A Class Integration Algorithm and Its Application for Supporting Consistent Object Views

Elke A. Rundensteiner

Department of Information and Computer Science
University of California, Irvine

May, 1992

Technical Report 92-50

Contents

List of Figures

1 INTRODUCTION

While the concept of views has been studied extensively in the context of the relational model, it is largely unexplored for the newly emerging object-oriented database systems (OODB). Most initial proposals of views on OODBs suggest a direct extension of the view mechanism from relational databases to OODB systems, namely, they define an object-oriented view to be equal to an object-oriented query [8, 21, 11]. Note however that a relational schema is simply a set of 'unrelated' relations [6], whereas an object-oriented data schema corresponds to a complex structure of classes interrelated via generalization and decomposition relationships [11, 12]. We therefore define an object-oriented view, also called a ***view schema***, to be a ***virtual, possibly restructured, subschema graph*** of the global schema [17]. The construction of these view schemata raises challenging research issues in terms of how to restructure view schema graphs and how to relate them with the global schema structure, which did not arise in the context of the relational model.

We have developed a methodology, called *MultiView* [17, 19], for supporting multiple ***view schemata*** that successfully addresses these problems. Some of the key ideas underlying the design of *MultiView* can be summarized as follows:

1. An object-oriented view should look like a (regular) object schema, so that we can for instance use it to define other views.
2. The view specification mechanism provided to the user should be as simple as possible, so that non-database experts may be able to use the view system.
3. The user should be relieved of tedious tasks, whenever they can be automated.
4. The view system should help the user in enforcing the consistency of the view schema structure while defining the view.

MultiView breaks view specification into three independent subtasks: (1) customization and derivation of virtual classes, (2) integration of virtual classes into *one* consistent global schema graph and (3) the specification of arbitrarily complex, but consistent, view schemata on this comprehensive global schema. *MultiView*'s division of view specification into a number of well-defined subtasks, some of which have been successfully automated, makes it a powerful tool for supporting the specification of views by non-database experts while enforcing view consistency. In this paper, we present a solution to the second subtask, while solutions to the first and the third subtasks of *MultiView* are given in [17] and in [19], respectively.

MultiView's integration of virtual classes into one global schema graph takes care of the maintenance of explicit relationships between stored and derived classes in terms of type inheritance and subset relationships. This is useful for sharing property functions and object instances consistently among classes without unnecessary duplication. Furthermore, this organization may result in performance increases, since the query optimizer could exploit these known relationships among classes for query processing. For instance, the union of two classes C1 and C2 could be reduced to be equal to C1 without actual query processing if we knew that C2 is a subclass of C1. Last but not least, the integration of all virtual classes into one schema graph is a necessary basis for the third subtask of *MultiView*, namely, for the formation of arbitrarily complex view schema graphs composed of both base and virtual classes. If the virtual classes are not integrated with the classes in the global schema, then a view schema would correspond to a collection of possibly 'unrelated' classes rather than a generalization schema graph. In short, the integrated global schema graph represents the backbone of our view support system based on which view schemata are being designed.

Class integration tackles the problem of how a virtual class (as well as a complete view schema) derived during the first step of *MultiView* relates to, and can be integrated with, the remaining classes in the global schema. Note that in the relational model, where each relation is physically independent from all other relations, the integration of a virtual relation with the global schema corresponds to simply adding it to the list of existing relations (the data dictionary). In the context of OODBs, however, this is less straightforward. A class in an object schema is interrelated with other classes via an is-a hierarchy (for property inheritance and subsetting) and via a property decomposition hierarchy (for forming complex objects). Class integration in OODBs needs to guarantee the consistency of these class relationships when adding new classes (and thus new relationships) into the schema graph.

Not just individual virtual classes but complete (possibly conflicting) view schemata have to be integrated with another and with the underlying global schema into one consistent whole. This integration has to maintain the difference in the generalization and decomposition hierarchies of the view schemata. The proposed *MultiView* methodology solves this problem by separating the definition of view schemata into two independent steps, namely, one, the integration of virtual classes into *one* consistent global schema graph and, two, the definition of view schemata composed of both base and virtual classes on top of this augmented global schema. View schemata are consistently integrated with one another simply by being consistently integrated with the same underlying global schema.

We have identified two class integration problems that existing work does not appropriately handle, which are (1) the type inheritance mismatch for virtual classes and (2) the composition of *is-a* incompatible subset and subtype hierarchies into one consistent class hierarchy. The characterization of both problems is made possible by our approach of distinguishing between the type and the set content of a class as two independent concepts [18]. The first problem is concerned with constructing a type (and also class) hierarchy that includes the type of the new virtual class while assuring the correct type inheritance for all classes. To demonstrate this problem, we present examples for which a correct placement for a class C in a given schema graph G cannot be found. The second problem is caused by the fact that the class hierarchy combines the subset and the subtype relationships among classes into one *is-a* relationship. Differences between the subtype and the subset relationships of a class may cause problems in determining one consistent class hierarchy. For example, if a class's set content is lower in the corresponding set hierarchy and the class's type is higher in the corresponding type hierarchy, then there is a conflict in where to place the class in the combined class generalization hierarchy. In this paper we present a class integration algorithm that successfully solves both problems.

Inserting arbitrary subclass relationships between classes may result in an inconsistent schema graph in terms of property inheritance and subset relationships. Therefore, rather than requiring manual placement of classes in the schema graph and then checking the entered information for consistency, *MultiView* supports the automatic integration of classes. Automatic class integration does not only prevent the introduction of inconsistencies into the schema, but it also simplifies the task of the view definer. It decreases the time needed for view specification, and, more importantly, it makes it possible for a non-database expert to specify an application-specific view on his or her own.

For these reasons, we have developed an algorithm for the automatic classification of virtual classes into a global schema graph. This algorithm solves the two problems of type inheritance and of is-a incompatibility. The solution is based on type lattice theory [13], the essence of which is the creation of additional intermediate classes that restructure the schema graph. We present proofs of correctness and a complexity analysis for the classification algorithm. Furthermore, we characterize classification requirements of virtual classes derived by different object algebra operators. This characterization helps us to reduce the complexity of the classification task for most cases. For instance, we reduce classification from quadratic to linear complexity for classes derived by the Select, the Union, and the Difference operators and to constant complexity for those derived by the Refine operator.

The paper is organized as follows. In Section 2, we introduce object-oriented concepts related to views. In Section 3, we outline the *MultiView* methodology and describe the object algebra used for class derivation. Sections 4 and 5 outline the basics of class integration and characterize the two class integration problems, respectively. In Sections 6, 7 and 8, we present a solution to the first problem. In Section 9, we then present a solution to the second problem. The class integration algorithm is refined for virtual classes derived by some of the object algebra operators in Section 10. We compare *MultiView* to related work in Section 11 and conclude with Section 12.

2 OBJECT-ORIENTED CONCEPTS

2.1 The Object Model

Below, we introduce the basic concepts of object-oriented database (OODB) models needed for the remainder of the paper. Let P be an infinite set of property functions. property abstractions that includes attributes Each property function p ∈ P can be a value from a simple 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). For simplicity, we assume for the following that all properties in the schema have unique property names[1]. Let T be the set of all types. Let $t \in T$ be a type in T. Let $\mathbf{properties}_t$ be the set of attributes property functions (attributes) of type t and $\mathbf{domain}_p(t)$ denote the range (domain) of the property function p in type t. 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 by means of the interface of the respective ADT.

Let C be the set of all classes. A **class** $C_i \in$ C has a unique class name, a type description and a set membership. The type associated with a class corresponds to a common interface for all instances of the class, that is, the collection of applicable property functions. We refer to the name of the type associated with a class C by **type**(C) and to the set of property functions defined for C by **properties**(**type**(C)), or short, **properties**(C). If p ∈ P is a property function defined for C, i.e., p ∈ **properties**(C), then we refer to the domain of the property function p for C by $\mathbf{domain}_p$(C). A **class** is also a container for a set of objects. The collection of objects that belong to a **class** C is denoted by **extent**(C) := $\{o \mid o \in C\}$ with the member-of predicate "∈" defined based on the object identities of the object instances [16].

2.2 Type Hierarchy and Type Relationships

Definition 1. *We define a partial order on types T as follows. For two types t1 and t2 ∈ T, t2 is called a* **subtype** *of t1, denoted by t2 ⪯ t1, if and only if*

- $\mathbf{properties}_{t1} \subseteq \mathbf{properties}_{t2}$, *and*
- $(\forall\, p \in \mathbf{properties}_{t2})(\,\mathbf{domain}_p(t2) \subseteq \mathbf{domain}_p(t1)\,)$.

The first condition of Definition 1 states that a subtype must have the same property functions as its supertype and possibly additional ones. The second condition states that the domains of the property functions of a subtype must be contained within the domains of the respective property functions of the supertype, but that they could possibly be restricted.

Given a finite set of types T, we call t_1 a *direct subtype* of t_n and t_n a *direct supertype* of t_1, denoted by $t_1 \prec t_n$, if $(t_1 \preceq t_n)$ and $(t_1 \neq t_n)$ and there are no other types $t_{k_j} \in$ T (with

[1] Note that to determine whether two property functions are identical is equally hard to proving that two programs are equivalent. The uniqueness of property names assumption thus provides a simple yet elegant solution for this otherwise np-complete problem. We ensure uniqueness of properties by associating a unique property identifier with each newly defined property. Two properties that have the same property name can thus be distinguished internally based on their identifier. For other schemes of disambiguation of property names see [18].

j=1, ..., m) for which the following subtype relationships hold: $(t_1 \preceq t_{k_1})$ and $(t_{k_1} \preceq t_{k_2})$ and ... and $(t_{k_m} \preceq t_n)$. Based on this partial ordering function "$\preceq$", we can define a set of types T to correspond to a *type hierarchy* that explicitly represents all *direct subtype* relationships in terms of edges of a graph.

Definition 2. *A* **type hierarchy** *is a directed acyclic graph TH=(TV,TE), where TV is a finite set of types and TE is a finite set of directed edges. Each element in TV corresponds to a type t_i, while TE corresponds to a binary relation on TV × TV that represents all direct subtype relationships between pairs of types in TV. In particular, each directed edge e from t_1 to t_2, denoted by $e = <t_1, t_2>$, represents the direct subtype relationship $(t_1 \prec t_2)$ between t_1 and t_2 in TV.*

Next, operations on type descriptions are introduced which form a new type based on the sets of properties of two existing types [18].

Definition 3. *Let t1,t2 ∈ T. Then ⊔ is a function from $T^2 \rightarrow T$ that defines a new type t3 by*

$$t3 = t1 \sqcup t2.$$

The property functions $\textbf{properties}_{t3}$ *of t3 are defined by:*

$$\textbf{properties}_{t3} = \textbf{properties}_{t1} \cup \textbf{properties}_{t2}.$$

And, for each property function $p \in \textbf{properties}_{t3}$ of t3, the following domain is defined:

$$\textbf{domain}_{t3}(p) = \textbf{domain}_{t1}(p) \cap \textbf{domain}_{t2}(p).$$

Intuitively, the new type *t1* ⊔ *t2* denotes the collection of all properties defined for either *t1* or *t2*. In other words, the ⊔ function creates a new type from two existing ones (1) by building the union of the properties of both types and (2) by forming the intersection of the domains of properties, for all properties common to both source types. The following type relationships hold between the new type (*t1* ⊔ *t2*) and the two source types *t1* and *t2*:

$(t1 \sqcup t2) \preceq t1$, and

$(t1 \sqcup t2) \preceq t2.$

In fact, (*t1* ⊔ *t2*) is the *greatest common subtype* of *t1* and *t2*.

Definition 4. *Let t1,t2 ∈ T. Then ⊓ is a function from $T^2 \rightarrow T$ that defines a new type t3 by:*

$$t3 = t1 \sqcap t2.$$

The property functions $\textbf{properties}_{t3}$ *of t3 are defined by:*

$$\textbf{properties}_{t3} = \textbf{properties}_{t1} \cap \textbf{properties}_{t2}.$$

And, for each property function $p \in \textbf{properties}_{t3}$ of t3, the following domain is defined:

$$\textbf{domain}_{t3}(p) = \textbf{domain}_{t1}(p) \cup \textbf{domain}_{t2}(p).$$

Intuitively, the new type $t1 \sqcap t2$ denotes the collection of properties common to both types $t1$ and $t2$. In other words, the $\sqcap$ function creates a new type from two existing ones by building the intersection of the properties of both types and by forming the union of the domains of properties, for all properties common to both source types. The following type relationships hold between the new type ($t1 \sqcap t2$) and the source types $t1$ and $t2$:

$t1 \preceq (t1 \sqcap t2)$, and

$t2 \preceq (t1 \sqcap t2)$.

The type $t1 \sqcap t2$ is the *lowest common supertype* of $t1$ and $t2$. We distinguish between four cases for applying the $\sqcap$ function to two types $t1$ and $t2$, which result in different placements of the resulting type $t1 \sqcap t2$ in the type hierarchy:

1. $t1 \sqcap t2 = \emptyset$.
2. $(t1 \sqcap t2 = t1)$.
3. $(t1 \sqcap t2 = t2)$.
4. $(t1 \sqcap t2) \not\in \{t1, t2, \emptyset\}$.

In the first case, $t1$ and $t2$ have no common properties. By Definition 2, the lowest common supertype of $t1$ and $t2$ is then equal to the root of the type hierarchy. This implies that $t1$ and $t2$ will be placed into different subgraphs of the type hierarchy.

In the second case, $t2$ has the same properties that are defined for $t1$, and, $t2$ may also have additional properties. Also the domains for some of $t1$'s properties may be further restricted for $t2$. By Definition 1, $t2$ is a *subtype* of $t1$. $t2$ must therefore be placed below $t1$ in the type hierarchy. Case 3 is identical to case 2 with the roles of $t1$ and $t2$ reversed. If, by default, ($t1 \sqcap t2 = t1 = t2$) then the two types $t1$ and $t2$ are identical and their relative positioning in a type hierarchy would be the same location.

In the fourth case, $t1$ and $t2$ share some common property functions, but, in addition, each of them has their own property functions. By Definition 1, $t1$ and $t2$ are *type incompatible*. Since no type relationship can be established among them, they have to be placed into different subgraphs of the type hierarchy. As we will show below, this fourth case plays an important role in type classification.

2.3 Class Hierarchy and Class Relationships

Definition 5. *For two classes C1 and C2* $\in$ *C, C1 is called a* **subset** *of C2, denoted by* $C1 \subseteq C2$*, if and only if* $(\forall\, o \in O)\ ((o \in C1) \Longrightarrow (o \in C2))$.

Definition 6. *For two classes C1 and C2* $\in$ *C, C1 is called a* **subtype** *of C2, denoted by* $C1 \preceq C2$*, if and only if* (**properties**$(C1) \supseteq$ **properties**$(C2)$) *and* $(\forall\, p \in$ **properties**$(C2))$ ($\mathbf{domain}_p(C1) \subseteq \mathbf{domain}_p(C2)$).

Definition 6 is directly based on Definition 1, i.e., a class *C1* is defined to be a subtype of *C2* if and only if type(*C1*) is a subtype of type(*C2*).

Definition 7. *For two classes C1 and C2 $\in$ C, C1 is called a* **subclass** *of C2, denoted by C1 is-a C2, if and only if (C1 $\preceq$ C2) and (C1 $\subseteq$ C2).*

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

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 maintenance of the subset class relationships allows us to determine the containment of the object instances associated with one class within the extent of another class. This may for instance be useful for query processing. The maintenance of the subtype relationship is useful for the reuse of property function code; this feature is commonly known as property inheritance.

Let S = $\{ C_i \mid i = 1, ..., n\}$ be a set of classes. We call C_1 a *direct subclass* of C_n and C_n a *direct superclass* of C_1 if (C_1 *is-a* C_n) and ($C_1 \neq C_n$) and there are no other classes $C_{k_j} \in$ S (with j=1, ..., m) for which the following *is-a* relationships hold: (C_1 *is-a* C_{k_1}) and (C_{k_1} *is-a* C_{k_2}) and ... 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. The *direct subclass* relationship between C_1 and C_n is denoted by (C_1 *is-a*^{d} C_n); the *indirect subclass* relationship with (j$\geq$1). by (C_1 *is-a** C_n) for 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 8. *An* **object schema** *is a rooted directed acyclic graph*[2] *S=(V,E), where V is a finite set of vertices and E is a finite set of directed edges. Each element in V corresponds to a class C_i, while E corresponds to a binary relation on V $\times$ V that represents all direct is-a relationships between all pairs of classes in V. In particular, each directed edge e from C_1 to C_2, denoted by e = $<C_1, C_2>$, represents the direct is-a relationship between the two classes (C_1 is-a C_2). The designated root node, called Object, contains all object instances of the database and its type description is empty.*

We refer to the collection of *is-a* relationships of a set of classes as the **generalization hierarchy** of the object schema. Since the *is-a* relationship is *reflexive, antisymmetric* and *transitive*, the schema graph is a directed acyclic graph without any loops.

Definition 9. *Given a set of classes S = $\{ C_i \mid i = 1, ..., n\}$.*

a. *For all classes C_1,C_2 in S, an arc from source C_1 to sink C_2 is defined to be* **required** *in S, if (C_1 is-a C_2) in S and there is no C_x in S such that (C_1 is-a C_x) and (C_x is-a C_2), i.e., C_1 is a direct subclass of C_2 in S denoted by (C_1 is-a^d C_2).*

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

b. *For all classes C_1,C_2 in S, an arc from source C_1 to sink C_2 is defined to be* **redundant** *in S, if there is a class C_x in S such that (C_1 is-a C_x) and (C_x is-a C_2), i.e., C_1 is a indirect subclass of C_2 in S denoted by (C_1 is-a* C_2).*

c. *For all classes C_1,C_2 in VV, an arc from source C_1 to sink C_2 is defined to be* **inconsistent** *in S if the subclass relationship (C_1 is-a C_2) does not hold.*

d. *The object schema graph G=(V,E) is defined to be* **valid** *if and only if (V=S) and the set E of is-a arcs of G contains* **all required** *and* **no redundant** *and* **no inconsistent** *arcs in S.*

A **valid** schema graph G is complete since, by Definition 9.a and by the transitivity property of the *is-a* relationship, two classes C_1 and C_2 in VS are – directly or indirectly – connected via an arc in G if and only if they are also *is-a* related in S. A **valid** schema graph G is minimal since, by Definition 9.b, there is no direct *is-a* arc between two classes if if there is already an indirect *is-a* path between them. Lastly, a **valid** schema graph is consistent since, by Definition 9.c, an *is-a* arc from source C_1 to sink C_2 exists in G if and only if the two classes are *is-a* related in S.

Once these class relationships are compiled and stored 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.

Definitions 3 and 4 that define operators on types are now extended to operators on classes.

