQueue in order to be compared to its children. Each of these comparisons with the same node as parent can be charged to the respective children rather than to the parent node. Hence, every node is inspected exactly once, and the complexity is $O(|GS|)$.

q.e.d.

5.3 View Schema Generation For Global Schemata with Multiple Inheritance

The algorithm A1 presented in the previous section does not generate a valid view schema if the underlying global schema is a DAG rather than a tree, as shown in the lemma below.

Lemma 1. *For a global schema GS with multiple inheritance, the algorithm A1 generates a view schema with all required but possibly also redundant is-a arcs.*

Proof: Part II of the proof for Theorem 1, which shows that *all required* edges are added to *VS*, also applies here. However, part I of Theorem 1, which shows that no *redundant* edges will be added to *VS*, is no longer applicable. In a schema with multiple inheritance, some classes have more than one path connecting them. In this case, the algorithm does not prevent the insertion of redundant edges as shown in the example in Figure 11. There are two different paths to reach C_4 in *GS* in Figure 11. The algorithm A1 first searches along the first path and creates the (C_4 *is-a* C_1) arc (Figure 11.c). It then searches along the second path and discovers the (C_4 *is-a* C_3) arc (Figure 11.d).

q.e.d.

In order to create a valid view schema for a global schema with multiple inheritance, we need the ability to remove redundant arcs. Below we present the algorithm A2 which solves this problem

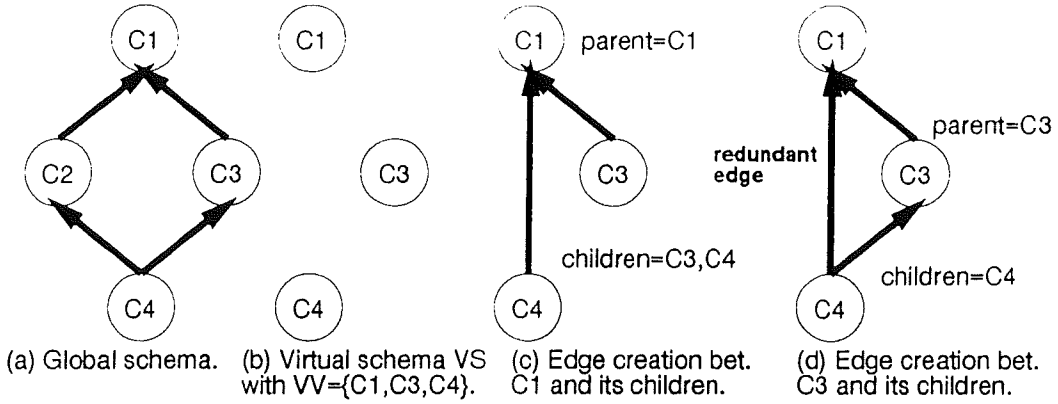


Figure 11: An Example of Creating Redundant View *Is-A* Arcs.

based on the transitive closure property of the *is-a* relationship. Recall that an *is-a* arc (path of length 1) between two nodes C_1 and C_2 is *redundant* if and only if there is another path between these two nodes that is of length 2 or larger. Assume that for every pair of nodes, we have the length of the longest path between the two classes. This length is set to negative infinity, if they are not *is-a** related. We then insert an *is-a* arc between two classes in *VS* if the length of the longest found path between the two classes is equal to one. Such an edge is a *required* edge, since there is no secondary path to connect the two classes. If this length is larger than one then this edge is a *redundant* edge. Therefore, we do not insert it into *VS*. The exact algorithm for the removal of redundant arcs is given in Figure 12.

Algorithm A2 uses a Longest-Path procedure that is a variation of the well-known all-pair cheapest path algorithm, which computes for each ordered pair of nodes (v,w) of a graph G the highest cost of all paths between them. This algorithm is based on the assumption that we have a directed graph G=(V,E) in which each edge is labeled by an element from a closed semiring (S,+,*,0,1) (e.g., [1], pg. 195–201]). We thus label the edges of the schema graph by the length of the longest paths between two nodes. The system (I∪"$-\infty$",*Max*,*Plus*,"$-\infty$",0) then is a closed semiring with I the integer labels of the edges of the graph, *Max* and *Plus* the addition and multiplication operators on the semiring, respectively, and "$-\infty$" and "0" the identity elements. The Longest-Path procedure then computes $A^k_{(i,j)}$ with $1 \leq i,j \leq n$ and $0 \leq k \leq n$ with $A^k_{(i,j)}$ the maximum of the longest of the paths from C_i to C_j, such that all classes on the paths, except the two endpoints, are in the set $\{C_1, C_2, ..., C_k\}$. First, the algorithm computes the length of the longest paths between all pairs of nodes that go through no other nodes (i.e., with indices $k \leq 0$). Then, it computes the length of longest paths that go through nodes with indices $k \leq 1$. And so on. Informally, this is based on the idea that the longest paths between C_i and C_j which pass through no node with an index higher than C_k is the longer one of the following two: (1) the longest path which passes

Input:

VS=(VV, VE) a view schema with the classes VC_i (i=1,..., n)
with VE containing all required and possibly some redundant edges of *VS*.