Definition 10. *Let C1 and C2 $\in$ C be two classes. Then $\sqcup$ is a function from $C^2 \rightarrow C$ that defines a new class C3 by*

C3 := C1 $\sqcup$ C2

with

type(C3) := type(C1) $\sqcup$ type(C2)

with the later $\sqcup$ function equal to the function on types given in Definition 3. Content(C3) is not specified.

C3 is a common subclass of both C1 and C2 if and only if *C3* $\preceq$ C1 $\sqcup$ C2 and content(*C3*) $\subseteq$ C1 $\sqcap$ C2. *C3* is called the *greatest common subclass* of C1 and C2 if and only if *C3* = C1 $\sqcup$ C2 and content(*C3*) = C1 $\sqcap$ C2.

Definition 11. *Let C1 and C2 $\in$ C be two classes. Then $\sqcap$ is a function from $C^2 \rightarrow C$ that defines a new class C3 by*

C3 := C1 $\sqcap$ C2

with

type(C3) := type(C1) $\sqcap$ type(C2)

with the later $\sqcap$ function as defined in Definition 4. Content(C3) is not specified.

C3 is a common superclass of both C1 and C2 if and only if $C3 \succeq$ C1 $\sqcap$ C2 and content(*C3*) $\supseteq$ C1 $\cup$ C2. We call *C3* the *lowest common superclass* of C1 and C2 if and only if *C3* = C1 $\sqcap$ C2 and content(*C3*) = C1 $\cup$ C2.

In both definitions, the overloading of the functions $\sqcup$ and $\sqcap$ for the class and for the type parameter is a matter of convenience, since we generally deal with classes rather than with types. We use the notation "$C_i \in G$" to denote that the class C_i is a member of the set V of the schema graph G=(V,E). For notational convenience, we overload the "$\in$" operator to also apply to types. For t_i an element of the set of types T underlying the schema graph G=(V,E), we say that t_i is an element of G, denoted by $t_i \in G$, if and only if $(\exists C_i \in G)(\text{type}(C_i){=}t_i)$.

2.4 Object-Oriented Views

We distinguish between **base** and **virtual** classes. **Base classes** are defined during the initial schema definition. Object instances that are members of base classes are explicitly stored as base objects. **Virtual classes** are defined during the lifetime of the database using 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 12. *The* **base schema** *(BS) is an object schema S=(V,E), where all nodes in V correspond to base classes with stored rather than derived object instances.*

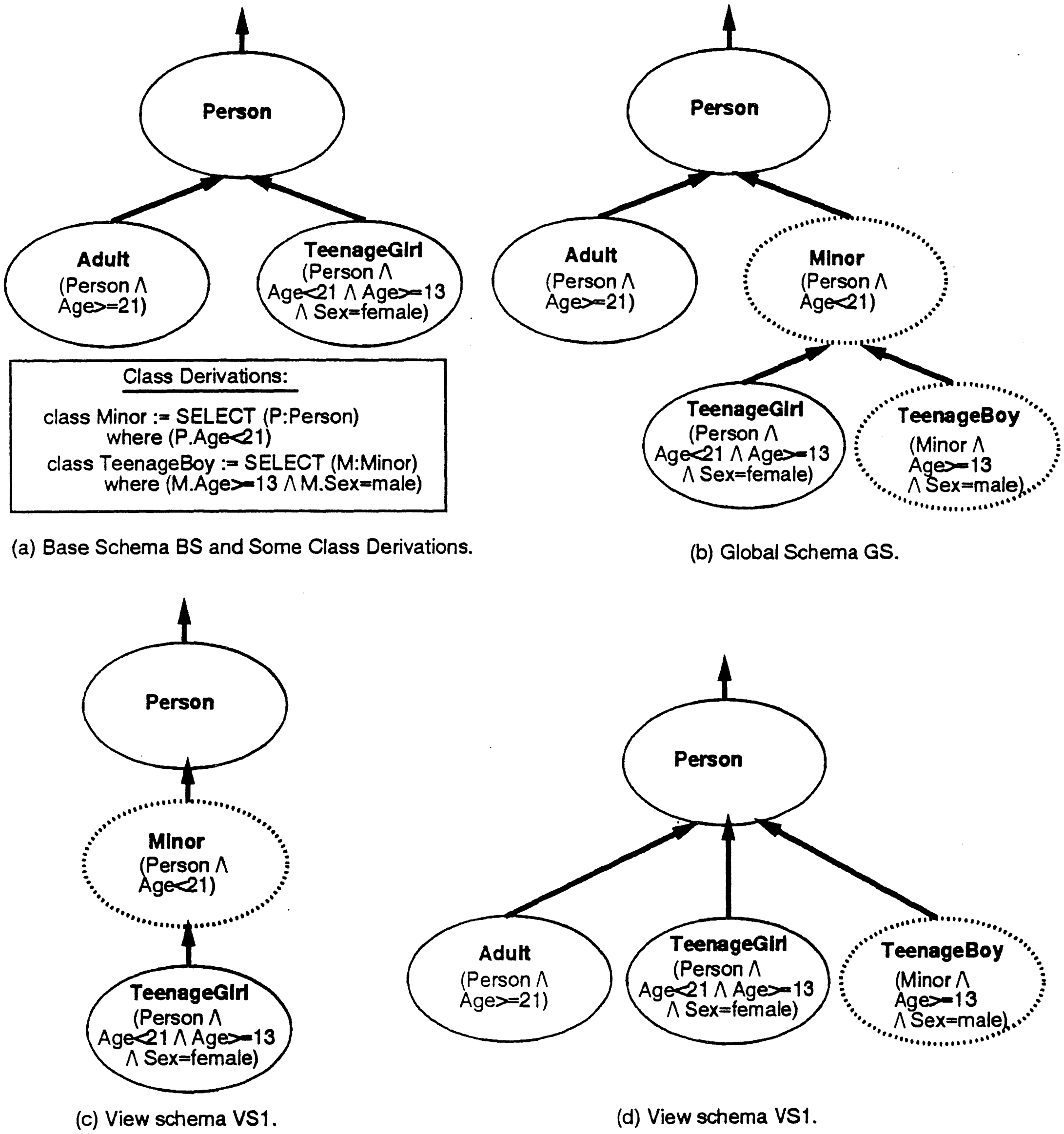
Definition 13. *Let BS be a* **base schema**. *The* **global schema** *(GS) is an extension of the base schema that is augmented by the collection of all virtual classes defined during the lifetime of the database as well as is-a relationships among this extended set of classes.*

A subgraph of the global schema which contains only virtual classes and their *is-a* relationships is commonly called a *virtual schema* [24, 1].

Definition 14. *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 *is-a** C_j) in GS. A view schema is a special case of an object schema. Therefore all properties of an object 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 and Some Class Derivations.

(b) Global Schema GS.

(c) View schema VS1.

(d) View schema VS1.

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

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

Note that the base schema is a special case of a view schema that consists exclusively of all base classes and no virtual classes. A base, a global, and a view schema are all special forms of an object schema, and, for that reason, each of them must obey the conditions of schema graph validity outlined in Definition 9. In addition, we have developed criteria for evaluating the consistency of the class generalization hierarchy of a view schema with the underlying global schema as well as the closure of the property decomposition hierarchy of a view schema [17]. These issues are not particular to the work presented in this paper, and therefore are omitted.

3 THE *MultiView* METHODOLOGY

3.1 Key Features of *MultiView*

In this section, we outline our approach for supporting multiple object-oriented views, called the *MultiView* methodology. Object-oriented views, also called ***view schemata***, correspond to ***virtual, possibly restructured, subschema graphs*** of the global schema with the later equal to a complex structure of classes interrelated via various relationships, such as, the orthogonal generalization and decomposition hierarchies [11, 12]. *MultiView* breaks view specification into the following three subtasks:

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

In this paper, we present a solution to the second subtask. Solutions to the first and the third subtasks of *MultiView* are given in [17] and in [19], respectively.

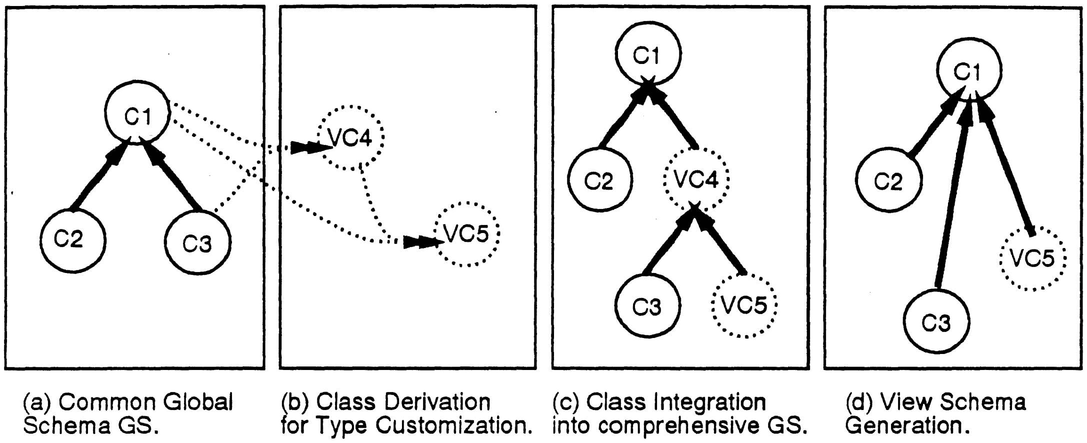
The separation of the view schema design process into a number of well-defined subtasks has several advantages. First, it simplifies the view specification and maintenance, since each of the subtasks can be solved independently from the others. Second, it increases the level of schema design support by allowing for the automation of some of the subtasks. In this paper, we present, for instance, algorithms that automate the second subtask of integrating virtual classes into *one* consistent global schema graph. Similarly, we have proposed algorithms for the third subtask, the automatic generation of the view schema hierarchy elsewhere [19]. In short, *MultiView*'s division of view specification into a number of well-defined subtasks, some of which have been successfully automated, makes it a powerful tool for supporting the specification of views by non-database experts while enforcing view consistency.

The first subtask of *MultiView* supports the customization of existing classes by deriving virtual classes with a modified type description and membership content. *MultiView* uses these class derivation mechanisms for a number of different purposes, e.g., to customize type descriptions, to limit the access to property functions, to collect object instances into groups meaningful for the task at hand, and so on. We assume that virtual classes are derived using object-oriented queries. Since there is no generally agreed-upon object algebra available in the literature [11, 8, 21], we define, for the purpose of this work, our own object algebra similar in flavor to the ones proposed in the literature (see Section 3.2). We focus, in particular, on the ***subset***, ***subtype*** and ***subclass*** relationships among the source and result classes derived by object algebra. This issue, important for successfully addressing the class integration problem, is generally ignored in the literature.

The second subtask of *MultiView*, which is the subject of this paper, supports the integration of virtual classes into one underlying global schema. This explicit maintenance of relationships between stored and derived classes is a necessary basis for the third subtask of *MultiView*, namely, for the formation of arbitrarily complex view schema graphs composed of both base and virtual classes. If the virtual classes are not integrated with the classes in the global schema, then a view schema would correspond to a collection of 'unrelated' classes rather than a generalization schema graph. In short, the integration of virtual classes into one global schema assures the consistency of all views with the global schema and with one another. Motivation for class integration is given in more detail in Section 4.

The third subtask of *MultiView* utilizes the augmented global schema graph for the selection of both base and virtual classes and for arranging these view classes in a consistent class hierarchy, called a *view schema*. This 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 [19]. After class selection, *MultiView's* view generation algorithm can automatically augment the set of selected view classes to generate a *valid* view schema hierarchy [19].

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



(a) Common Global Schema GS.
(b) Class Derivation for Type Customization.
(c) Class Integration into comprehensive GS.
(d) View Schema Generation.

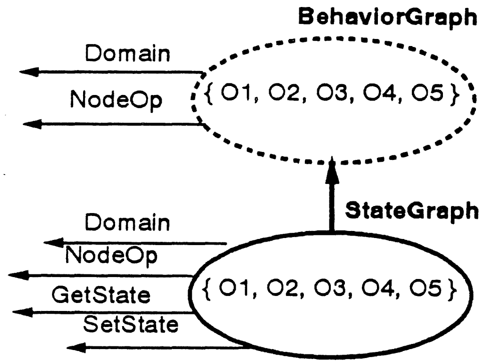
Figure 2: The *MultiView* Approach: From Base over Global to View Schemata.

Example 2. *This example of the view schema construction process is based on Figure 2. In this figure we depict base and virtual classes by circles and dotted circles, respectively. Given the global schema GS in Figure 2.a, the view definer first specifies the two virtual classes VC4 and VC5 using object-oriented queries (Figure 2.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 2.a to Figure 2.b. The second subtask then integrates the virtual classes VC4 and VC5 into GS as shown in Figure 2.c. View schema definition now proceeds by selecting a subset of classes from the augmented schema GS and interconnecting them into one schema graph. The resulting virtual schema graph, called a view schema, is given in Figure 2.d.*

3.2 Class Customization Using Object Algebra

The *MultiView* methodology is independent from the particular object algebra operators chosen for the class derivation subtask. However, since there is no agreed-upon standard for object algebra, we present a representative set of algebra operators below. As we will demonstrate in this section, the resulting class relationships between the derived class and the source classes vary with the type of the query operator. The determination of these *subclass* relationships, a necessary basis for the integration of virtual classes into the global schema, is generally not covered in the literature. The table in Figure 1 summarizes the object algebra operators, in particular, it gives their syntax, semantics and the resulting class relationships, while a more detailed description of the operators and some examples follow (See also [17]). The proposed object algebra consists of the following six operators, **hide**, **refine**, **select**, **union**, **intersect**, and **diff**; each of which is explained below.

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



Behavior-Graph = HIDE [SetState,
GetState] from State-Graph;

Figure 3: An Example of the **hide** Operator.

Example 3. *In Figure 3, the query "***BehaviorGraph** = **hide** *[SetState, GetState]* **from** *(***StateGraph***)" is used to derive the virtual class* **BehaviorGraph** *from the source class* **StateGraph**. *Then* **extent***(***BehaviorGraph***)* = **extent***(***StateGraph***), i.e.,* **StateGraph** $\subseteq$ **BehaviorGraph**. *Also* **type***(***StateGraph***) = [Domain, NodeOp, SetState, GetState] and* **type***(***BehaviorGraph***) = [Domain, NodeOp] imply* **StateGraph** $\preceq$ **BehaviorGraph**. *The is-a relationship (***StateGraph** *is-a* **BehaviorGraph***) is indicated by the edge from* **StateGraph** *to* **BehaviorGraph**.

The **refine** operator is a type-manipulating operator that adds additional property functions to a type rather than removing existing ones. It is similar in flavor to calculating a derived value for each tuple of a relation and then joining this derived value to the relation in the form of an additional column. It has the syntax "<*virtual*-class> = **refine** [<prop-function-defs>] **for** (<*source*-class>)" with <prop-function-def> being the definition of a new property function in the form of a new property name and a function body with the latter a legal arithmetic, boolean or set expression. The property functions in <prop-function-defs> are assumed to be distinct from all others in the global schema and therefore get assigned a unique property

hide	syntax	<*virtual*-class> := **hide** [<prop-functions>] **from** (<*source*-class>)
	semantics	**type**(<*virtual*-class>) := {p $\in$ P $\mid$ p $\in$ **properties**(<*source*-class>) $\wedge$ p $\notin$ <prop-functions>} **extent**(<*virtual*-class>) := **extent**(<*source*-class>)
	class rels	<*source*-class> $\preceq$ <*virtual*-class> <*source*-class> $\subseteq$ <*virtual*-class> <*source*-class> *is-a* <*virtual*-class>
refine	syntax	<*virtual*-class> := **refine** [<prop-function-defs>] **for** (<*source*-class>)
	semantics	**type**(<*virtual*-class>) := {p $\in$ P $\mid$ p $\in$ **properties**(<*source*-class>) $\vee$ p $\in$ <prop-function-def>} **extent**(<*virtual*-class>) := **extent**(<*source*-class>)
	class rels	<*virtual*-class> $\preceq$ <*source*-class> <*virtual*-class> $\subseteq$ <*source*-class> <*virtual*-class> *is-a* <*source*-class>
select	syntax	<*virtual*-class> := **select from** (<*source*-class>) **where** (<predicate>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class>) **extent**(<*virtual*-class>) := {o $\in$ O $\mid$ o $\in$ <*source*-class> $\wedge$ <predicate>(o) = true}
	class rels	<*virtual*-class> $\preceq$ <*source*-class> <*virtual*-class> $\subseteq$ <*source*-class> <*virtual*-class> *is-a* <*source*-class>
union	syntax	<*virtual*-class> := **union**(<*source*-class1>,<*source*-class2>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) $\sqcap$ **type**(<*source*-class2>) **extent**(<*virtual*-class>) := {o $\in$ O $\mid$ o $\in$ <*source*-class1> $\vee$ o $\in$ <*source*-class2>}
	class rels	<*source*-class1> $\preceq$ <*virtual*-class> $\wedge$ <*source*-class2> $\preceq$ <*virtual*-class> <*source*-class1> $\subseteq$ <*virtual*-class> $\wedge$ <*source*-class2> $\subseteq$ <*virtual*-class> <*source*-class1> *is-a* <*virtual*-class> $\wedge$ <*source*-class2> *is-a* <*virtual*-class>
intersect	syntax	<*virtual*-class> := **intersect**(<*source*-class1>,<*source*-class2>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) $\sqcup$ **type**(<*source*-class2>) **extent**(<*virtual*-class>) := {o $\in$ O $\mid$ o $\in$ <*source*-class1> $\wedge$ o $\in$ <*source*-class2>}
	class rels	<*virtual*-class> $\preceq$ <*source*-class1> $\wedge$ <*virtual*-class> $\preceq$ <*source*-class2> <*virtual*-class> $\subseteq$ <*source*-class1> $\wedge$ <*virtual*-class> $\subseteq$ <*source*-class2> <*virtual*-class> *is-a* <*source*-class1> $\wedge$ <*virtual*-class> *is-a* <*source*-class2>
diff	syntax	<*virtual*-class> := **diff**(<*source*-class1>,<*source*-class2>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) **extent**(<*virtual*-class>) := {o $\in$ O $\mid$ o $\in$ <*source*-class1> $\wedge$ o $\notin$ <*source*-class2>}
	class rels	<*virtual*-class> $\preceq$ <*source*-class1> <*virtual*-class> $\subseteq$ <*source*-class1> <*virtual*-class> *is-a* <*source*-class1>

Table 1: Derivation Operators: Syntax, Semantics and Class Relationships.

identifier. Its semantics are to refine the type description of the source class by adding to it the property functions listed in <prop-function-defs>. All other property functions of the source class are preserved. Again, the set content of the virtual class is equal to the set content of the source class.