Output:

VS=(VV, VE) a valid view schema.

Data Structures:

A and tmp are integer matrices of size n = $|VS|$.
A(i,j) $\in I \cup -\infty$ will indicate the length of the longest path bet. VC_i and VC_j.

Algorithm A2: Removal of Redundant Edges.

```
procedure Initialize-LP (VS) return A is
    for i,j from 1 to n do
        if (i=j) then A[i,j] := 0;
        elseif (VC_i is-a VC_j) in VE then A[i,j] := 1;
        else A[i,j] := -∞
        endif
    endfor
end procedure

procedure Longest-Path (A) return A is
    tmp := A;
    for k from 1 to n do
        for i,j from 1 to n do
            tmp[i,j] := Max( tmp[i,j], A[i,k] + A[k,j] );
        endfor
        A := tmp;
    endfor
end procedure

procedure Edge-Removal(A,VS) is
    for i,j from 1 to n do
        if (A[i,j]>1) then
            remove the edge (VC_i is-a VC_j) from VE;
        endif
    endfor
end procedure;

algorithm Edge-Reduction( VS) is
    A := Initialize-LP ( VS).
    A := Longest-Path (A);
    Edge-Removal(A,VS);
end algorithm
```

Figure 12: Algorithm A2 for Removal of Redundant Arcs.

through no node higher than C_{k-1}, or (2) the path which is the longest path from C_i to C_k and then the longest from C_k to C_j, passing through no node higher than C_{k-1} between these points. This recurrence of the maximal path calculation holds for graphs with no cycles, and indeed our schema graphs are non-cyclic.

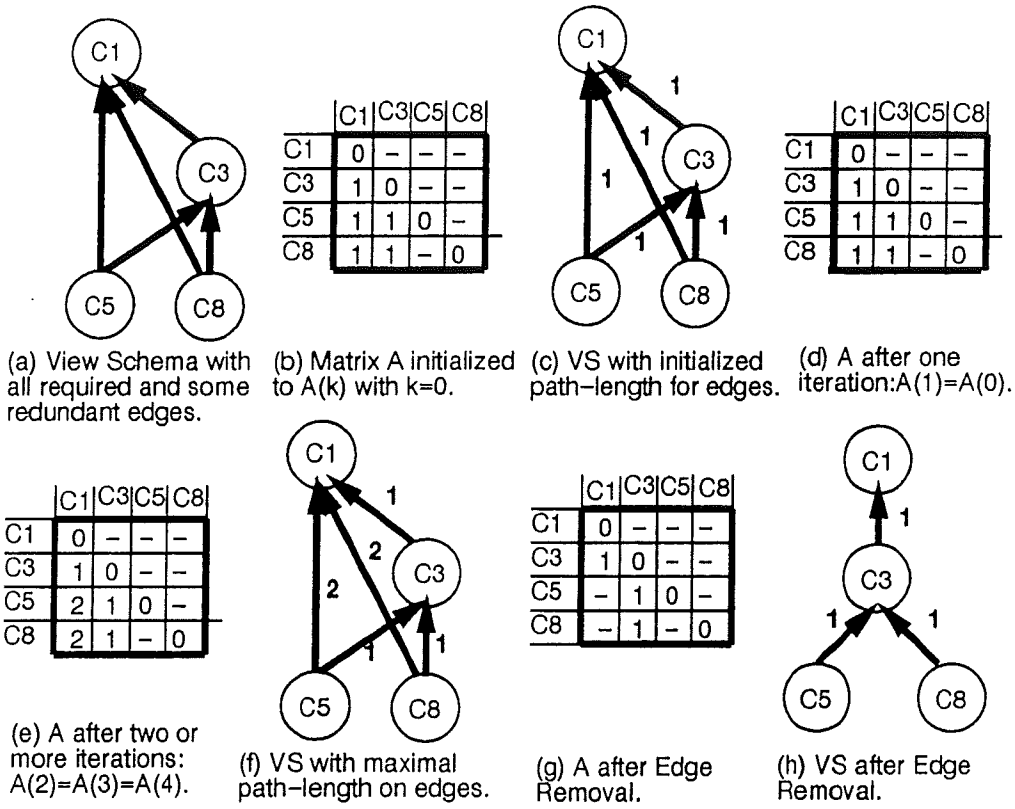


Figure 13: An Example of Removing Redundant Edges Using Algorithm A2.

Example 9. *In Figure 13, we present an example of applying the A2 algorithm to remove redundant edges from a view schema. Figure 13.a depicts the view schema VS with all required and possibly some redundant edges. The Initialize() procedure of algorithm A2 expresses VS in matrix format as shown in Figure 13.b. For each pair of classes we place a "1" into the matrix A if they are connected by an is-a edge, a "0" if they are identical, and a "-" (representing a "-∞"), otherwise.*

This is equal to $A^k_{(i,j)}$ with k=0, meaning that considered paths go through no other nodes, i.e., are of length one (Figure 13.c). Then the Longest-Path() procedure of A2 is applied to the matrix A to find the longest path between all pairs of classes. The first iteration of the Longest-Path function, (see Figure 13.d) which searches for paths that go through no node with an index higher than k=1 (which in our example corresponds to the node C_1), finds no new paths, i.e., $A^1_{(i,j)}=A^1_{(i,j)}$. The second iteration (see Figure 13.e) then finds paths that go through no node with an index higher

than k=2 (which in our example corresponds to the nodes C_1 and C_3). Iterations three and four do not find any new paths (there are none left). The longest paths found are shown in Figure 13.f. Now the Edge-Removal() procedure removes all edges that correspond to matrix entries larger than one (from Figure 13.e to 13.g). An entry larger than one means that there is a path of length greater or equal to 2 between two nodes in VS, and hence a direct is-a arc between these two nodes would be redundant. Finally, the virtual schema VS shown in Figure 13.h remains, containing all required and no redundant edges.

Theorem 3. (Correctness) *Given a view schema V=(VV,VE) with all required and possibly redundant is-a edges, the algorithm A2 generates a valid view schema V'=(VV',VE') with VV'=VV.*

Proof: We need to show that the algorithm A2 removes *all* of the *redundant* and *none* of the *required* edges. First, A2 initializes the matrix A by storing the following value at each position $A(C_i, C_j)$:

$$A(C_i, C_j) = \begin{cases} 0 & if\ C_i = C_j \\ 1 & if\ (C_i\ is\text{-}a\ C_j)\ in\ VS \\ -\infty & otherwise \end{cases} \quad (1)$$

This thus represents the graph in a matrix format by indicating the initial distance between each pair of classes C_i and C_j in VS by the entry $A(C_i, C_j)$. This assumes that the paths go through no any other node with index greater than 0, i.e., we only consider direct arcs. If there is a direct edge (C_i *is-a* C_j) then the distance is initialized to one. If there is no edge (C_i *is-a* C_j) then the distance $A(C_i, C_j)$ is initialized to "-∞".

As discussed above, the Longest-Path procedure is a variation of a well-known algorithm of cost calculations of paths. The proof of correctness for this part of the algorithm can be derived from a standard algorithms book (e.g., [1]) and thus is not given here. For the following, we assume that the Longest-Path procedure indeed terminates with each entry of the matrix *A(i,j)* indicating the length (i.e., the number of *is-a* edges) of the longest path between the nodes C_i and C_j. We thus have:

$$A(C_i, C_j) = \begin{cases} 0 & if\ C_i = C_j \\ n\ (with\ 1 \leq n \leq \infty) & if\ (C_i\ is\text{-}a\ ^*\ C_j) \\ -\infty & if\ not(C_i\ is\text{-}a\ ^*\ C_j) \end{cases} \quad (2)$$

Now it remains to show that the Edge-Removal() procedure removes all redundant and none of the required edges.

Part I: The Edge-Removal() procedure removes all *redundant* edges.

Assume that the algorithm did not remove a *redundant* edge (C_1 *is-a* C_2). By Definition 11, an edge (C_1 *is-a* C_2) is *redundant* if there exists some other class C_3 in the schema such that (C_1 *is-a** C_3) and (C_3 *is-a** C_2). Hence, there is a path of length greater or equal to two from C_1 to C_2, namely, the path going through the node C_3. Therefore, the Longest-Path procedure would produce an entry $A(C_1, C_2) \geq 2$. The Edge-Removal() procedure removes all edges (C_i *is-a* C_j) from *VS* with $A(C_i, C_j) > 1$. Therefore, this is a contradiction to the assumption that the edge is redundant.

q.e.d.