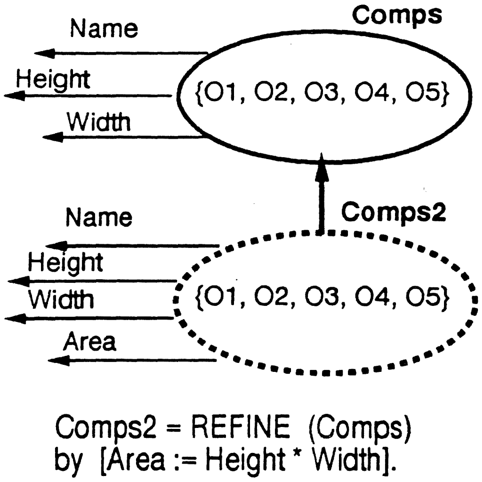


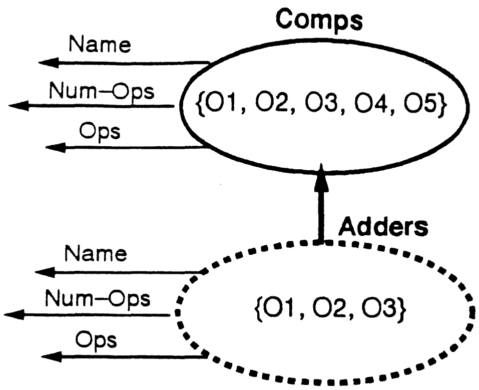
Figure 4: An Example of the **refine** Operator.

Example 4. *In Figure 4, the* **refine** *operator is used in the following query to derive* **Comps2** *from* **Comps***:* **Comps2** = **refine** *[Area = Height * Width]* **for** *(***Comps***). We have* **extent***(***Comps2***) =* **extent***(***Comps***), i.e.,* **Comps2** $\subseteq$ **Comps**. *The type of* **Comps2** *has been extended by the new method Aera. Hence* **type***(***Comps***) = [Name, Height, Width] and* **type***(***Comps2***) = [Name, Height, Width, Area], which together imply* **Comps2** $\preceq$ **Comps**. **Comps2** *is integrated into the global schema by placing* **Comps2** *below* **Comps** *as direct subclass.*

The **select** operator is a set-manipulating operator that selects a subset of object instances from a given set of objects – similar to the select operator of relational algebra [6]. It has the syntax "<*virtual*-class> = **select from** (<*source*-class>) **where** (<predicate>)" with <predicate> being some possibly complex function on the source class and its type description. Its semantics are to return a subset of object instances of the source class based on the evaluation of the associated predicate, namely, all object instances that satisfy the predicate are collected into the virtual class.

Example 5. *In Figure 5, the* **select** *operator is used in the query "***Adders** = **select from** *(***Comps***)* **where** *(Plus in* **Comps***.Ops)" to derive* **Adders** *from* **Comps**. *The* **Adders** *class consists of a selected subset of object members from the* **Comps** *class, namely, all components that implement the Plus operator. Thus* **Adders** $\subseteq$ **Comps**. *Also* **type***(***Adders***) =* **type***(***Comps***). The is-a relationship (***Adders** *is-a* **Comps***) has been added as indicated by the edge from* **Adders** *to* **Comps**.

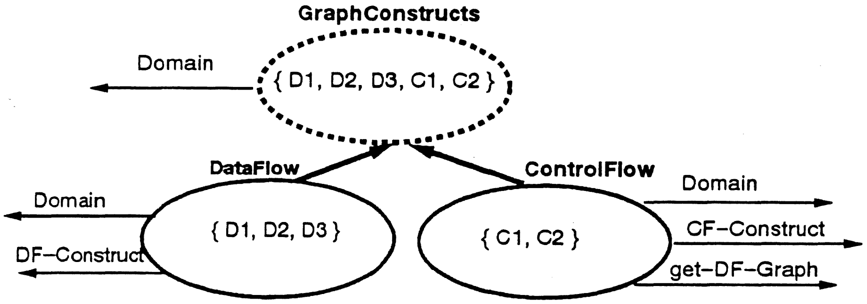
Set operators manipulate both the type description and the set membership of their two source classes. A detailed analysis of these set operators for OODBs can be found in [18]. The semantics of the **union** operator are to return a set of object instances composed of the



Adders =SELECT (Comps)
where (Plus in Comps.Ops);

Figure 5: An Example of the **select** Operator.

members of either or both of the source classes. The resulting type description is equal to the lowest common supertype of the two sources classes (Definition 4).



GraphConstructs = UNION (DataFlow,ControlFlow)

Figure 6: An Example of the **union** Operator.

Example 6. *In Figure 6, the* **union** *operator is used in the query "***GraphConstructs =** **union(DataFlow,ControlFlow)***" to derive the virtual class* **GraphConstructs** *from the source classes* **DataFlow** *and* **ControlFlow***. Then* **extent(GraphConstructs)** $=$ **extent(DataFlow)** $\cup$ **extent(ControlFlow)** $= \{D1, D2, D3\} \cup \{C1, C2\} = \{D1, D2, D3, C1, C2\}$. *Hence* **DataFlow** $\subseteq$ **GraphConstructs** *and* **ControlFlow** $\subseteq$ **GraphConstructs***. Also* type(**GraphConstructs**) $=$ type(**DataFlow**) $\sqcap$ type(**ControlFlow**) $=$ *[Domain, DF-Construct]* $\sqcap$ *[Domain, CF-Construct, get-DF-Graph]* $=$ *[Domain]. Hence* **DataFlow** $\preceq$ **GraphConstructs** *and* **ControlFlow** $\preceq$ **GraphConstructs***. The is-a relationships (***DataFlow** *is-a* **GraphConstructs***) and (***ControlFlow** *is-a* **GraphConstructs***) are indicated by the edges from* **DataFlow** *to* **GraphConstructs** *and from* **ControlFlow** *to* **GraphConstructs***, respectively.*

The **intersect** operator returns a set of object instances that are members of both source classes. The type description of the resulting virtual class is equal to the greatest common subtype of the two sources classes (Definition 3).

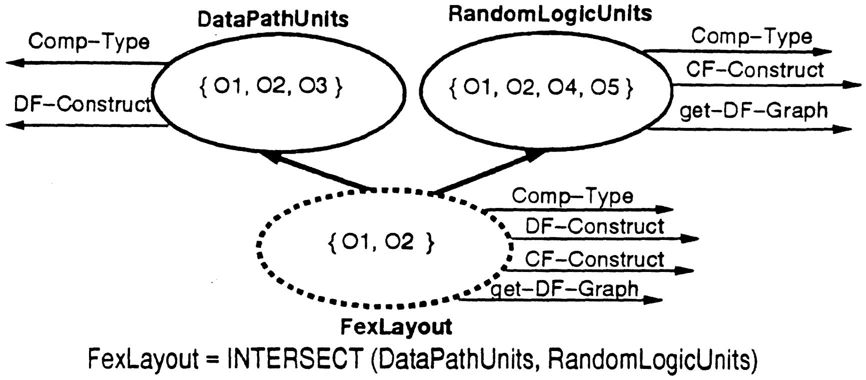


Figure 7: An Example of the **intersect** Operator.

Example 7. *In Figure 7, the* **intersect** *operator is used to derive the virtual class* **FexLayout** *from* **DataPathUnits** *and* **RandomLogicUnits** *using the query* **FexLayout** = **intersect**(**DataPathUnits**,**RandomLogicUnits**). *Then* **extent**(**FexLayout**) = **extent**(**DataPathUnits**) $\cap$ **extent**(**RandomLogicUnits**) = $\{O1, O2, O3\} \cap \{O1, O2, O4, O5\}$ = $\{O1, O2\}$. *Hence* **FexLayout** $\subseteq$ **DataPathUnits** *and* **FexLayout** $\subseteq$ **RandomLogicUnits**. *Also* **type**(**FexLayout**) = **type**(**DataPathUnits**) $\sqcup$ **type**(**RandomLogicUnits**) = *[Comp-Type, DF-Construct]* $\sqcup$ *[Comp-Type, CF-Construct, get-DF-Graph] = [Comp-Type, DF-Construct, CF-Construct, get-DF-Graph]. Hence* **FexLayout** $\preceq$ **DataPathUnits** *and* **FexLayout** $\preceq$ **RandomLogicUnits**. *The is-a relationships (***FexLayout** *is-a* **DataPathUnits***) and (***FexLayout** *is-a* **RandomLogicUnits***) are indicated by the edges from* **FexLayout** *to* **DataPathUnits** *and from* **FexLayout** *to* **RandomLogicUnits**, *respectively.*

Lastly, the **difference** operator returns a set of object instances that are members of the first but not of the second source class. The resulting type description is equal to the type description of the first source class. No subset, subtype or subclass relationships hold between the second source class and the virtual class.

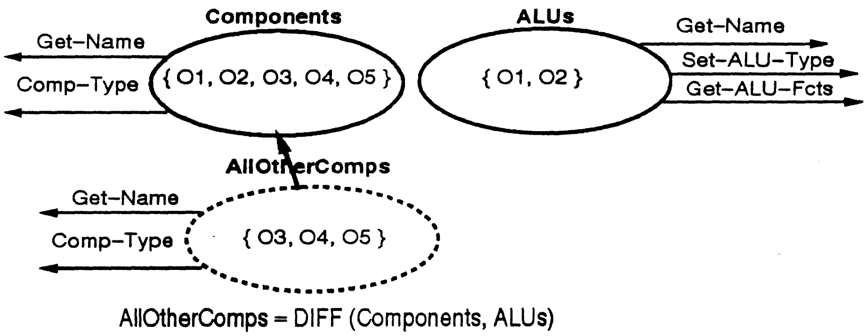


Figure 8: An Example of the **diff** Operator.

Example 8. *In Figure 8, the* **diff** *operator is used in the query "***AllOtherComps** = **diff**(**Components**,**ALUs**)*" to derive* **AllOtherComps** *from* **Components** *that are not in* **ALUs**. *We have* **extent**(**AllOtherComps**) = **extent**(**Components**) – **extent**(**ALUs**) = $\{O1, O2, O3, O4, O5\}$ - $\{O1, O2\}$ = $\{O3, O4, O5\}$. *Thus* **AllOtherComps** $\subseteq$ **Components**. *Also* **type**(**AllOtherComps**) = **type**(**Components**) = *[Get-Name, Comp-Type] and thus*

AllOtherComps $\preceq$ **Components**. *The relationship (***AllOtherComps** *is-a* **Components***) has been added to Figure 8.*

4 CLASS INTEGRATION: SUPPORTING A GLOBAL SCHEMA GRAPH

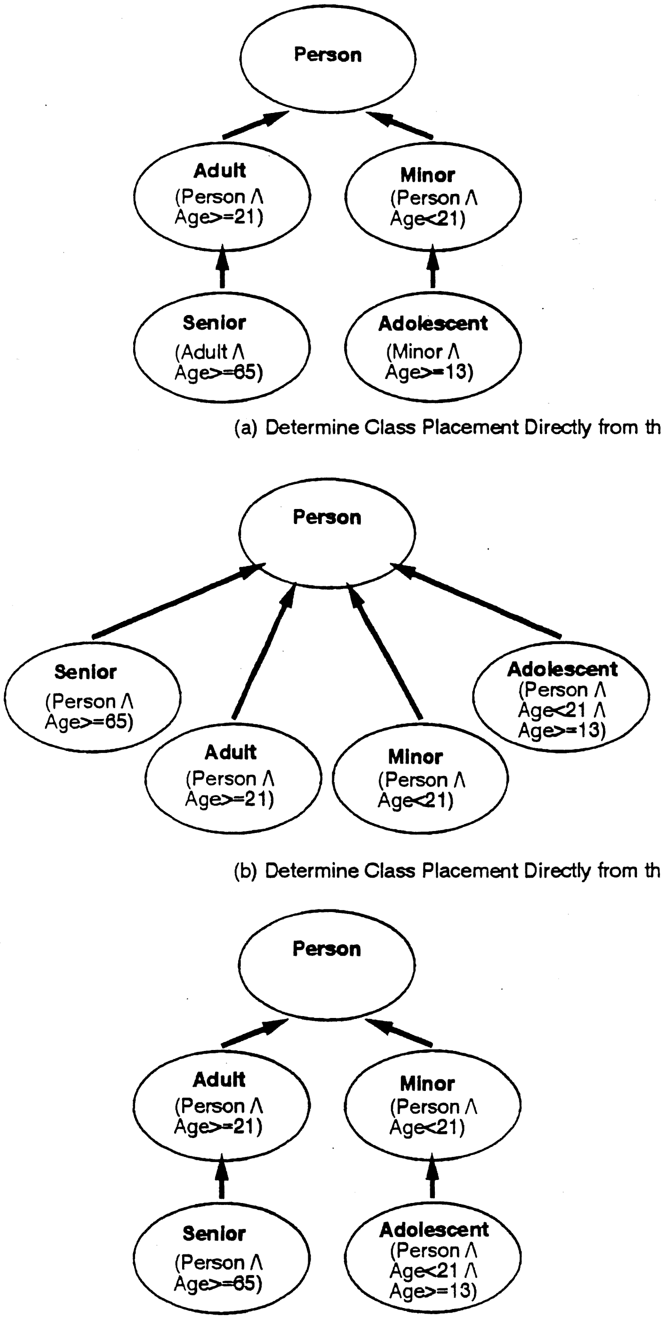
4.1 Motivation

MultiView supports the integration of virtual classes created for different view schemata into one comprehensive global schema. Class integration is concerned with finding the most appropriate location in the schema graph for a given virtual class with the term 'appropriate' meaning correct in terms of property inheritance and subset relationships between classes (Definition 9). The derivation specification of a new class explicitly states some subsumption relationship between the source and the result class (as indicated in Section 3.2). For example, a virtual class created by the **select** operator is a subclass of its source class. We hence could place the result class of a selection as *direct subclass* of its source class. This placement is not semantically incorrect, but it results in an incomplete schema graph that does not capture all existing class relationships (Definition 9). Namely, there may be additional subsumption relationships between the derived class and other classes in the schema that are not directly derivable from the class derivation. It is the task of class integration to find these class relationships and to explicitly represent them in the schema graph. These ideas are illustrated with the example below (for which the class derivations used in Figure 9.a have been borrowed from ([1], pg. 242).

Example 9. *Figure 9.a depicts the specifications for deriving the virtual classes* **Adult**, **Minor**, **Senior**, *and* **Adolescent**. *Figure 9.a also depicts the resulting virtual class hierarchy that incorporates these four classes. This hierarchy that has been generated using the simplistic strategy of inferring the class placement directly from the definitions of the virtual classes [1]. In this case, the simple placement strategy did result in a valid class hierarchy (Definition 9). In Figure 9.b, we present alternative derivations for the semantically equivalent set of classes. Note that the simple placement strategy now generates a different virtual class hierarchy (Figure 9.b). This virtual class hierarchy is still consistent in the sense that it does not represent any incorrect is-a relationships (Definition 9). On the other hand, it is not complete since some subclass relationships, like, for instance, the relationship (***Senior** *is-a* **Adult***), are not explicitly captured in the structure of the schema graph. Clearly, the schema graph in Figure 9.b is less informative than the one in Figure 9.a. A classifier could determine these subclass relationships by comparing the class derivation predicates. For instance, predicate (Person* $\wedge$ *Age*$<$*21) of the class* **Minor** *clearly subsumes the predicate (Person* $\wedge$ *Age*$<$*21* $\wedge$ *Age*$\geq$*13) of the class* **Adolescent**. *A complete classification of classes derived using the class derivations given in Figure 9.b would result in the schema graph shown in Figure 9.c. Note that the schema graph shown in Figure 9.c is identical to the one in Figure 9.a, with the exception of some of the class derivation predicates.*

With the above example we want to convey that our goal is complete classification (Figure 9.c) rather than partial classification driven exclusively by the view-defining query (Figure 9.b).

Explicit capture of class relationships between stored and derived classes in the form of a global schema graph brings numerous advantages. It is for instance useful for sharing property functions and object instances among classes without unnecessary duplication. Since virtual classes are defined in terms of existing classes, they often use property functions of these



Class Derivations:

(1) class Adult := SELECT (P:Person)
where (P.Age>=21)

(2) class Minor := SELECT (P:Person)
where (P.Age<21)

(3) class Senior := SELECT (A:Adult)
where (A.Age>=65)

(4) class Adolescent := SELECT (M:Minor)
where (M.Age>=13)

(a) Determine Class Placement Directly from the Class Derivation.

Class Derivations:

(1) class Adult := SELECT (P:Person)
where (P.Age>=21)

(2) class Minor := SELECT (P:Person)
where (P.Age<21)

(3) class Senior := SELECT (P:Person)
where (A.Age>=65)

(4) class Adolescent := SELECT (P:Person)
where (P.Age<21 Λ P.Age>=13)

(b) Determine Class Placement Directly from the Class Derivation.

Class Derivations:

(same derivation as b.)

(c) Use Subsumption Inferencing to Determine Final Class Placement.

Figure 9: Complete Classification Versus Partial Classification.

base classes in their type description. Type inheritance between base and virtual classes thus becomes an issue - and this is exactly what is supported by global schema integration.

In addition, class integration serves data modeling purposes. Recall that one of the functions of an object schema is to explicitly model class relationships rather than having to recompute these relationships, whenever needed. Class integration follows this philosophy by organizing the concepts and objects of the application domain in a systematic manner such that they are more easily comprehensible by the users of the system.

Disadvantages of ignoring class integration would not only be the less informative class hierarchy but possibly also performance penalties. A known subclass relationship between two classes can be exploited by a query optimizer during query processing. For instance, if we know that C2 is a subclass of C1 then the union of the two classes C1 and C2 would be equal to C1. In this example, computationally expensive query processing has been replaced by a simple check of the existence of a subclass relationship among two classes. Moreover, insertion of an object instance into a class C1 automatically implies that the instance is also inserted in all superclasses of C1. If this class C1 is not placed at its optimal location (i.e., the lowest possible place in the schema graph), then the membership predicate of other classes in the class hierarchy would have to be checked to determine whether the new instance is also a member of that class.

Last but not least, the comprehensive global schema graph is a necessary basis for the third subtask of *MultiView*, namely, for the formation of view schema graphs composed of both base and virtual classes. If the virtual classes are not integrated with the classes in the global schema, then a view schema would correspond to a collection of possibly 'unrelated' classes rather than a generalization schema graph as defined in Definition 14.