Part II: The Edge-Removal() procedure removes none of the *required* edges.

Assume that the algorithm did remove a *required* edge (C_1 *is-a* C_2). By Definition 11, an edge (C_1 *is-a* C_2) is *required* if there exists no other class C_3 in the schema such that (C_1 *is-a** C_3) and (C_3 *is-a** C_2). If there is no class on the *is-a* path between C_1 and C_2, but C_1 and C_2 are *is-a* related, then the only connection between the two nodes is a direct *is-a* arc. The Longest-Path procedure would produce the entry $A(C_1, C_2) = 1$ for such a path of length one. The Edge-Removal() procedure however removes edges (C_i *is-a* C_j) from *VS* only if $A(C_i, C_j) > 1$. Therefore, this is a contradiction to the assumption that the removed edge is required.

q.e.d.

Theorem 4. (Complexity) *The worst-case complexity of algorithm A2 is* $O(|VS|^3)$ *with* $|VS|$ *the number of classes in the view schema VS.*

Proof: We break the proof of complexity into the following steps:

- The algorithm A2 first initializes the matrix A by storing a value at each position A(i,j). The complexity thus is the size of the matrix, i.e., $O(|VS|^2)$.
- The Longest-Path procedure involves the computation of three nested for-loop. These three nested for-loops are of size $|VS|$ each. The body of the three for loops is of constant time. Hence, this computation has the complexity of $O(|VS|^3)$.
- Lastly, the removal of redundant edges is done by inspecting each field in the matrix A once. The complexity thus is the size of the matrix, i.e., $O(|VS|^2)$.

We now can combine these complexities to get the total complexity of A2. Complexity(A2) = O($|VS|^2 + |VS|^3 + |VS|^2$) = O($|VS|^3$).

q.e.d.

Now we put the edge generation algorithm A1 together with the redundant edge removal algorithm A2 to create a complete algorithm for automatic view generation.

Theorem 5. *Given a global schema GS=(V,E) with possibly multiple inheritance and a set of view classes VV ⊆ V, the algorithm A3 shown in Figure 14 generates a valid view schema VS=(VV, VE).*

Input:

Global Schema *GS = (V,E)* and View Schema *VS=(VV, VE)*
with VV ⊆ V marked by the view identifier <VS> and *VE*=∅.

Output:

The View Schema *VS=(VV, VE)* with *VS* valid.

Algorithm A3: Complete Valid View Schema Generation 1

algorithm View-Schema-Creation1(*GS, VS*) is
  Edge-Creation(*GS, VS*);
  Edge-Reduction(*VS*);
end algorithm;

Figure 14: View Schema Creation Algorithm A3.

Proof: (Correctness) The proof of correctness of algorithm A3 is straightforward. First, it applies algorithm A1 to create *all required* and possibly some redundant edges of the view schema. Theorem 1 shows the correctness of this edge creation algorithm A1. Then, it applies the edge removal algorithm A2 to the resulting schema to remove *all redundant* edges. The correctness of this second step has been shown in Theorem 3. The result thus is a view schema with all required and no redundant edges, i.e., *VS* is a valid view schema.

q.e.d.

Algorithm A2 removes all redundant edges from a view schema. Consequently, we need no longer be concerned with preventing the creation of redundant edges during the first stage of the view schema creation algorithm. We can therefore replace the edge generation algorithm A1 by a transitive closure algorithm that does the following: Given a graph *G=(V,E)*, it creates a new graph *G*=(V,E*)* with $(\forall C_i, C_j \in V)$ $((C_i \textit{ is-a } C_j) \in E* \iff (C_i \textit{ is-a } * \ C_j) \in E)$. As can easily be seen, this generates all required but also all redundant *is-a* edges. This leads to a simpler implementation of the view schema creation algorithm, since the edge creation algorithm A1 is not implemented at all. Instead, the initialization procedure of mapping the schema graph structure into a matrix

representation is applied directly to *GS*, rather than *VS*, and all subsequent computations are performed on the matrix representation. Hence, rather than first manipulating the schema graph to insert edges by algorithm A1 and then manipulating the schema graph to remove redundant edges by algorithm A2, this algorithm works directly on the matrix representation. At the very end, once all required edges have been identified, it inserts these edges into the graph.