4.2 Towards a Simple Classification Algorithm of Virtual Classes

Classification is the process of taking a new [class] description and putting it where it belongs in the [class] hierarchy [20]. A class is in the "right place" if it is below all classes that subsume it and if it is above all classes that it subsumes. We thus need to define a boolean function *subsumes()* that given two classes C1 and C2 will determine whether the first subsumes the second:

$$subsumes(C1, C2) = \begin{cases} true & if\ C1\ is - a\ C2 \\ fail & otherwise \end{cases}$$

Class C1 is said to subsume class C2 if and only if the set denoted by C1 ***necessarily*** includes the set denotes by C2. While exact details of the ***subsumes()*** function are dependent on the object model, we characterize its general features [20]. The ***subsumes***(C1,C2) function returns true if and only if the following holds:

- type description (both defined and inherited properties)
  - For each property function p1 defined for C1, there must be an equivalent function p2 defined for C2, where a property function could be a storable value, an object pointer, or a complex method. Since we assume for simplicity that property functions have a unique property identifier, two property functions are determined to be equivalent by

comparing these identifiers rather than by comparing the actual function body, the later being a potentially undecidable problem.

- declarative constraints on property functions
  - The domain for each function p1 defined for C1 must subsume the domain of the equivalent function p2 defined for C2.
  - Constraints for a property function p1 defined for C1 must include the constraints for the equivalent function p2 defined for C2, where constraints could be cardinality restrictions, access modes, and the like.
- membership constraints
  - Membership constraints are predicates that restrict the set content of a class, i.e., this could be a subset-predicate for base classes or a derivation query for virtual classes. Since a comparison of arbitrary expressions (possibly involving functions) is in general undecidable, one would either have to limit the expressiveness of the derivation specification such as to be computable, or, we could require a canonical predicate expression that can be broken into decidable subexpressions. In the later case, we would base the classification on the comparison of this partial information.

Clearly, more work is needed in each of these three issues. Details of a *subsumes()* function for the KL-ONE knowledge representation schema are for instance given in [20]. It would of course also be interesting to develop efficient *subsumes()* functions for some of the newly emerging object-oriented data models. However, this investigation is beyond the scope of this paper.

In general, the classification problem is not decidable since it may involve the comparison of arbitrary functions and predicates. Therefore, our classification algorithm is sound but not complete. The *subsumes()* function being *sound* means that if the function returns true for a pair of classes then the two classes are necessarily is-a related. Put differently, any subsumption relationship discovered by the classifier is legitimate. Second, the *subsumes()* function is *total*, i.e., it always terminates and returns either true or fail. However, the *subsumes()* function is not *complete*, i.e., the function is not guaranteed to discover a relationship between two classes even if one exists. In other words, we cannot guarantee that all subsumption relationships are discovered. For instance, if two classes have property functions with equivalent semantics but different property identifiers, then the *subsumes()* function will fail even though these two classes may indeed by equivalent. In the worst case, if some *is-a* relationship is not discovered, then the virtual class is placed higher in the class hierarchy than would theoretically be possible. This would still be a correct but possibly not the most informative class arrangement. For example, Figure 9.b represents the result of such a partial classification whereas Figure 9.c shows the result of a complete classification.

Once we have developed the *subsumes()* function for a particular object model, then the basic algorithm for finding the correct position for a class VC in a schema G=(V,E) can be summarized as follows. First, the classifier determines the subsumption relationships between VC and all other classes in the global schema using the *subsumes()* function. VC then is placed below all its direct parent classes and above all its direct children classes (Definition 9). As we will show in later sections, this simplified classification algorithm does not correctly account for the type inheritance underlying the schema graph. In fact, we will describe two problems, called the type inheritance mismatch problem and the *is-a* incompatibility problem, that this simple class integration algorithm does not properly address. In the remainder of the paper, we then present an algorithm for automatic classification that solves both problems.

4.3 Manual Placement Versus Automatic Classification

There are two obvious but not necessarily mutually exclusive approaches towards global schema integration:

1. we can require the view definer to manually place a newly defined virtual class into the global schema graph, and
2. we could develop an algorithm for automatic classification.

Clearly, class integration is required by the view methodology rather than being of direct modeling interest to the view definer. The view definer would have to be knowledgeable about all classes in the global schema, even those that are not related to his or her view schema, in order to correctly perform class integration. Also, as the size of the schema graph grows, class integration becomes a more and more involved process. For these reasons, we do not want to force the view definer to have to take care of this task. Instead, we suggest automatic classification. This would decrease the view creation time, and more importantly, it may allow a non-database expert to specify an application-specific view on his or her own.

Automatic class integration does not only simplify the task of the view definer, but it also prevents the introduction of inconsistencies into the schema. A view definer could easily place a class into an incorrect location or insert incorrect generalization arcs. A class VC is for instance placed incorrectly, if it is below a class C that is its subtype and/or its subset. The former would mean that the schema graph would incorrectly capture the fact that VC inherits property functions from C. The later, which corresponds to a mistake in set membership inclusion, would incorrectly represent VC as being a subset of C. A consequence of these errors would be an inconsistent global schema graph in terms of property inheritance and subset relationships – and, since view schemata are defined on top of the global schema, also inconsistent view schemata. This inconsistency would not only confuse the user of the database system but it may also result in inefficiencies in terms of query processing and it may potentially lead to incorrect results.

Besides the assertion of blatantly incorrect *is-a* relationships, there is also the possibility of adding redundant or omitting required class relationships (Definition 9). If the view definer omits *is-a* arcs that are ***required*** to describe the complete semantics of the view schema, then this would leave some of the type inheritance that is actually taking place unexposed. If the view definer inserts ***redundant is-a*** arcs, then this may obscure the structure of the schema graph and it may result in inefficient query processing. The manual classification approach thus requires the development of a consistency checker that verifies the correctness of the user-inserted classes and arcs. This consistency checker would be equivalent in flavor (and in complexity) to the automatic classifier. Therefore we have opted to automate this process of classification. This decision does not necessarily prevent manual specification, if so desired. In fact, the automatic classifier could be used to guide the user during the view specification process, or, it could serve as basis of a consistency checker for manually-specified class placement.

To summarize, automatic classification provides a means of enforcing semantics and of checking for consistency of the class hierarchy. Therefore, automatic classification is a superior alternative to manual classification of the taxonomy.

5 THE CLASS INTEGRATION PROBLEMS

In Section 4 we have described a simple solution approach to class integration that has been advocated repeatedly in the literature [20, 24, 1]. In this section we will show that this simple classification approach does not appropriately handle classification in all cases. Based on our distinction between the type and the set content of a class as two independent concepts [18], we were able to characterize the two problems that any class integration algorithm has to address:

1. inheritance mismatch problem in the type hierarchy, and
2. the problem of composing *is-a* incompatible subset and subtype hierarchies into one class hierarchy.

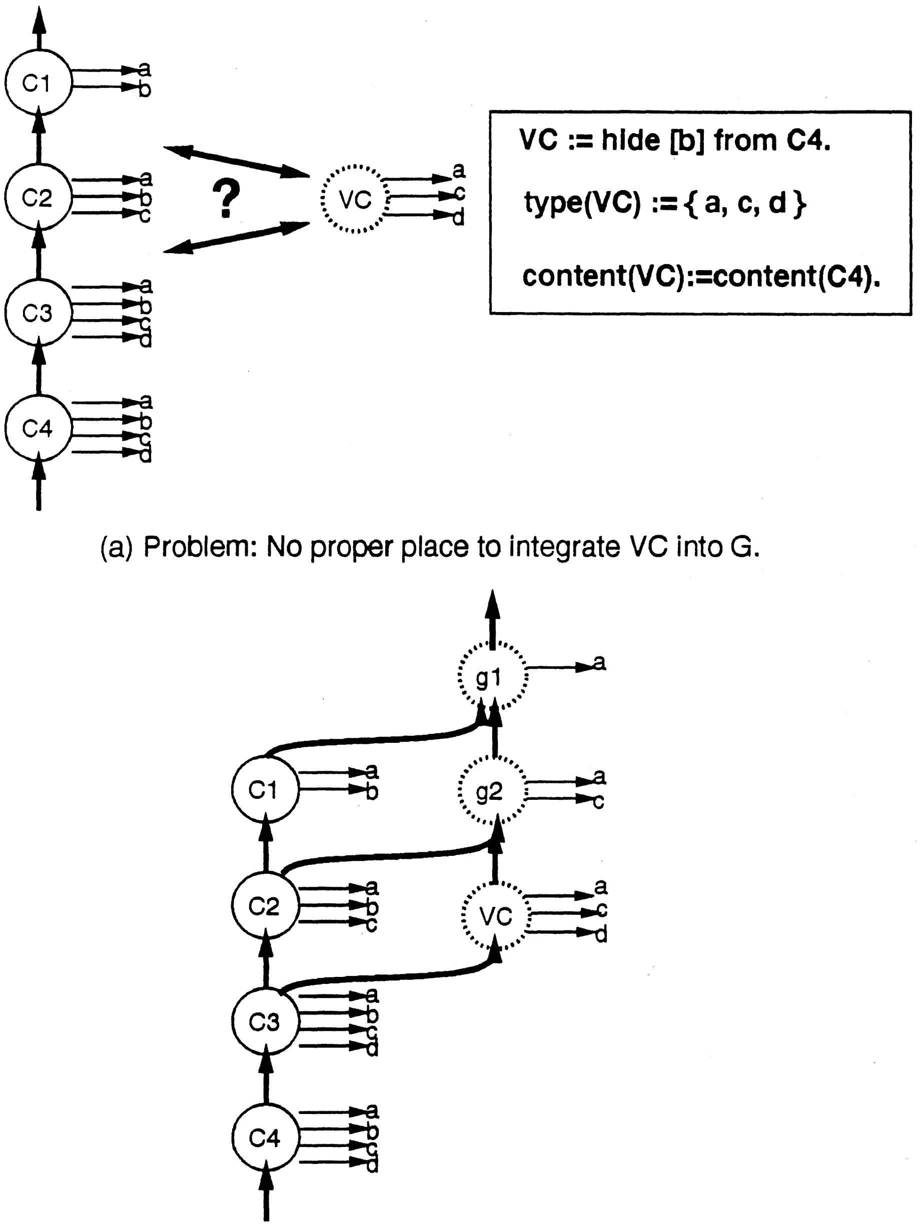
5.1 The Type Inheritance Problem

The first problem is concerned with constructing a type hierarchy that assures the proper inheritance of property functions after the insertion of new classes. As we will demonstrate below, in some cases there may be no correct placement for a class C in a given schema graph G. In such situations, we have to reorganize the schema graph such as to allow for the insertion of VC, while preserving the type inheritance for all existing classes. This problem is best explained with an example as is done in the following.

Example 10. *This example explains the type hierarchy mismatch problem based on Figure 10. Figure 10.a depicts the schema graph G and the virtual class VC derived by the query "VC := hide [b] from C4." Clearly, VC is a supertype (and superclass) of both C4 and C3, and therefore must be placed above C3 and C4 in the class hierarchy. We cannot determine any subtype relationships between C2 and VC, i.e., neither (C2 $\preceq$ VC) nor (VC $\preceq$ C2) hold. This is so because even though C2 and VC share some common properties, each of them also has properties that are not defined for the other. Therefore, VC can be placed neither below nor above C2 in G. The same is true for C1 and VC. As a consequence, there is no correct location for VC in G.*

In Figure 10.b, we present a solution to this problem. It is based on the idea of integrating intermediate classes into G that filter out the properties that are common to the existing classes in G and to the new class VC, so that they can be inherited by both. The intermediate class g1, for instance, filters out the property function "a" so that it can be inherited by both the base class C1 and by VC. As shown in Figure 10.b, this extended class hierarchy now allows for the consistent integration of VC into G.

In the example above, we assume that the set contents of all classes are identical, i.e., content(C1) = content(C2) = content(C3) = content(C4). This then clearly shows that the type inheritance mismatch is a problem of the type hierarchy alone - not dependent on the characteristics of the set hierarchy. In a later section, we will present a general solution to this problem based on work in type lattice classification.



(a) Problem: No proper place to integrate VC into G.

(b) Solution: Create intermediate classes for preserving type inheritance.

Figure 10: The Type Inheritance Problem.

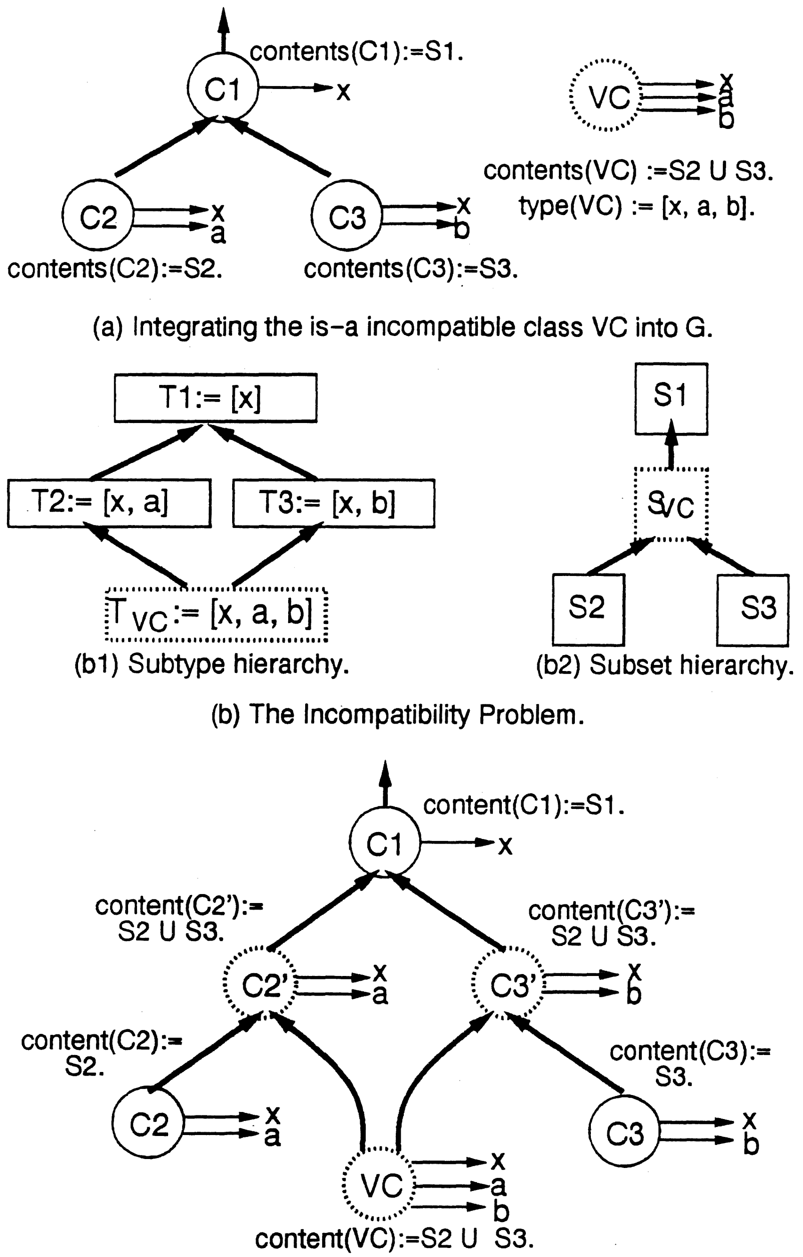
5.2 The *is-a* Incompatibility Problem

The second class integration problem results from the fact that we are combining the subset and the subtype relationships among classes into one relationship, called the *is-a* or subclass relationship. Differences between the subtype and the subset hierarchies underlying the class hierarchy may lead to conflicts when trying to compose these two hierarchies into one graph. More precisely, if a class's set content is lower in the corresponding set hierarchy and the class's type is higher in the corresponding type hierarchy, then there is a conflict in where to place the class in the combined class hierarchy. We can again solve this problem by reorganizing the schema graph such as to allow for the proper insertion of VC, while preserving the semantics of all existing classes. Below we explain the *is-a* incompatibility problem based on an example.

Example 11. *This example explains the problem caused by the incompatibility between subtype and subset relationships underlying a class hierarchy based on Figure 11. For this example, we assume that the type and the set content of a class C_i are denoted by the symbols T_i and S_i, respectively. Figure 11.a depicts the schema graph G and the virtual class VC with content(VC) := content(C2) $\cup$ content(C3) = S2 $\cup$ S3 and type(VC) := type(C2) $\sqcup$ type(C3) = T2 $\sqcup$ T3 = [x, a, b]. When integrating VC into G, we must first determine the subtype and the subset relationships among all classes. As shown in Figure 11.b2, type(VC) is a subtype of T2 and of T3 and therefore must be placed below both of them in the subtype hierarchy. On the other hand, as shown in Figure 11.b1, content(VC) is a superset of S2 and of S3 and therefore must be placed above both of them in the subset hierarchy. This clearly represents a conflict since in the final class hierarchy VC needs to be* above *C2 and C3 as dictated by the subset relationships and* below *C2 and C3 as dictated by the subtype relationships. As a consequence, there is no correct location for VC in G. We say that VC is is-a incompatible with respect to G.*

In Figure 11.c, we present a solution to the is-a incompatibility problem based on the creation of additional intermediate classes. For this example, we create the two intermediate classes C_2' and C_3'. Both C_2' and C_3' have the types of their respective source classes C_2 and C_3, such that VC can correctly inherit their combined type [x, a, b]. On the other hand, C_2' and C_3' both have a larger set content (namely, their set content is equal to the one of VC), so that VC can indeed be placed below them in the class hierarchy without violating the set hierarchy constraints.

Note that in Example 11, the problem for class integration is not caused by the type hierarchy itself, but by composing a subclass hierarchy out of the subtype and the subset hierarchy (see Figures 11.b1 and 11.b2). If we assume that the set contents of all classes in Figure 11 were identical, then a correct position can be found for VC without intermediate class creation. In this case, the resulting class hierarchy would be equal to the type hierarchy graph shown in Figure 11.b1. In the remainder of this work, we present a class integration algorithm that solves the two problems characterized in this section.



(a) Integrating the is-a incompatible class VC into G.

(b1) Subtype hierarchy.

(b2) Subset hierarchy.

(b) The Incompatibility Problem.

(c) Solving the Is-a Incompatibility Problem by Creating Intermediate Classes.

Figure 11: The *Is-a* Incompatibility Problem.

6 SOLVING THE TYPE INHERITANCE PROBLEM

In this section, we present an approach to class integration that solves the type inheritance problem introduced in Section 5. This problem refers to the fact that in general there is not always a placement for a class C in a given schema graph G that results in correct type inheritance (see also Example 10). We thus are interested in a class hierarchy structure where the correct placement of a new class can always be enforced. Our solution to this problem is based on type lattice theory [4, 14]. More precisely, we present an algorithm that solves the problem by inserting additional intermediate classes that reorganize the schema graph such as to allow for the insertion of VC, while preserving the type inheritance for all existing classes. In the following, we assume that there are no conflicts between the subset and the subtype hierarchies underlying the class generalization graph. This problem of *is-a* incompatibility introduced in Section 5 will be dealt with in Section 9.