We list the transitive closure algorithm at the top of Figure 15 for the sake of completeness. Note that only a minimal change is required to convert the Longest-Path algorithm (top of Figure 12) to the Transitive Closure algorithm, which conceptually is equivalent to converting the system (IU"$-\infty$",*Max*,*Plus*,"$-\infty$",0) to the system ($\{0,1\}$,*And*,*Or*,1,0) with 1=true and 0=false. This code conversion is done by modifying the Longest-Path algorithm in two ways: (1) initialize the matrix representation for transitive closure by storing whether there is (is not) an *is-a* arc rather than storing the path length, and (2) the body of the nested for loop is replaced by a test for whether a path exists rather than accumulating the distance of the longest path. Due to this reuse of code, the implementation effort is minimal. The detailed algorithm A4, which uses the transitive closure algorithm in place of algorithm A1, is presented in Figure 15. Note that this algorithm has the additional advantage that the complexity analysis is straightforward.

Next, we explain the algorithm A4 for creating a valid view schema based on the example presented in Figure 16.

Example 10. *The steps of the algorithm A4 for valid view schema creation are:*

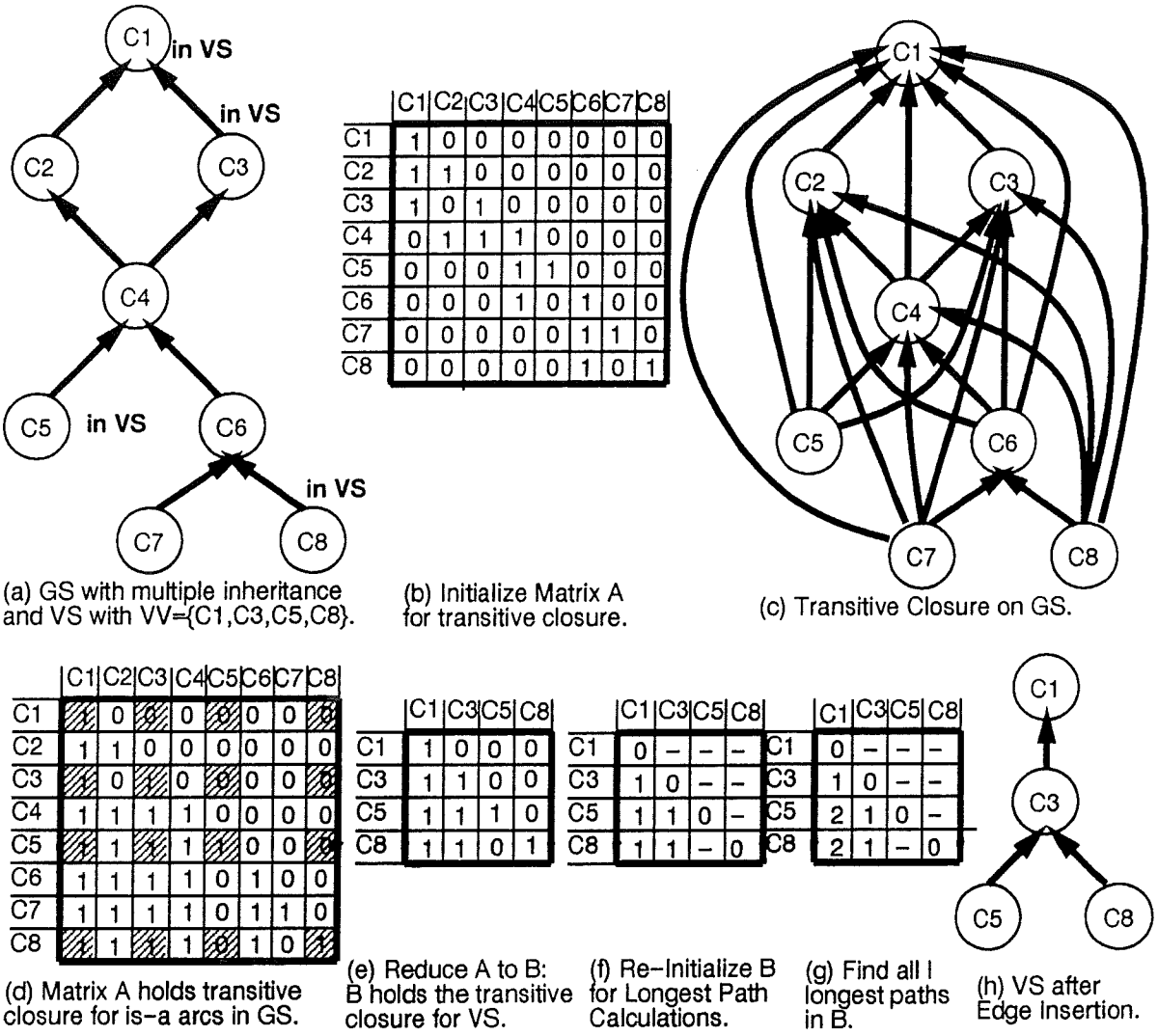
1. *A4 is given as input a global schema graph GS with a subset of nodes marked by the identifier <VS> (Figure 16.a).*
2. *Step 1 of A4 initializes the matrix A for the calculation of the transitive closure (Figure 16.b).*
3. *Step 2 of A4 then computes the transitive closure on the is-a relationships of the classes in GS. The transitive closure for GS in a graph representation and in a matrix representation are shown in Figure 16.c and 16.d, respectively.*
4. *Step 3 of A4 then reduces the matrix representation A for GS to the matrix representation B for VS by selecting all classes that belong to the virtual schema VS and all arcs of GS among these classes. The matrix A and the reduced matrix B are shown in Figure 16.d and 16.e, respectively.*

Input:
Let $GS=(V,E)$ be a global and $VS=(VV,VE)$ a view schema with VV$\subseteq$V and VE=$\emptyset$.
Output:
$VS=(VV,VE)$ a valid view schema.
Data Structures:
A and tmp are integer matrices of size $|GS|$; B is a matrix of size $|VS|$;
A(i,j) $\in \{0,1\}$ indicates whether (VC_i *is-a* VC_j) or not.

Algorithm A4: Complete Valid View Schema Generation 2

```
procedure Initialize-TC (GS) return A is
    for i,j from 1 to |GS| do
        if (i=j) then A[i,j] := 1;
        elseif (C_i is-a C_j) in V then A[i,j] := 1;
        else A[i,j] := 0;
        endif
    endfor
end procedure

procedure Transitive Closure (A) return A is
    tmp := A;
    for k from 1 to |GS| do
        for i,j from 1 to |GS| do
            tmp[i,j] := tmp[i,j] or (A[i,k] and A[k,j]);
        endfor
        A := tmp;
    endfor
end procedure

procedure Reduce-Matrix (A) return B is
    for i,j from 1 to |A| do
        if (C_i ∈ VS) and (C_j ∈ VS) then
            if (i=j) then B[C_i, C_j] := 0;
            elseif (A[C_i, C_j]=1) then B[C_i, C_j] := 1;
            elseif (A[C_i, C_j] = 0) then B[C_i, C_j] := -∞;
            endif
        endif
    endfor
end procedure

procedure Edge-Insertion(B,VS) is
    for i,j from 1 to |VS| do
        if (B[i,j]=1) then
            insert the edge (VC_i is-a VC_j) into VS;
        endif
    endfor
end procedure;

algorithm View-Schema-Creation2(GS, VS) is
    A := Initialize-TC (GS);
    A := Transitive-Closure (A);
    B: = Reduce-Matrix(A);
    B := Longest-Path (B);
    Edge-Insertion(B,VS);
end algorithm;
```

Figure 15: View Schema Creation Algorithm A4.



	C1	C2	C3	C4	C5	C6	C7	C8
C1	1	0	0	0	0	0	0	0
C2	1	1	0	0	0	0	0	0
C3	1	0	1	0	0	0	0	0
C4	0	1	1	1	0	0	0	0
C5	0	0	0	1	1	0	0	0
C6	0	0	0	1	0	1	0	0
C7	0	0	0	0	0	1	1	0
C8	0	0	0	0	0	1	0	1

	C1	C2	C3	C4	C5	C6	C7	C8
C1	1	0	0	0	0	0	0	0
C2	1	1	0	0	0	0	0	0
C3	1	0	1	0	0	0	0	0
C4	1	1	1	1	0	0	0	0
C5	1	1	1	1	1	0	0	0
C6	1	1	1	1	0	1	0	0
C7	1	1	1	1	0	1	1	0
C8	1	1	1	1	0	1	0	1

	C1	C3	C5	C8
C1	1	0	0	0
C3	1	1	0	0
C5	1	1	1	0
C8	1	1	0	1

	C1	C3	C5	C8
C1	0	–	–	–
C3	1	0	–	–
C5	1	1	0	–
C8	1	1	–	0

	C1	C3	C5	C8
C1	0	–	–	–
C3	1	0	–	–
C5	2	1	0	–
C8	2	1	–	0

Figure 16: Example of the Algorithm A4 for Creating A Valid Schema.