6.1 The Type Closure and Class Closure Properties

In our object model (as well as in most others), a property is defined exactly once and, if used elsewhere, it is inherited from this original definition class. A direct consequence of this is the fact that if two types *t1* and *t2* share some common properties, then they must have ultimately inherited them from the same type. As defined in Definition 4, this lowest common supertype of *t1* and *t2* must have all attributes common to *t1* and *t2* and no others. We then define the type closure property of a type hierarchy as stated below.

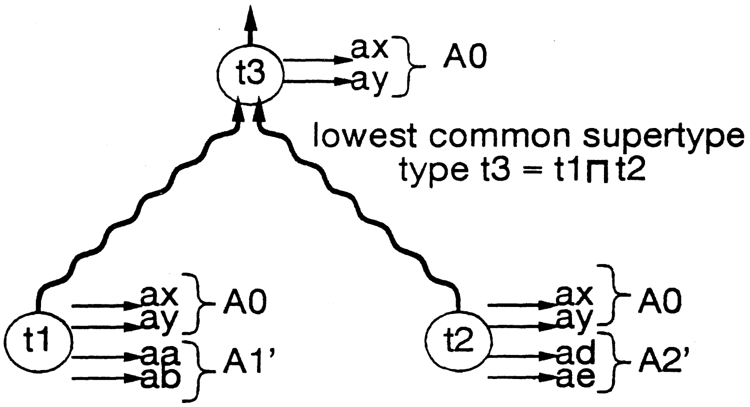


Figure 12: Lowest Common Supertype of Two Types in a Type Hierarchy.

Definition 15. [**Type Closure**] *A type specialization hierarchy T is said to be closed under "⊓" if and only if for any two types t1 and t2 in T, there exists a third type t in T which has exactly all properties that are common to t1 and t2, i.e.,* $t = t1 \sqcap t2$ *with the "⊓" function defined in Definition 4.*

Since a class hierarchy G has an underlying type hierarchy T, we now extend Definition 15 from types to classes.

Definition 16. [**Class Hierarchy Closure**] *A class hierarchy G is said to be closed under ⊓ if and only if for any two classes C1 and C2, there must exist a third class C3 in G with*

1. *type(C3) = type(C1) ⊓ type(C2) and*
2. *content(C3) ⊇ contents(C1) ∪ contents(C2).*

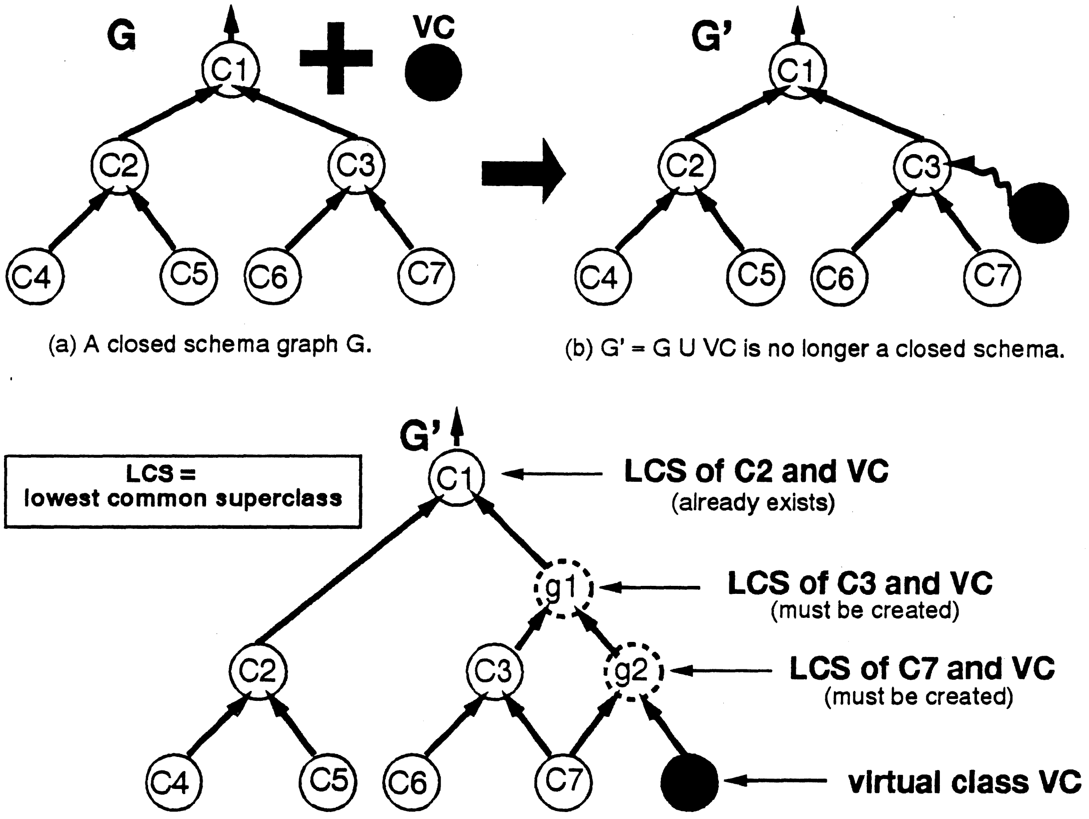
Definition 16 is a direct consequence of Definition 15. By Definition 15, for any two classes C1 and C2, there must exist a third class C3 in G with the type description of C3 equal to the lowest common supertype of type(C1) and type(C2). Since *C3* is a supertype of both *C1* and *C2*, *C3* must be a superclass of *C1* and *C2* in the class hierarchy. This then implies the second condition; namely, the subset relationships *C3* ⊇ *C1* and *C3* ⊇ *C2* imply *C3* ⊇ *C1* ∪ and *C2*. If two types (classes) don't share any common attributes, then their common lowest supertype (superclass) is the general root type (class) of the hierarchy. It is fairly easy to see that if a class hierarchy is closed then there is no problem with the type inheritance.

For the remainder of this work, we assume that the type hierarchy and the class hierarchy are closed under the "⊓" operator. We then will show that the lattice structure of the type hierarchy and of the class hierarchy can be maintained when inserting new classes. Put differently, we will show how to assure correct type inheritance when inserting new classes. This approach is effective in the design of views for complex applications, since it is generally difficult to proceed linearly top-down (by first introducing the most general types and then repeatedly specializing them into several subtypes, etc.) or bottom-up (by introducing the most specific types and then repeatedly finding generalizations of existing types). On the contrary, when a new virtual class is being specified for a view schema, then the view definer has no knowledge about which (super)-classes will be required by other views later on. It therefore is more natural to introduce types at the intermediate level and to either specialize downward or generalize upward, whenever needed. Intermediate types generated by the system that are not of interest to the view definer can be dropped (hidden) at the end of the view definition process, while all others are kept explicitly.

6.2 Using the Closure Property for Class Integration

In this section, we want to determine how to keep a schema graph G closed after the insertion of a new virtual class VC, i.e., how to maintain its lattice structure. This is done by coercing the generation of the lowest common superclasses required by the closure property of the schema graph (Definitions 15 and 16).

We shortly explain the requirements of the closure property on class integration using an example before presenting a more formal treatment. In Figure 13.a, we depict a closed schema graph G and a virtual class VC to be inserted into G. Figure 13.b then demonstrates an attempt to integrate VC into G. Note that the resulting schema graph G' is not necessarily closed, since the lowest common superclasses g_i for the virtual class VC and each of the existing classes C_i in G may not exist in G'. Hence, in order for G' to be closed, we have to insert all required lowest common superclasses. In the example in Figure 13.c, for instance, we had to insert the lowest common superclasses g_1 = C3 ⊓ VC and g_2 = C7 ⊓ VC. On the other hand, we did not have to insert the lowest common superclass of C2 and VC, since C2 ⊓ VC is equal to C1 which obviously already exists in G.



(a) A closed schema graph G.

(b) G' = G U VC is no longer a closed schema.

(c) Generating hidden classes to make G' a closed schema graph.

Figure 13: The Necessity of Creating Intermediate Classes.

Lemma 1. [**Necessity**] *Given a closed schema graph G=(V,E) and a new class VC that is to be integrated into G. Assume that the result of this integration is G'=(V',E'). By Definition 15, the following types t_i' must exist in G':*

$$(\forall t_i \in G)(\exists t_i' \in G')(t_i' = t_i \sqcap type(VC))$$

in order for G' to be closed.

By Definition 16, the following classes g_i must exist in G':

$$(\forall C_i \in G)(\exists g_i \in G')(type(g_i) = type(C_i \sqcap VC) \wedge content(g_i) \supseteq content(C_i) \cup content(VC)).$$

in order for G' to be closed.

Proof: The first part of Lemma 1 follows directly from Definition 15, which states that for any pair of types *t1* and *t2* in G, their lowest common supertype t = $t1 \sqcap t2$ must also exist in G. As a matter of course, this condition must hold for all pairs consisting of VC and any of the existing types. The second part of Lemma 1 follows directly from Definition 16, which states that for every pair of classes *C1* and *C2* in G their lowest common superclass *C3* must also exist in G. **q.e.d.**

Lemma 1 indicates the **necessity** of certain types (classes) to be created, when inserting a new class VC into a schema G. The creation of these new classes $g_i = C_i \sqcap VC$ with $C_i \in G$ may recursively cause the creation of additional classes, namely, the lowest common superclasses defined by $(g_i' = C_i \sqcap g_i)(\forall C_i \in G)$. Missikoff and Scholl [13] prove

that the first set of types g_i as specified in Lemma 1 is **sufficient** to guarantee the closure of the resulting type hierarchy. This work is directly applicable to our research, and as shown below we extend this sufficiency criteria from the type hierarchy to the class hierarchy.

Lemma 2. [**Sufficiency for Types**] *Given a closed type hierarchy TG=(TV,TE) and a new type t_{VC} to be integrated into TG. We denote the result of this integration by TG'=(TV',TE'). Then the integration of the types t_i' into TG' defined by*

$$GG = \{t_i' \mid t_i' = t_i \sqcap t_{VC} \text{ and } t_i \in TG\}$$

and

$$TV' = TV \cup GG \cup t_{VC}$$

will result in a closed type hierarchy TG'=(TV',TE').

Lemma 2 states that integrating the newly generated types t_i' into G does not cause the generation of additional new types. It indicates that the types t_i' obtained by the first iteration are **sufficient** for assuring the closure of the resulting schema graph TG'. In particular, it suggests that the computation of the new types t_i' can be done in a single pass - without recursive iteration over the newly generated types t_i'. The proof for Lemma 2 can be found in [13], and thus is not repeated here.

We are interested in extending this result from the type hierarchy to the class hierarchy. Note we now deal with a class, which is a two-facet concept consisting of both an associated type and a set content. We are hence concerned with two tasks: First,. the adjustment of the underlying type hierarchy such as to make it closed, and second, the determination of the correct set contents for these newly generated classes g_i such as to maintain the consistency of the schema graph.

Lemma 3. **[Sufficiency for Classes]** *Given a closed schema graph G=(V,E) and a new class VC to be integrated into G (Definition 15). The result of this integration defined by*

$$V' = V \cup VC \cup GG$$

with

$$GG = \{g_i \mid g_i = C_i \sqcap VC \text{ and } content(g_i) = content(C_i \cup VC) \text{ and } C_i \in G\ \}$$

represents a closed schema graph G'=(V',E').

Proof: We divide the proof into the following three cases: (a) both classes in V, (b) one class in V and one in GG, and (c) both classes in GG.

case a. We show that for $C_i \in$ V and $C_j \in$ V, the class $C_i \sqcap C_j$ already exists in V'.

This is true by assumption, since G=(V,E) being a closed schema graph implies that $C_i \sqcap C_j$ in V. And, V' is a superset of V.

case b. We show that for $g_i \in$ GG and $C_j \in$ V, the class $g_i \sqcap C_j$ exists in V'.

$g_i \sqcap C_j$	
$= (C_i \sqcap VC) \sqcap C_j$	by definition of GG
$= (VC \sqcap C_i) \sqcap C_j$	by commutativity of $\sqcap$
$= VC \sqcap (C_i \sqcap C_j)$	by associativity of $\sqcap$
$= VC \sqcap C_k$	by G closed, $\exists C_k = C_i \sqcap C_j \in V$
$= g_k \in$ V'	by definition of GG

This demonstrates that the newly generated classes g_i do not cause the generation of new classes when combining them with existing classes C_j, since the classes $g_i \sqcap C_j$ are already in GG.

case c. We show that for $g_i \in$ GG and $g_j \in$ GG, the class $g_i \sqcap g_j$ already exists in V'.

$g_i \sqcap g_j$	
$= (C_i \sqcap VC) \sqcap (C_j \sqcap VC)$	by definition of GG
$= (VC \sqcap VC) \sqcap (C_i \sqcap C_j)$	by commutativity and associativity of $\sqcap$
$= VC \sqcap (C_i \sqcap C_j)$	by idempotence of $\sqcap$
$= VC \sqcap C_k)$	by G closed, $\exists C_k = C_i \sqcap C_j \in V$
$= g_k \in$ V'	by definition of GG

This demonstrates that newly generated classes g_i do not cause the generation of new classes when combining them with other newly generated classes g_j, since the classes $g_i \sqcap g_j$ are already in V'.

The three cases together then prove that G' is closed. **q.e.d.**

Lemma 3 states that the computation of the intermediate classes g_i required for the closure of the schema graph G' can be done in a single pass – without recursive iteration over the newly generated types g_i.

6.3 Minimizing the Generation of Intermediate Classes

Next, we discuss how to limit the number of intermediate classes g_i generated for assuring the closure of a schema graph after class integration. This work is again based on [13].

Definition 17. *Given a type hierarchy TG=(TV,TE) and a new type t_{VC} to be integrated into TG. Let $\equiv_{t_{VC}}$ be an equivalence relationship defined by*

$$(\forall t_i, t_j \in TV)((t_i \equiv_{t_{VC}} t_j) \Longleftrightarrow (t_i \sqcap t_{VC} = t_j \sqcap t_{VC})).$$

Also we define the set of required lowest common supertypes in G with respect to $\equiv_{t_{VC}}$ by

$$GG = \{t'_i \mid t'_i = t_i \sqcap t_{VC} \wedge t_i \in G\}$$

We divide the set of types TV of TG into equivalence groups G_i for i = 1, ..., $|GG|$ with

$$G_i = \{t_j \mid ((t_{VC} \sqcap t_j) = t'_i) \wedge (t_j \in TV)\}$$

with $t'_i \in GG$ some fixed type per group G_i.

Definition 17 defines two types to be equivalent with respect to VC if and only if both types have the same lowest common supertype with respect to VC. An equivalence group G_i then is composed of all types t_i that have the same lowest common supertype t'_i with respect to VC.

Theorem 1. *Given a closed type hierarchy TG=(TV,TE) and a new type t_{VC} to be integrated into TG. Let $\{G_i\}$ denote the set of equivalence groups of G with respect to $\equiv_{t_{VC}}$. For each equivalence group G_i, there is one type $t_i \in G_i$ that is* **minimal** *and* **unique** *in G_i.*

The proof of correctness for Theorem 1 can be found in [13]. For a type t_i in G_i to be **minimal** means that it is a supertype of all other types in the equivalence group G_i. For t_i in G_i to be **unique** and **minimal** means that it is the only type that is a supertype of all other types in G_i, i.e., it is root type of the subgraph representing G_i. We denote this **unique** and **minimal** member t_i of G_i by rep(G_i). Next, we extend Definition 17 and Theorem 1 from types to classes.

Definition 18. *Given a schema graph G=(V,E) and a new class VC to be integrated into G. Let $\equiv_{VC}$ be an equivalence relationship defined by*

$$(\forall C_i, C_j \in V)((C_i \equiv_{VC} C_j) \Longleftrightarrow (type(C_i \sqcap VC) = type(C_j \sqcap VC))).$$

Also we define the set of required lowest common supertypes in G with respect to $\equiv_{VC}$ by

$$GG = \{t_i \mid t_i = type(C_i \sqcap VC) \wedge C_i \in G\}$$

We divide the set of classes V of G into equivalence groups G_i for $i = 1, \ldots, |GG|$ with

$$G_i = \{C_j \in V \mid (type(VC \sqcap C_j) = t_i) \wedge (C_j \in V)\}$$

with t_i some fixed type per group G_i.

Theorem 2. *Given a closed schema graph G=(V,E) and a new class VC to be integrated into G. Let $\{G_i\}$ denote the set of equivalence groups of G with respect to $\equiv_{VC}$. For each equivalence group G_i, there is one member class $C_i \in G_i$ that is* **minimal** *and* **unique** *in G_i.*

The proof of correctness for Theorem 2 can be directly derived from Theorem 1, and thus is omitted here. For a class C_i to be **minimal** in G_i means that it is a supertype and a superset of all other classes in the equivalence group G_i. For C_i in G_i to be **unique** and **minimal** means that it is the only class that is a superclass of all other classes in G_i, i.e., it is root class of the subgraph representing G_i. We denote this **unique** and **minimal** member C_i of G_i by rep(G_i).

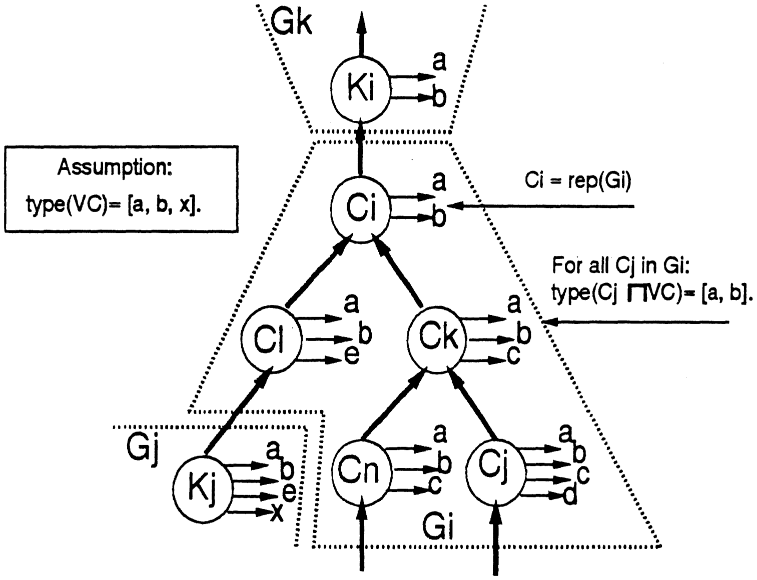


Figure 14: Partitioning of G Using the Equivalence Relation $\equiv_{t_{VC}}$.

Example 12. *In this example, we explain the concepts introduced in Theorems 1 and 2 based on Figure 14. In Figure 14, the two classes C_i and C_k are equivalent with respect to VC, since both have the same lowest common supertype [a,b]: type($C_i \sqcap VC$) = [a,b] $\sqcap$ [a,b,x] = [a,b] and type($C_k \sqcap VC$) = [a,b,c] $\sqcap$ [a,b,x] = [a,b]. All classes with this same lowest common supertype are equivalent and thus form an equivalence group. For instance, the classes C_i, C_l, C_k, C_n, and C_j have the same lowest common supertype [a,b], and they form the equivalence group G_i. The class K_j does not belong to the equivalence group G_i, because type($K_j \sqcap VC$) = [a,b,x] $\neq$ [a,b] = type(rep(G_i)).*

It is straightforward to see that the equivalence function $\equiv_{VC}$ partitions G into a set of non-overlapping subgraphs G_i. This is so since for all $C_i \in V$, $C_i \sqcap VC$ is equal to exactly one type t_i. This type t_i then determines the membership of C_i in one and only one group G_i.

Lemma 4. *Given a closed schema graph G =(V,E) and a class VC that is to be integrated into G. Then the result of this integration defined by*

$$V' = V \cup VC \cup GG$$

and

$$GG = \{g_i \mid type(g_i) = type(rep(G_i) \sqcap VC) \text{ and}$$

$$(content(g_i) = content(rep(G_i)) \cup content(VC))^3 \text{ and}$$

$$(G_i \text{ an equivalence group in } G \text{ with respect to } \equiv_{VC})\ \}.$$

corresponds to a closed schema graph G'=(V',E').

Proof: By Lemma 3, the following intermediate classes g_i must exist in G (or must be created in G'):

$(\forall C_i \in G)(\exists g_i \in G')$ $(type(g_i) = type(C_i \sqcap VC)$ and $content(g_i) \supseteq content(C_i) \cup content(VC))$.

By Definition 18, all classes in an equivalence group G_i have the same lowest common supertype t_i with respect to VC. Therefore, we only need to create one intermediate class g_i for each equivalence group G_i. More precisely, the following intermediate classes g_i must be created:

$(\forall G_i \in G)(\exists g_i \in G')$ $(type(g_i) = type(rep(G_i) \sqcap VC)$ and $content(g_i) = content(rep(G_i) \cup VC))$.

q.e.d.

Lemma 5. *Given a closed schema graph G =(V,E) and a class VC that is to be integrated into G. Assume that this integration of VC into G forces the creation of intermediate classes GG = \{ g_i | i = 1, ..., m \}. If an intermediate class $g_i \in GG$ already exists in G, then g_i is a member of the equivalence group G_i of G with $(\forall C_i \in G_i)(type(C_i \sqcap VC) = type(g_i))$. In fact, the type of g_i would be equal to the type of $rep(G_i)$.*

Proof: Lemma 5 can be explained as follows. $g_i \in$ GG means that g_i is a lowest common superclass for some group G_i. For all classes C_i in G_i, $type(C_i \sqcap VC) = type(g_i)$. This observation together ($g_i \in$ G) imply that $g_i \in G_i$. Obviously, g_i corresponds to the smallest type in G_i. Hence, $type(g_i)$=$type(rep(G_i))$.

q.e.d.

[3] When dealing with an *is-a* compatible schema graph, then VC being a subtype of these classes would also be a subset of rep(G_i). Hence, $content(rep(G_i)) \cup content(VC) = content(rep(G_i))$

From Lemma 5 we can conclude that the existence of a class C in G with a type equal to g_i's type can be determined by checking whether type(rep(G_i))=type(C) for each G_i of G. Hence, we can assure that if an intermediate class $g_i \in$ GG exists in G, then we can easily find it and thus won't unnecessarily create redundant ones.

Next, we show that all classes that are members of the same equivalence group G_i correspond to a connected subgraph of G with C_i=rep(G_i) the root of the subgraph.

Theorem 3. *Given a schema graph G=(V,E) and a new class VC to be integrated into G. For each equivalence group G_i of G, the classes $C_j \in G_i$ form a connected subgraph of G. More formally, for all $C_j \in G_i$, all classes $C_k \in V$ with (C_j is-a * C_k) and (C_k is-a * rep(G_i)) must also be members of G_i.*

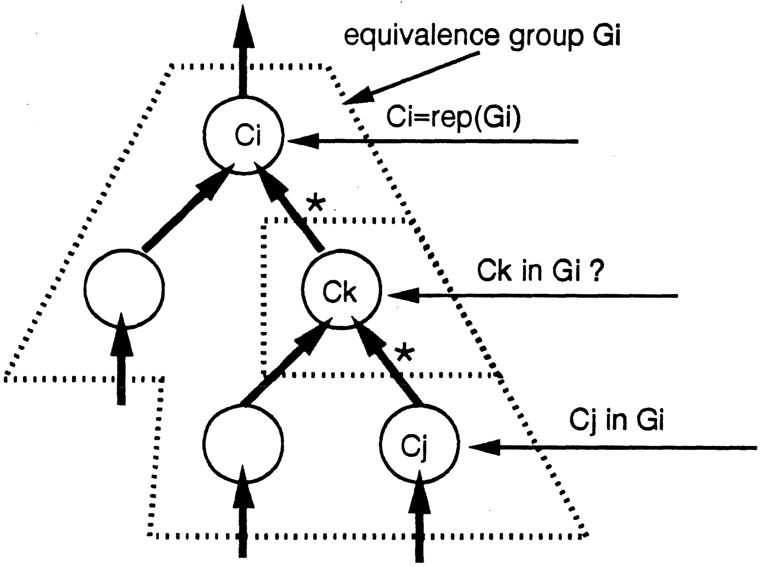


Figure 15: An Equivalence Group G_i Forms a Connected Subgraph of G.

Proof (By contradiction): By Theorem 2, for all $C_j \in G_i$, (C_j *is-a* * rep(G_i)) holds because rep(G_i) is minimal in G_i. In other words, each class $C_j \in G_i$ is a subclass of the unique representative rep(G_i) of G_i. Next, we show that if there is a class C_k between $C_j \in G_i$ and rep(G_i), then $C_k \in G_i$. The argument below is based on the situation depicted in Figure 15.

Assumption: Let $C_k \in V$ be a class with (C_j *is-a* * C_k) and (C_k *is-a* * rep(G_i)) and $C_k \notin G_i$.

(a) (C_j *is-a* * C_k) implies ($C_j \preceq C_k$). And ($C_j \preceq C_k$) and (type($C_j \sqcap VC$)=type(g_i)) imply (($C_k \sqcap VC) \succeq g_i$).

(b) (C_k *is-a* * rep(G_i)) implies ($C_k \preceq$ rep(G_i)). And ($C_k \preceq$ rep(G_i)) and (type(*rep*(G_i)$\sqcap$ VC)=type(g_i)) imply (($C_k \sqcap VC) \preceq g_i$).

(c) By (a) and (b), we have (($C_k \sqcap VC) \preceq g_i$) and (($C_k \sqcap VC) \succeq g_i$). This then implies (($C_k \sqcap VC) = g_i$). We thus have shown C_k to be a member of the equivalence group G_i. This is a contraction to the assumption $C_k \notin G_i$. **q.e.d.**

6.4 Interconnecting Intermediate Classes

Let f() denote the function defined by $f(t_j) = (t_j \sqcap VC) = g_j$. Then, the reverse function $f^{-1}()$ is defined by $f^{-1}(g_j) = t_j$ with t_j the canonical representative of the group G_i. Lattice properties that lead to the interconnection of these intermediate classes are discussed next based on ([13], Lemma 4.4 and Theorem 4.2).

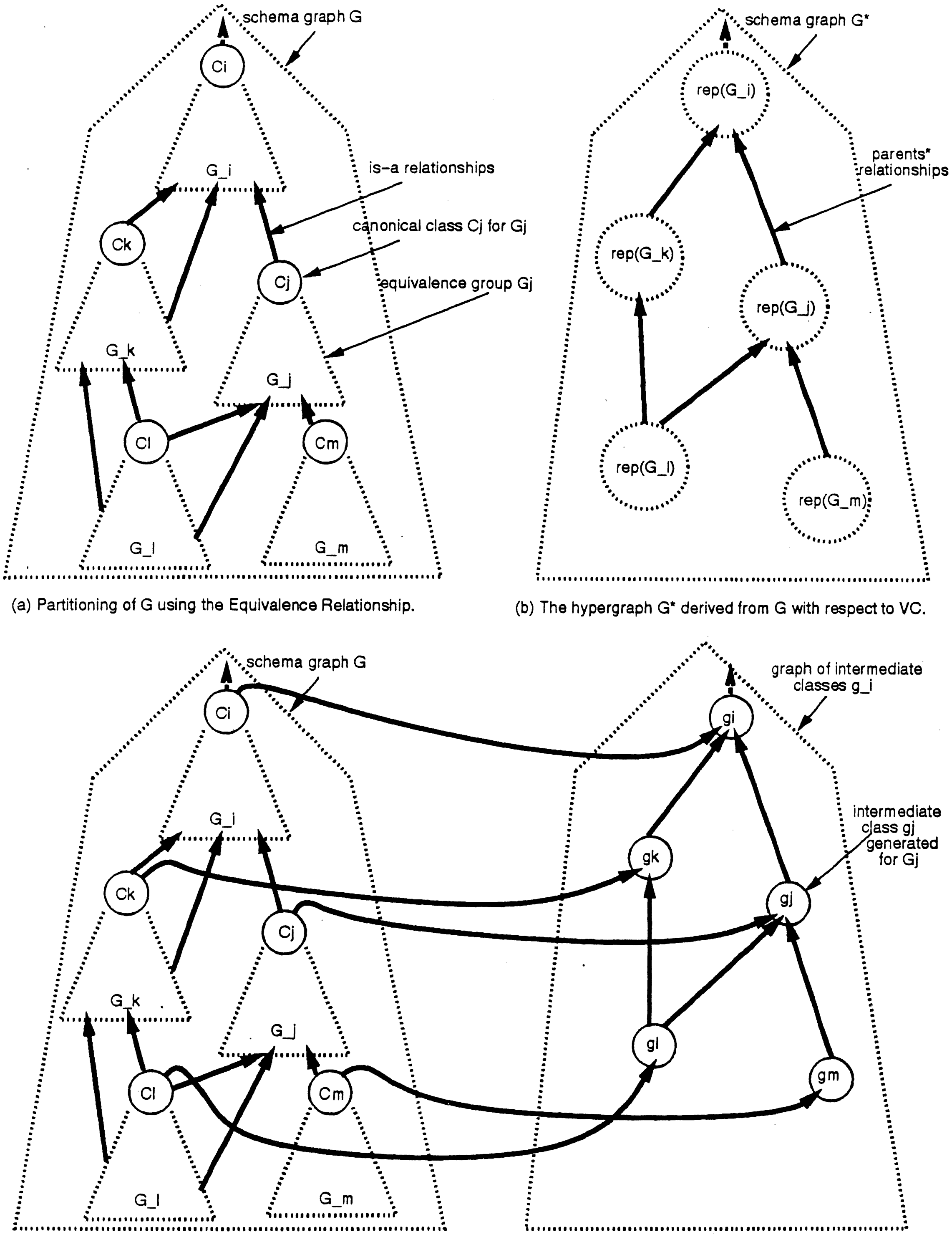
Definition 19. *Given a schema graph G=(V,E) and a class VC to be inserted into G. By Theorem 3, the equivalence relation* $\equiv_{VC}$ *defines a partition GG of equivalence groups* G_i *on G of size m. We define* $G^*=(V^*,E^*)$ *to be a schema graph with* $V^*=GG$ *and* E^* *the set of edges* $e=< G_i, G_j >$ *with* $G_i, G_j \in V^*$ *and* $(\exists C_i \in G_i)$ $(\exists C_j \in G_j)((C_i$ *is-a* ^{d} $C_j)$ *in G).*

G* corresponds to the set of class representatives modulo $\equiv_{VC}$. Put differently, G* is a hypergraph on G since each node in G* is equal to an equivalence group G_i. For C_i, $C_j \in G$, we say that C_j is a parent* of C_i, denoted by parent*(C_i)=C_j, if C_j=rep(G_j) and there is an edge e=$< G_i, G_j >$ in G*. In [13], it is shown that G* is isomorphic to the graph GG defined above.

Theorem 4. *Given a closed schema graph G=(V,E) and a class VC to be inserted into G. The integration of a class C with type(C)=type(VC) and arbitrary set content results in a closed schema graph G'=(V',E') if we add the set of classes* $GG = \{g_i\ \}$ *as defined in Lemma 1. In addition, the following is-a edges must be established:*

1. *Each class* $g_i \in GG$ *has a single child in G which is the canonical representative of the group to which* $f^{-1}(g_i)$ *belongs. Put differently, we add the edge* $e = <rep(G_i),g_i>$ *between each* $rep(G_i)$ *and the corresponding generated class* $g_i=rep(G_i) \sqcap VC$.
2. *For all* g_i, g_j *in GG, we add the edge* $e = <g_i,g_j>$ *if and only if* $C_i=rep(G_i)$ *and* $C_j=rep(G_j)$ *and* C_j *is a direct parent* of* C_i *in G*.*

Note that the creation of intermediate classes g_i coerced by the type lattice problem is driven by the closure requirements of the type hierarchy. No requirements are made on the set content of these classes, since these types are hidden and not necessarily of interest to the database user. Therefore, we can specify the set aspect of these new intermediate classes as needed. By setting the content of the new intermediate class g_i equal to the content of the representative class rep(G_i) of $G_i \cup$ VC, we assure that g_i is the superset of all classes in G_i. By Theorem 2, g_i for G_i is also the supertype of all classes in G_i. This implies that g_i is indeed a superclass of all classes in G_i. Hence, the largest class in G_i, namely, rep(G_i) should be the direct child of g_i, which is stated in part 1 of Theorem 4. By part 2 of Theorem 4, two newly generated types g_i and g_j are only subtypes of one another if the corresponding representative classes C_i and C_j of the equivalence groups G_i and G_j are also subtypes of one another. If C_i and C_j are subtypes of one another in G then they must also be subsets of one another. Since the contents of g_i and g_j are equal to the contents of C_i and C_j, we can imply that g_i is a subclass of g_j. This then justifies part two of Theorem 4. A more detailed proof of Theorem 4 with respect to the type hierarchy can be found in [13].



(a) Partitioning of G using the Equivalence Relationship.

(b) The hypergraph G* derived from G with respect to VC.

(c) The final schema graph G with the type hierarchy prepared for the insertion of VC.

Figure 16: Connecting Intermediate Classes with G.

Example 13. *Figure 16.a shows a schema graph G with a partition induced by the class VC. The partition consists of the five equivalence groups G_i, G_j, G_k, G_l, and G_m, which have the representatives C_i, C_j, C_k, C_l, and C_m, respectively. The matching hypergraph G^* of G, in which each node in G^* corresponds to an equivalence group G_i in G, is depicted in Figure 16.b. Figure 16.c then shows the schema graph that results from integrating a class C with type(C) = type(VC) into G. First, for each equivalence group G_i in G, we create an intermediate class g_i as shown on the right hand side of Figure 16.c. By Theorem 4.a, each representative class C_i of G_i is connected to the respective class g_i=C_i⊓VC. This is depicted by the dark arrows going from the left to the right of Figure 16.c. By Theorem 4.b, the newly generated classes g_i are connected with one another as dictated by the relationships of the corresponding representatives classes C_i (as shown in Figure 16.b). For instance, the edge (g_i is-a g_j) is inserted into GG because the relationship (C_i is-a * C_j) holds in G.*

6.5 Class Integration After Type Preparation

In previous sections, we have demonstrated how to prepare a schema graph hierarchy for the insertion of a new class VC. Next, we need to handle the actual integration of VC into G. Contrary to the flexibility in arbitrary determining the set content of the intermediate classes g_i generated for type hierarchy preparation, the type and the set content of the virtual class VC are predetermined. Hence, the integration of the class VC itself must obey both the subtype and the subset relationships of VC with all other classes in the schema graph. The integration of VC into G can be characterized as follows.

Lemma 6. *Given a closed schema graph G=(V,E) and a class VC to be integrated into G. The integration of a class VC into G results in a closed schema graph G'=(V',E') if*

1. *first, we extend the class hierarchy G to include type(VC) using Theorem 4, and*
2. *second, we add VC into its correct location in G' by connecting it to its direct parents and direct children in G'.*

By Theorem 4, step 1 of Lemma 6 results in an extension of G that correctly incorporates a class with type(VC) into its class hierarchy. Once, a class with its the type equal to VC is present in G, class integration of VC becomes a matter of finding the correct location for VC in G. Details on finding the correct position of VC given a prepared type hierarchy are presented in a later section. We will use the lemma above for implementing the integration process as explained in Section 8.

We have discussed the creation of intermediate classes and the associated arcs required to consistently integrate a virtual class VC into a schema graph G. The discussion and hence the solution are driven by the type inheritance problem (Section 5). We now need to assure that the edges created are also sufficient in terms of capturing the subset relationships between all classes. It is easy to see that all suggested edges are correct and non-redundant. It is not necessarily clear whether they are also sufficient in capturing all subset relationships. Edges could not be missing between the original graph G and the virtual graph GG consisting of intermediate classes g_i, since we assumed G to be complete and showed GG to be complete. It is relatively straightforward to show that no additional edges from G to GG can exist. At present, we are not able to prove that no additional edges from GG to G need be added to complete all information on the combined schema graph. If this is the case, then we would apply the algorithm described in Section 7 to add these extra direct-parent relationships for each intermediate class g_i. Since this can be done in linear time, this will not change the complexity of the type hierarchy preparation algorithm.

6.6 The Type Hierarchy Preparation Algorithm

In the previous sections, we have described the theory underlying the preparation of a class hierarchy for the insertion of a new class. Based on these results, we now develop an algorithm to solve this problem. This algorithm creates the additional intermediate classes required by the insertion of a new type into a schema graph. This algorithm, a direct extension of ([13], page 77-80), handles both the type and set content of the class concept while Missikoff's work focuses on type classification.

For the following, we assume that the graph G is represented by a table with sorted rows (one for each class) and with the following four columns, the name of the class C, the set of parents of C in G, a label "*" to mark members of G*, and the set of parents* of C in G*. The former two are given initially and the later two are generated by the algorithm. We also assume that the rows are sorted according to the generalization relationship. The notation f() is again used to denote the function f(C) = C ⊓ VC.

The Compute-G*(G,VC) procedure in Figure 17 computes the hypergraph G*. In particular, it computes the representative class *rep(Gi)* for each equivalence group G_i with respect to VC. By Theorem 2, a class C_i is a representative of an equivalence group G_i if and only if it is the highest class in the group G_i. This means that all its parents must belong to different equivalence groups. This is exactly what is tested by statement (2) of the procedure. These unique representatives C_i=*rep*(G_i) are marked by the label "*" and they are said to belong to G*. By Lemma 4, each of them will trigger the generation of a new intermediate class g_i with type(g_i) = C_i ⊓ VC.