5. *Step 3 of A4 also re-initializes the matrix B to prepare it for the longest-path calculations by replacing the true/false values by path lengths. The matrix B with the transitive closure and the re-initialized matrix B are shown in Figure 16.e and 16.f, respectively.*

6. *Step 4 of A4 finds the length of all longest paths between each pair of nodes in VS. (transition from Figure 16.e to 16.f). A detailed example of how the longest-path calculation is carried out can be found in Figure 13. In particular, Figure 13.b is equal to the input to the procedure shown in Figure 16.e and Figure 13.f is equal to the result given in Figure 16.g.*

7. *Step 5 of A4 then inserts all arcs into VS that have a maximal path length of exactly one in B (Figure 16.h). The resulting graph is the final virtual schema VS.*

Having given the intuition behind the steps of the algorithm A4, we sketch a proof for its correctness.

Theorem 6. *The algorithm A4 presented in Figure 15 generates a valid view schema.*

Proof: Algorithm A4 implements the above described steps and thus creates a valid schema. The correctness of each step is shown below:

1. First, A4 initializes the matrix A to store the graph GS in a matrix representation.

2. Step 2 of A4 computes the transitive closure on the *is-a* relationships in matrix A to find all pairs of classes that are *is-a** related in GS. This procedure corresponds to a variation of the well-known transitive closure algorithm. The proof of correctness for this can be derived from a standard algorithms book (e.g., [1]) and thus is not given here. The result of this procedure is the transitive closure of the *is-a* relationships, which is the following:

$$A(C_i, C_j) = \begin{cases} 1\,(true) & if\ (C_1\ is\text{-}a\ ^*\ C_2)\ in\ GS \\ 0\,(false) & otherwise \end{cases} \tag{3}$$

Since a class is trivially *is-a* related to itself, this is equivalent to:

$$A(C_i, C_j) = \begin{cases} 1\,(true) & if\ (C_1 = C_2) \\ 1\,(true) & if\ not(C_1 = C_2)\ and\ ((C_1\ is\text{-}a\ ^*\ C_2)\ in\ GS) \\ 0\,(false) & if\ not(C_1\ is\text{-}a\ ^*\ C_2)\ in\ GS \end{cases} \tag{4}$$

Using the terminology defined in Section 2.3, this is equivalent to representing *all required* and *all redundant* but *no incompatible is-a* relationships of the graph.

3. Step 3 of A4 reduces matrix A to matrix B by extracting all entries $A(C_i, C_j)$ with $C_i, C_j \in VS$, i.e., $B(C_i, C_j)$:=$A(C_i, C_j)$ for $C_i, C_j \in VS$. Since A represents the transitive closure for GS, B will now represent the transitive closure for VS as shown below. The transitive closure of *is-a* edges means that for each class $C_i \in GS$, the matrix A will hold the edges to all other classes C_j with which C_i is *is-a* related, i.e., ($\forall$ j with $1 \leq j \leq |GS|$) ((C_i *is-a* * C_j) $\Longrightarrow$ A(C_i,C_j)=1). We can consequently conclude the following: (for each $C_i \in VS$) ($\forall$ j with $1 \leq j \leq |GS|$) ((C_j $\in VS$) and (C_i *is-a* * C_j)) $\Longrightarrow$ B(C_i,C_j)=1). The matrix representation of B thus corresponds to the following:

$$B(C_i, C_j) = \begin{cases} 1\ (true) & if\ (C_1 = C_2) \\ 1\ (true) & if\ not(C_1 = C_2)\ and\ ((C_1\ is\text{-}a\ *\ C_2)\ in\ VS) \\ 0\ (false) & if\ not(C_1\ is\text{-}a\ *\ C_2)\ in\ VS \end{cases} \quad (5)$$

4. For the longest path calculations, Step 3 of A4 replaces the true/false values of Equation 5 by path lengths. In particular, if B(C_i, C_j)=false, then it sets B(C_i, C_j):=$-\infty$. For C_i and C_j with i=j, it sets B(C_i, C_j):=0. In all other cases, B(C_i, C_j) remains "1". B thus is:

$$B(C_i, C_j) = \begin{cases} 0 & if\ C_1 = C_2 \\ 1 & if\ C_1\ is\text{-}a\ *\ C_2 \\ -\infty & if\ not(C_1\ is\text{-}a\ *\ C_2) \end{cases} \quad (6)$$

   This clearly corresponds to the initial path length for direct paths as is required for input to the longest-path calculation algorithm of A2.

5. Step 4 corresponds to algorithm A2. It runs the Longest-Path algorithm on this reduced matrix B to determine the length of the longest paths between any pair of classes in VS. This step has already been shown correct in Theorem 3, and the result is given in Equation 2.