The Compute-G*(G,VC) procedure also finds the parents of each unique representative class C_i in the hypergraph G*, denoted by parents*(C_i). These parents*(C_i) correspond to the canonical representatives of the equivalence groups G_i that the class C_i is directly *is-a* related to. Put differently, parents*(C_i) corresponds to the set of all parents of C_i in G*. For a class C, if any of its parents C_k are marked then these parents in G are also its parents* in G*. However, if a parent C_k of C is not marked, then we need to find the parent* of C higher in the graph. If we could assume that the parents* of all classes above the current class C have already been determined before attempting to calculate parents* of C, then we could simply set parents*(C) equal to parents* of C_k. Indeed, this is assured by scanning the list of classes ordered according to the generalization relationship among classes from left to right.

Next, the Generate-Intermediate-Classes(G,VC) procedure (Figure 17) is applied to construct all required intermediate classes g_i and interconnects them with one another and with the classes in G. The type of the new class g_i is determined by the lowest common supertype operator ⊓ while the set membership of g_i is determined by setting the content of g_i equal to the content of the respective rep(G_i) ∪ VC. Note that this represents an important extension to the algorithm in [13] since we generate a complete class (rather than just a type).

Algorithm outline: Generation of Intermediate Classes.

Input:

A schema G = (V,E) with possibly multiple inheritance.
A class VC to be integrated into G.

Output:

The schema G augmented by all intermediate classes required by the integration of VC into G.

Algorithm:

```
procedure Generate-Intermediate-Classes(G,VC)
begin
    (1) Compute-G*(G,VC);
    (2) for all $C \in G*$ do
        if $(C \sqcap VC) \neq type(C)$ then
            $type(g) = C \sqcap VC$;
            $contents(g) = contents(C) \cup contents(VC)$;
            $V = V \cup \{ g \}$;
            for all $p \in f(parents^*(C))$
                add the edge (g is-a p) to G.
            endfor
            add the edge (C is-a g) to G.
            if $parents(g) \cap parents(C) \neq \{\emptyset\}$ then
                for all $p \in parents(g)$ remove the edge (C is-a p) endfor
            endif
        endfor
end procedure

procedure Compute-G*(G,VC)
begin
    for all $C \in G$ do
        (1) $parents^*(C) = parents(C)$;
        (2) if $(\forall\ Ck \in parents(C))\ (C \sqcap VC \neq Ck \sqcap VC)$ then
                mark C by the label "*";
        (3) for all $Ck \in parents(C)$ do
            if Ck is not marked then
                $parents^*(C) = parents^*(C) - \{ Ck \}$;
                for all $Cj \in parents^*(Ck)$ do
                    if $parents^*(C) = \{\emptyset\}$ then $parents^*(C) = \{ Cj \}$;
                    else if $(\forall\ Ci \in parents^*(C)(Cj$ is not a supertype of $Ci)$
                        then $parents^*(C) = parents^*(C) \cup \{ Cj \}$;
                endfor
            endfor
    endfor
end procedure
```

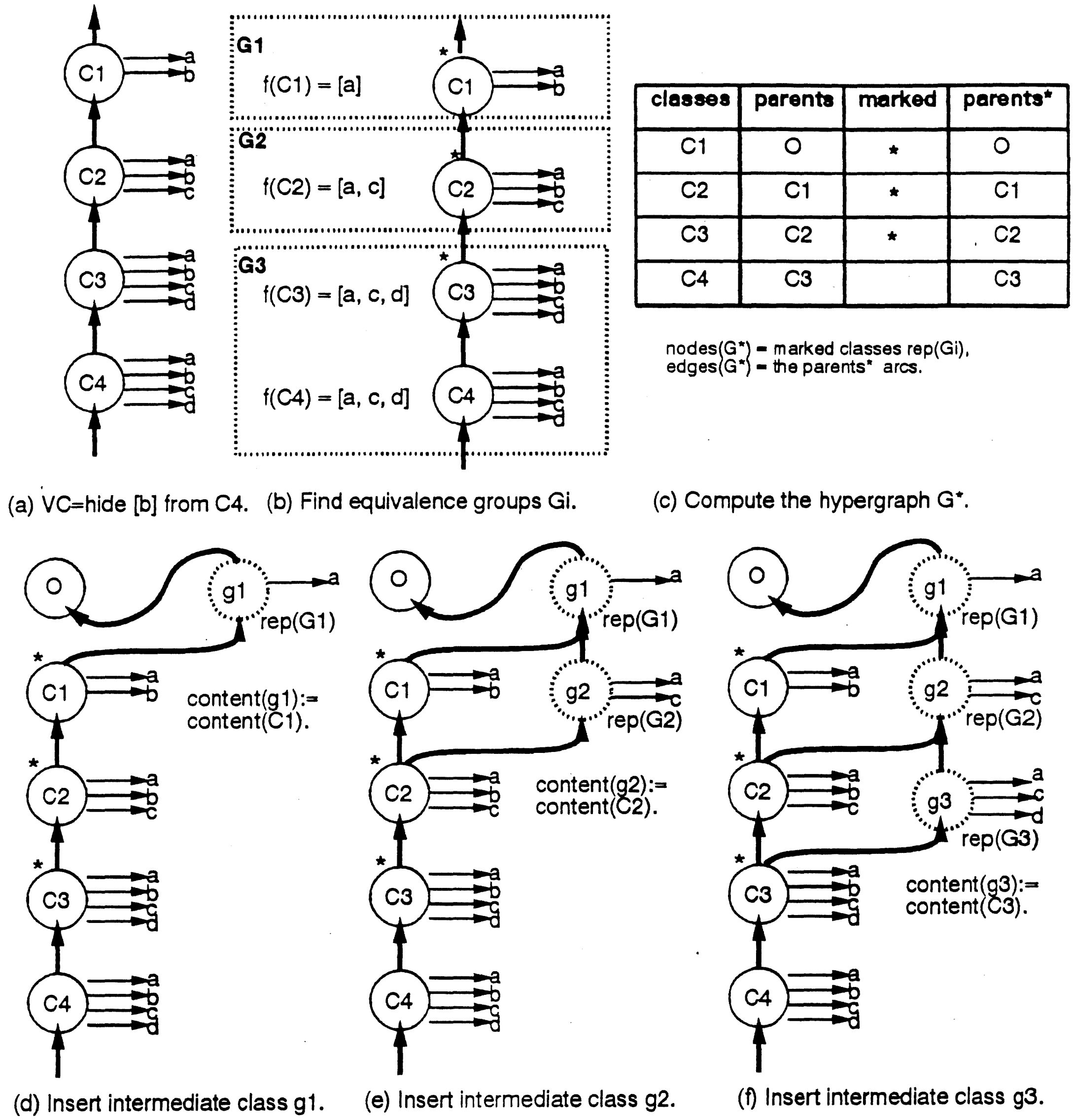
Figure 17: The Type-Hierarchy-Preparation Algorithm.

For each class C in $G*$, we check whether the condition $(C \sqcap VC \neq type(C))$ evaluates to false. This would mean that an intermediate class g with type(g) = C ⊓ VC (i.e., the required type) exists in G and therefore need not be generated. In this case, the class C is already properly connected since the original schema graph is assumed to be closed. Thus the if-statement is skipped. If, on the other hand, the condition $(C \sqcap VC \neq type(C))$ evaluates to true, then the intermediate class g_i with type(g) = C ⊓ VC does not yet exist in G and therefore needs to be created. In this case we associate a new class g_i with type(g_i) = C ⊓ VC and content(g_i) = content(C) with C. In addition, we create an edge e=$< C, g_i >$ between C and g_i to make C the unique child of g_i. We also add edges from g_i to all its parents in the hypergraph G*. These parents* of g_i could be other classes g_k in $G*$ or existing classes C_i in G. This is done by creating the edges $e =< g_i, p >$ with p = f(parents*(C)).

In order to avoid redundant arcs, the procedure removes edges $e =< C, p >$ from the class C, if C shares any parents with its newly generated intermediate class g_i, i.e., if p ∈ parents(g_i). g_i becomes a direct parent of C and thus these arcs are redundant.

Example 14. *Figure 18 demonstrates how to apply the type classification algorithm given in Figure 17 to prepare a class hierarchy G for class insertion. Figure 18.a shows the schema G before type classification. Assume that the virtual class VC is derived by the view derivation "VC = hide [b] from C4". Then the type of VCis defined by type(VC)=[a,c,d] and the object membership of VCis defined by content(VC) = content(C4). The Generate-Intermediate-Classes(G, VC) procedure first calls the Compute-G*(G, VC) procedure to compute the hypergraph G*. There are three equivalence groups with respect to VC, namely, G_1 with the lowest common supertype of [a], G_2 with the lowest common supertype of [a, c], and G_3 with the lowest common supertype of [a, c, d]. The outer for-loop of the Compute-G*(G, VC) procedure steps through the list of all classes in G. When processing the first class C1, step (2) of the for-loop marks the class C1, since its parent is the general root of the schema. Step (3) checks whether all parents of C1 are marked, and since they are, parents*(C1) = parents(C1) is not modified. The second iteration of the for-loop marks the class C2, since C2 and C1 are in the different equivalence groups G_2 and G_1, respectively. Step (3) checks whether the parent of C2, which is C1, is marked. Since C1 is indeed marked, parents*(C2) = {C1} is not modified. The third iteration of the for-loop marks the class C3, since its only parent C2 is again in a different equivalence group. Since the parent of C3 is marked, the parents*(C3) set is not modified. The fourth and last iteration of the for-loop does not mark the class C4, since its parent C3 is in the same equivalence group G_3 as C4. Step (3) then checks whether the parent of C4, which is C3, is marked. Since C3 is marked, parents*(C4) = {C3} is not modified. This completes the computation of the hypergraph $G*$ on G. The equivalence groups G_1, G_2 and G_3 and their marked representatives rep(G_1)=C1, rep(G_2)=C2, and rep(G_3)=C3 are shown in Figure 18.b. The hypergraph G*, i.e., the marked classes C_i = rep(G_i) and their parents* relationships, are shown in Figure 18.c.*

The second step of the Generate-Intermediate-Classes(G, VC) procedure now generates the intermediate classes g_i for each equivalence group G_i. For the first marked class C1, it checks whether a class g_1 with type(g_1) = (C1 ⊓ VC) exists in the schema. Since it does not, g_1 is created with type(g_1) = C1 ⊓ VC = [a] and content(g_1) = content(C1) as shown in Figure 18.d. We add edges from g_1 to the generated intermediate classes of all its parents in G*. Since parents*(C1)= {O} and O ⊓ VC = O, this corresponds to the edge $< g_1, O >$. The edge $< C1, g_1 >$ becomes redundant and is removed. We also add the edge $< C1, g_1 >$ to connect g_1 to its canonical representative in the original schema graph G (Figure 18.d). The next marked*



classes	parents	marked	parents*
C1	O	*	O
C2	C1	*	C1
C3	C2	*	C2
C4	C3		C3

nodes(G*) = marked classes rep(Gi),
edges(G*) = the parents* arcs.

(a) VC=hide [b] from C4. (b) Find equivalence groups Gi. (c) Compute the hypergraph G*.

(d) Insert intermediate class g1. (e) Insert intermediate class g2. (f) Insert intermediate class g3.

Figure 18: An Example of Class Hierarchy Preparation.

class in the list in Figure 18.c is C2. We check whether a class g_2 with type(g_2) = (C2 $\sqcap$ VC) exists in the schema. Type(g_2) = [a, c] $\neq$ [a, c, d] = type(C2) implies that a class with the required type of g_2 does not exist in the schema. Hence, we create the class g_2 with type(g_2) = (C2 $\sqcap$ VC) = [a, c] and content(g_2) = content(C2) as shown in Figure 18.e. We add edges from g_2 to the intermediate classes g_i that are g_2's parents in G^*. Since parents*(C2) = $\{C1\}$ and C1 $\sqcap$ VC = g_1, this corresponds to the edge $< g_2, g_1 >$. The edge $< C2, g_2 >$ is also added to connect g_2 to its canonical representative in G. The resulting schema is depicted in Figure 18.e. Lastly, the algorithm processes the marked class C3 in a similar manner. The final result, the graph G prepared for the insertion of the virtual class VC, is given in Figure 18.f.*

In [13], the algorithm has been shown to be of quadratic complexity $O(m^2)$ with m the number of edges in G. Our extension of the algorithm, namely, the generation of a complete class (rather than just a type) does not influence this complexity. This analysis assumes of course that the *subsumes()* function can be calculated in constant time.

7 ALGORITHMS FOR CLASS PLACEMENT

In this section, we discuss the integration of a class into a given class hierarchy assuming that the type hierarchy has been properly prepared for class insertion as described in Section 6. The proposed placement algorithm handles both single- and multiple-inheritance schema graphs. We prove the correctness of the algorithm and show the algorithm to be of linear complexity (assuming a *subsumes()* function of constant complexity). For the following, we assume the existence of a function *subsumes(C1,C2)* that determines whether the *is-a* relationship (C2 *is-a* C1) exists. As described in Section 4, this function is computed by comparing the type description and the membership predicate of the two classes. This is different from checking whether the edge e = <C2,C1> exists in a schema graph G.

7.1 The General Class Placement Algorithm

By Definition 8, a schema graph captures all *direct is-a* relationships between pairs of classes C_1 and C_2, namely, (C_1 *is-a* ^{d} C_2), by directed graph edges e = <C_1,C_2>. Put differently, each class C_i in a schema graph G is connected to its direct sub- and super-classes via graph edges. *Indirect is-a* relationships (C_1 *is-a* * C_2) are derivable via the transitive closure on the graph edges. The integration of a new virtual class VC into the schema graph G=(V,E) thus requires the identification of the direct *is-a* relationships between the virtual class VC and all other classes in the global schema G as defined below.

Definition 20. *Given a schema graph G=(V,E) and a class VC. Then we defined the set of all classes in G that directly subsume VC, i.e., the direct superclasses of VC, by*

$$DIRECT\text{-}PARENTS_{VC} :=$$

$$\{C_i \mid (VC \textit{ is-a } C_i) \land (\nexists C_j \in V)(j \neq i)((VC \textit{ is-a}^* \; C_j) \land (C_j \textit{ is-a}^* \; C_i))\}.$$

Similarly, we define the set of all classes in G that VC directly subsumes, i.e., the direct subclasses of VC, by

$$DIRECT\text{-}CHILDREN_{VC} :=$$

$$\{C_i \mid (C_i \textit{ is-a } VC) \land (\nexists C_j \in V)(j \neq i)((C_i \textit{ is-a}^* \; C_j) \land (C_j \textit{ is-a}^* \; VC))\}.$$

Example 15. *Definition 20 is explained based on the schema graph given in Figure 20. In this figure, the labels* **sup** *and* **sup** *are associated with a node C_i that is a* **sup***erclass or a* **sub***class of VC, respectively.*

The DIRECT-PARENTS$_{VC}$ set contains all classes that fulfill the following conditions: (1) they are superclasses of VC, i.e., they have the **sup** *label, and (2) there are no other classes below them in the schema graph that are also superclasses of VC. The later means that they are the* **lowest** *possible classes still marked by the* **sup** *label. In Figure 20, the members of the DIRECT-PARENTS$_{VC}$ set, which are C3 and C9, are marked by horizontal strips.*

The DIRECT-CHILDREN$_{VC}$ set contains all classes that fulfill the following conditions: (1) they are subclasses of VC, i.e., they have the **sub** *label, and (2) there are no other classes* *above them in the schema graph that are also subclasses of VC. The later means that they are the* **highest** *possible classes still marked by the* **sub** *label. In Figure 20, the members of the DIRECT-CHILDREN$_{VC}$ set, which are C16 and C25, are marked by vertical strips.*

Based on Definition 20, the algorithm for finding the correct position for the class VC in the schema G=(V,E) can be summarized as follows. First, we find all classes in G that are direct superclasses of VC, namely, the set of classes DIRECT-PARENTS$_{VC}$ as defined in Definition 20. Next, we find all classes in G that are direct subclasses of VC, namely, the set of classes DIRECT-CHILDREN$_{VC}$ as defined in Definition 20. VC then is placed directly below all classes in the DIRECT-PARENTS$_{VC}$ set and directly above all classes in the DIRECT-CHILDREN$_{VC}$ set by adding new *is-a* edges to the schema graph G. The just described algorithm is given in Figure 19.

Algorithm outline: Placement of A Virtual Class into the Global Schema.

Input:
A schema G = (V,E) with possibly multiple inheritance.
A class VC to be integrated into G.

Output:
The schema G with VC integrated into G.