6. Step 5 then inserts all required edges into VS as follows. For each pair of classes (C_i, C_j) with B[i,j]=1, it inserts the edge (C_i *is-a* C_j) into VS. The correctness of this step can be easily derived from Theorem 3.

q.e.d.

Theorem 7. (Complexity) *The worst-case complexity of the A4 algorithm is $O(|GS|^3)$ with $|GS|$ the number of classes in the global schema GS.*

Proof: We break the proof of complexity into the following steps:

- The algorithm A4 first initializes the matrix A by storing a value at each position A(i,j). The complexity thus is the size of the matrix, i.e., O($|GS|^2$).

- The Transitive-Closure procedure involves the computation of three nested for-loop of size $|GS|$ each. The body of the three for loops is of constant time. Hence, this computation has the complexity of $O(|GS|^3)$.
- The Reduce-Matrix function inspects each entry of matrix A exactly once, and thus has complexity of $O(|GS|^2)$.
- The complexity of the maximal path computation has been shown to be $O(|VS|^3)$ in Theorem 4.
- Lastly, the insertion of all required edges is done by inspecting each field in the matrix B once. The complexity thus is the size of the matrix, i.e., $O(|VS|^2)$.

We can now combine these complexities to get the total complexity of A4. Note that the two most time-consuming steps of the algorithm are the transitive closure on GS and the longest path computation on VS, while all other steps are of quadratic or lesser complexity. Since the number of classes in GS is larger or equal to the number of classes in VS, we can conclude that the complexity of the first computation outweighs the complexity of the second. Hence, the total complexity for algorithm A4 is $O(|GS|^3)$.

q.e.d.

Note that the view schema generation algorithm A4 has to be executed once for each view schema, namely, after the initial specification of the view schema. Furthermore, the first part of the A4 algorithm (the transitive closure on the global schema) will not have to be computed for each and every view schema. Instead it can be computed once a-priori (after creating the base schema). Thereafter it is updated only when new virtual classes are added to the global schema. Hence, the complexity for view schema generation in most instances only equals the complexity of the second part of A4, i.e., complexity(A4) = complexity(A2) = $O(|VS|^3)$, which may be considerably smaller than the complete complexity shown in Theorem 7.

6 RELATED WORK

Most initial efforts of defining views for OODBs suggest the use of the query language defined for their respective object model to derive a virtual class. For instance, the work by Heiler and Zdonik [5] and the work by Scholl et al. [14] fall in this category. *MultiView* can use any of these proposed class derivation mechanisms to implement the first phase of view schema generation, i.e., the customization of individual classes. It thus is a superset of these approaches.

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

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

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

Gilbert's proposal [4], similar to [15], is also based on the idea of defining multiple interfaces for a class object. *MultiView* does not require the extension of the traditional class concept, and thus can be implemented directly with the existing object-oriented database technology, while Gilbert's approach could not. Nonetheless, *MultiView* is as powerful as the multi-interface approach; any

view schema that can be defined using the multi-interface approach can also be defined using our strategy. In addition, our work allows for the direct application of the class derivation mechanisms proposed in the literature. The use of general query operators is currently not handled by [4].

7 CONCLUSIONS

In this paper, we have defined an object-oriented view to be a *virtual, possibly restructured, subschema graph* of the global schema rather than just one individual *virtual class*. We have presented a novel approach for supporting these *multiple view schemata* in OODBs, called *MultiView*. This approach is simple yet powerful; it allows for instance for the customization of a view schema by virtually restructuring both the generalization and the property decomposition hierarchies of the underlying global schema. We have also presented solutions to specific subtasks related to the proposed view paradigm. In particular, we have developed a language for view schema definition and two efficient algorithms for the automatic generation of the view schema hierarchy.

Note that our paradigm is not specific to a particular OODB model. This generality allows the *MultiView* approach to be incorporated into most existing OODBs. *MultiView* would then enrich these systems by allowing them to support a more powerful notion of views. Our paradigm builds on existing work in as much as it is independent of the class derivation operators chosen from the set of proposed operators in the literature [5, 7, 14, 11]. A major contribution of the proposed approach lies in its simplicity compared to alternative proposals [4], and hence the potential ease in adapting it for existing database systems and in implementing it with existing OODB technology.

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

Acknowledgements. We would like to thank Professor Daniel Gajski for steering us into this direction of investigating the view mechanism for object-oriented databases. Without his encouragement, this work would not have come about.

References

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

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

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

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

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

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

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

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

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

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

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

[12] Rundensteiner, E. A., "Supporting Multiple View Schemata in Object-Oriented Databases", Univ. of California, Irvine, Technical Report #92-07, Jan. 1992.

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

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

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

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

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