Data Structures:
variables DIRECT-PARENTS$_{VC}$, DIRECT-CHILDREN$_{VC}$: set of classes

Algorithm:

```
procedure Class-Placement-Algorithm(G,VC)
begin
   (1) Compute DIRECT-PARENTS$_{VC}$.
   (2) if $C_i$ ∈ DIRECT-PARENTS$_{VC}$ with ($C_i$ = VC) then STOP else V:=V∪VC; endif
   (3) Compute DIRECT-CHILDREN$_{VC}$.
   (4) Update-Edges$_{VC}$(G, VC, DIRECT-PARENTS$_{VC}$, DIRECT-CHILDREN$_{VC}$);
end procedure

procedure Update-Edges$_{VC}$(G, VC, DIRECT-PARENTS$_{VC}$, DIRECT-CHILDREN$_{VC}$)
begin
   (4.1) for all p ∈ DIRECT-PARENTS$_{VC}$ do
             create-edge(<VC,p>);
         end for
   (4.2) for all c ∈ DIRECT-CHILDREN$_{VC}$ do
             create-edge(<c,VC>);
         end for
   (4.3) for all p ∈ DIRECT-PARENTS$_{VC}$ do
             for all c ∈ DIRECT-CHILDREN$_{VC}$ do
                 if edge <c,p> exists then delete-edge(<c,p>) endif
             end for
         end for
end procedure
```

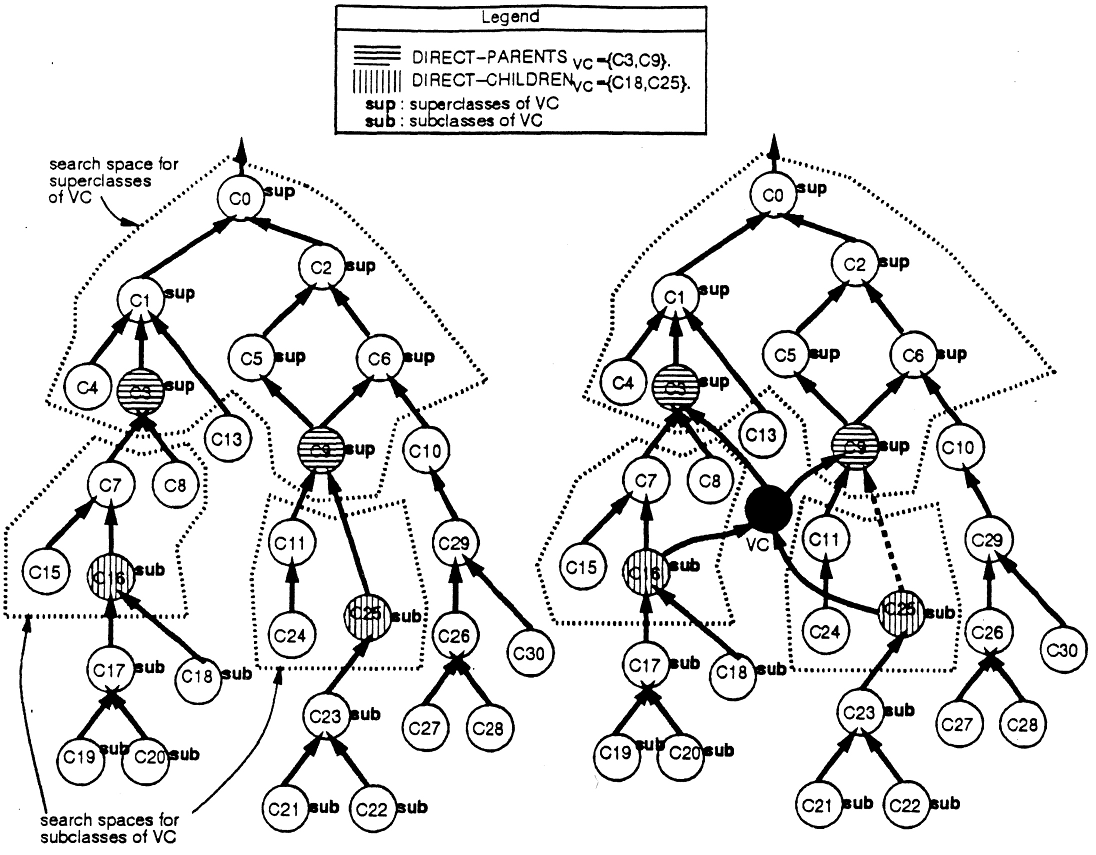
Figure 19: The Class-Placement Algorithm.

The Class-Placement algorithm shown in Figure 19 has the following steps. Step **(1)** of the algorithm computes the DIRECT-PARENTS$_{VC}$ set. As explained later, this is done by a depth-first downwards traversal of G starting from the root of G. If it finds a class in the DIRECT-PARENTS$_{VC}$ set that is equal to VC, then VC exists already in G and the algorithm terminates (step **(2)**). This equality of classes is determined by comparing their type descriptions and membership characteristics and not necessarily their class names. If a mismatch in class names exists, then the view designer may attach the new name of VC as synonym with the original name of the existing class C_i. For simplicity, we assume that (C_i=VC) is defined by *subsumes(C_i, VC)*=true and *subsumes(VC,C_i)*=true. If VC was not integrated in G to being with, then the else-branch of step (**2**) will now add the class VC to the set of classes V of G. Step (**3**) of the algorithm computes the DIRECT-CHILDREN$_{VC}$ set again by depth-first traversal of G. Lastly, step (**4**) updates the edges E of G so that the new class VC is now properly connected with the classes of G. This edge computation is described in detail in the Update-Edges procedure in Figure 19. The first for-loop creates edges between VC and all classes in the DIRECT-PARENTS$_{VC}$ set, i.e., it connects VC with its direct superclasses. The second for-loop creates edges between VC and all classes in the DIRECT-CHILDREN$_{VC}$ set, i.e., it connects VC with its direct subclasses. Finally, the third for-loop removes all edges from G that have become redundant due to the introduction of the edges listed above. Namely, all edges that directly connect classes in the DIRECT-PARENTS$_{VC}$ set with classes in the DIRECT-CHILDREN$_{VC}$ set have become redundant and thus are removed.

Example 16. *In this example, we demonstrate the Class-Placement algorithm given in Figure 19 on the example schema graph shown in Figure 20. The first step of the algorithm finds the DIRECT-PARENTS$_{VC}$ set by depth-first downwards traversal of G starting from the root C0. The search stops for a given branch if either a leaf node C_i is reached (e.g., for the class C4) or if the condition subsumes(C_i, VC) no longer holds (e.g., for the class C3). All classes at the borderline of this search space that still fulfill the superclass condition are put into the DIRECT-PARENTS$_{VC}$ set, in this case, the classes C3 and C9. The algorithm does not find any class C_i that is equal to VC (step* (**2**)*). The third step of the algorithm then computes the DIRECT-CHILDREN$_{VC}$ set by depth-first downwards traversal of G. The search stops for a given branch if either a leaf is reached (e.g., the classes C15 and C8) or if a class node C_i is reached for which the subclass condition subsumes(VC,C_i) is true (e.g., for the classes C16 and C25). All classes of the later category are placed into the DIRECT-CHILDREN$_{VC}$ set. Lastly, the fourth step of the algorithm updates the edges E of G so that the new class VC is now properly connected with the classes of G. First, we create the edges e1 = <VC,C3> and e2 = <VC,C9> to connect VC with all classes in the set DIRECT-PARENTS$_{VC}$={C3,C9}. Then, we create the edges e3 = <C18,VC> and e4 = <C25,VC> to connect all classes in the set DIRECT-CHILDREN$_{VC}$={C18,C25} with VC. There is one direct edge between classes in DIRECT-CHILDREN$_{VC}$={C18,C25} and those in DIRECT-PARENTS$_{VC}$={C3,C0}, namely, the edge <C25,C9>. Due to introduction of the edges <C25,VC> and <VC,C9>, this edge has become redundant and is thus removed.*

Theorem 5. (**Correctness**) *Given the schema G = (V,E) and a class VC, the Class-Placement algorithm shown in Figure 19 integrates VC into G with the resulting G representing a* **correct** *schema graph as defined in Definition 8.*

Proof: Definition 20 defines the direct superclasses and subclasses of the class VC in a set of classes S as DIRECT-PARENTS$_{VC}$ and DIRECT-CHILDREN$_{VC}$, respectively. For this proof, we assume that the Class-Placement algorithm (steps 1 and 3 in Figure 19) indeed calculates these two sets of classes correctly – as we will show later in this section. Then we only need to show the correctness of the fourth step, namely, of the manipulation of the graph edges to prove the correctness of the overall algorithm.



(a) Before the integration of VC into schema G.

(b) After the integration of VC into schema G.

Figure 20: An Example of Applying the Class-Placement Algorithm.

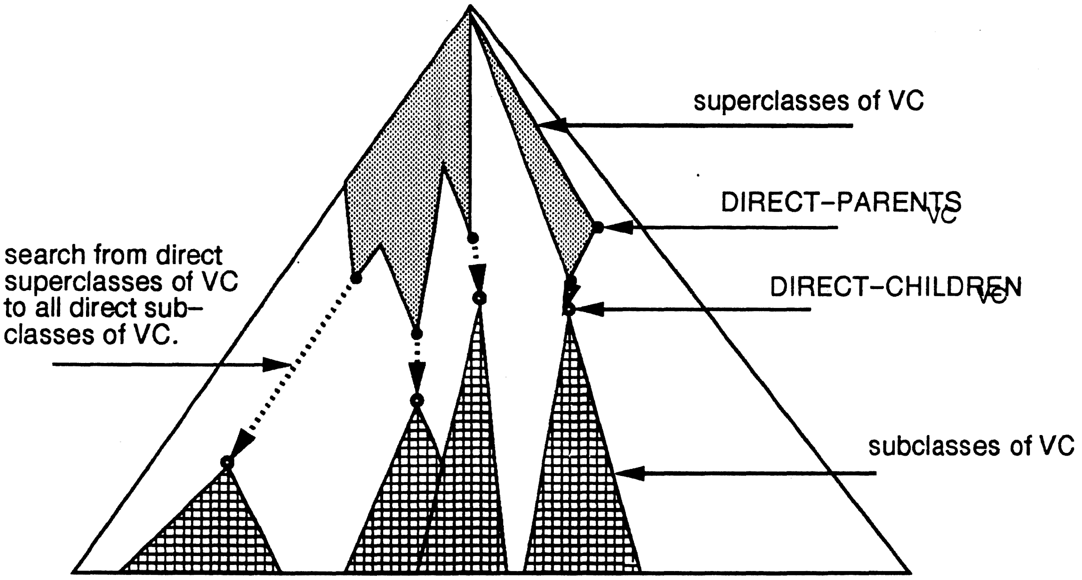

superclasses of VC
DIRECT-PARENTS VC
DIRECT-CHILDREN VC
search from direct superclasses of VC to all direct sub-classes of VC.
subclasses of VC

Figure 21: Search Space for Class Placement.

By Definition 8, a schema graph G=(V,E) is **correct** if it represents a set of classes S = $\{C_i | i = 1, ..., n\}$ and their *is-a* relationships in the following manner:

(1) V := S,

(2) E := { e = $<C_i,C_j>$ | (C_i *is-a* ^{d} C_j) in S and $\forall$ C_i, C_j $\in$ V }.

From the point of view of a class C_i, this is equivalent to each class C_i having an *is-a* edge e in G to all its direct super- and subclasses in S. More formally, ($\forall$ C_i $\in$ V)

($\forall$ p_j $\in$ V) ((C_i *is-a* ^{d} p_j) in S $\Longrightarrow$ $e_{ij} =< C_i, p_j >$ in E) $\wedge$

($\forall$ ch_k $\in$ V) ((ch_k *is-a* ^{d} C_i) in S $\Longrightarrow$ $e_{ki} =< ch_k, C_i >$ in E)

We assume that the input schema graph G=(V,E) is correct as defined above. If the new class VC is equal to one of the existing classes in G, then the Class-Placement algorithm in Figure 19 sets G':=G by **step 2.** G':=G is by assumption correct.

If the new class VC is not equal to any of the existing classes in G, then the Class-Placement algorithm constructs a new graph G'=(V',E') by adding VC to G. It is easy to see that the resulting graph G' will have the following characteristics:

(1) V' := V $\cup$ VC,

(2) E' := E $\cup$ DP $\cup$ DC - DPC with

DP := $\{e_{VC,j} =< VC, p_j > \mid p_j \in$ DIRECT-PARENTS$_{VC}\}$, and

DC := $\{e_{k,VC} =< ch_k, VC > \mid ch_k \in$ DIRECT-CHILDREN$_{VC}\}$, and

DPC := $\{e_{kj} =< ch_k, p_j > \mid ch_k \in$ DIRECT-CHILDREN$_{VC}$ $\wedge$ $p_j \in$ DIRECT-PARENTS$_{VC}\}$.

We now show that G' is correct, namely, that G' contains graph edges for **all** direct *is-a* relationships and no others, by examining the following four cases:

Part I: (C_i $\in$ V) $\wedge$ (C_i $\notin$ DIRECT-PARENTS$_{VC}$) $\wedge$ (C_i $\notin$ DIRECT-CHILDREN$_{VC}$).
Part II: (C_i $\in$ DIRECT-PARENTS$_{VC}$).
Part III: (C_i $\in$ DIRECT-CHILDREN$_{VC}$).
Part IV: (C_i=VC).

Part I: (C_i $\in$ V) $\wedge$ (C_i $\notin$ DIRECT-PARENTS$_{VC}$) $\wedge$ (C_i $\notin$ DIRECT-CHILDREN$_{VC}$).

Part I.a: The introduction of a new class to G can make an existing *direct is-a* relationship between two classes C_i and C_j redundant (indirect) if and only if the new class is placed between C_i and C_j, such as to provide an alternative *is-a* path of length greater or equal to two to the existing *is-a* edge between C_i and C_j. For the addition of VC to make an existing *is-a* edge redundant would imply that (C_i $\in$ DIRECT-PARENTS$_{VC}$) and (C_j $\in$ DIRECT-CHILDREN$_{VC}$), or, vice versa. This is a contradiction to the assumption of **Part I.** Hence, the introduction of VC will not affect any of the existing *is-a* relationships of C_i. It can easily be seen that for ($C_i \notin$ DIRECT-PARENTS$_{VC}$) $\wedge$ ($C_i \notin$ DIRECT-CHILDREN$_{VC}$) no *is-a* edges are added ore removed by the algorithm.

Part I.b: The introduction of a new class creates new *direct is-a* relationships only between classes that are the direct parents or the direct children of the new class. For the addition of VC, this would imply that it creates a new direct *is-a* relationship only for the classes ($C_i \in$ DIRECT-PARENTS$_{VC}$) or ($C_i \in$ DIRECT-CHILDREN$_{VC}$). This is a contradiction to the assumption of **Part I.** Hence, the introduction of VC will not create a new direct *is-a* relationship for C_i. As stated above, it can easily be seen that the algorithm does not add any new *is-a* edges for ($C_i \notin$ DIRECT-PARENTS$_{VC}$) $\wedge$ ($C_i \notin$ DIRECT-CHILDREN$_{VC}$. **q.e.d.**

Part II: ($C_i \in$ DIRECT-PARENTS$_{VC}$).

Part II.a: By Definition 20, C_i acquires a new direct subclass in G', namely, VC. Therefore, the edge e = <VC,C_i> has to be added to E'. It can easily be seen that **step 4.1** of the algorithm adds this edge <VC,C_i> for all $C_i \in$ DIRECT-PARENTS$_{VC}$.

Part II.c: Assume that there is a class C_j in V with which C_i had a direct *is-a* relationship in G. If this class C_j was a superclass of C_i, then it could not have been affected by the introduction of VC. It can easily be seen that no edges are added or removed by the algorithm for classes C_j that are neither in the DIRECT-PARENTS$_{VC}$ nor in the DIRECT-CHILDREN$_{VC}$ set. If this class node C_j was a subclass of C_i, then it can become indirect if VC has been placed between C_i and C_j. The later is only possible for $C_j \in$ DIRECT-CHILDREN$_{VC}$. In this case, the edge e_{ji} = <C_j,C_i> with ($C_j \in$ DIRECT-CHILDREN$_{VC}$) is redundant, since the integration of VC into G adds the edges e2=$< C_j, VC >$ and e3=$< VC, C_i >$ to E. By transitivity, e2=$< C_j, VC >$ and e3=$< VC, C_i >$ imply the (indirect) *is-a* relationship (C_j *is-a* * C_i). Hence, the edge e_{ji} = <C_j,C_i> must be removed from E. See Figure 22 for an example of the creation of redundant edges. It can easily be seen that **step 4.3** of the algorithm removes all edges <C_j,C_i> for $C_j \in$ DIRECT-CHILDREN$_{VC}$ and for $C_i \in$ DIRECT-PARENTS$_{VC}$.

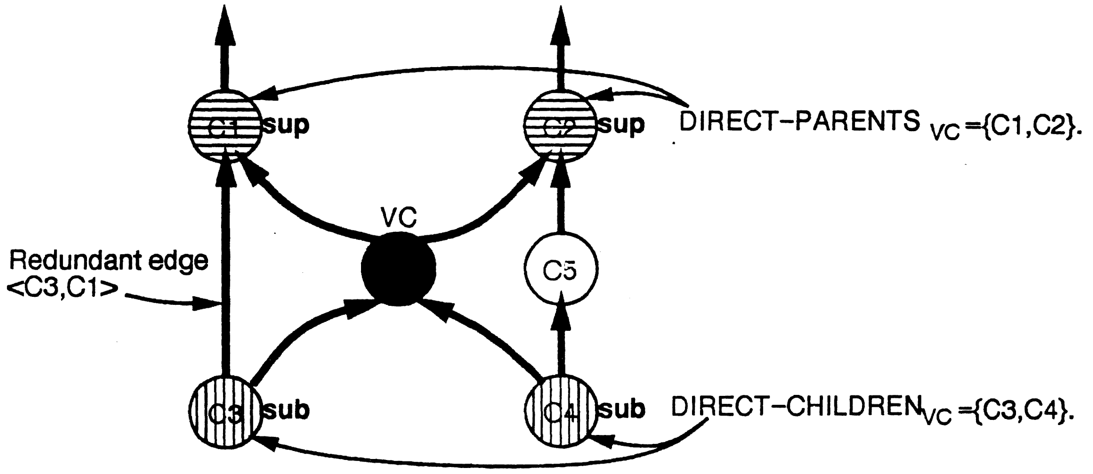


Figure 22: Removal of Redundant Edges During Class Placement.

Part II.d: Assume that there are classes C_j in V with which C_i had no direct *is-a* relationship in G. The introduction of the extra class VC into the schema G will not add any a direct *is-a* relationship between these two classes C_j and C_i, since neither C_i nor C_j are equal to VC. It can easily be seen that no new edges are added for a class $C_i \in$ DIRECT-PARENTS$_{VC}$, except for those connecting it to the new class VC done in **step 4.1.** **q.e.d.**

Part III: ($C_i \in$ DIRECT-CHILDREN$_{VC}$).

The argument for **Part III** is similar to the one for **Part II**, except for dealing with a direct subclass rather than a direct superclass of VC. The proof is thus omitted here. **q.e.d.**

Part IV: (C_i=VC).

As explained above (and in Definition 8), a schema graph G'=(V',E') is **correct** if each class C_i in G' has an *is-a* edge e in E' to all its direct super- and subclasses in G'. Hence, there need to be edges in G' to connect VC to all its direct superclasses, which by Definition 20 are equal to all classes in the DIRECT-PARENTS$_{VC}$ set. It can easily be seen that **step 4.1** of the algorithm adds exactly these required edges <VC,p_j> for all $p_j \in$ DIRECT-PARENTS$_{VC}$. In addition, there need to be edges in G' to connect VC to all its direct subclasses, which by Definition 20 are equal to all classes in the DIRECT-CHILDREN$_{VC}$ set. It can easily be seen that **step 4.2** of the algorithm adds exactly these edges <ch_k,VC> for all $ch_k \in$ DIRECT-CHILDREN$_{VC}$. **q.e.d.**

Lemma 7. (**Correctness**) *Given the schema G = (V,E) and a class VC. All classes C_i in G that are direct children of VC are subclasses of all classes C_k in G that are direct parents of VC.*

Proof: Lemma 7 follows directly from the transitive closure property of the *is-a* relationship. For a class C_i to be a direct subclass of a class VC in G implies, by transitivity, that C_i will also be subclass of all superclasses of VC. More formally,

$$(\forall\ C_i \in \text{V})\ (\forall\ C_k \in \text{V})\ ((C_i \textit{ is-a}\ ^d\ \text{VC}) \wedge (\text{VC} \textit{ is-a}\ ^d\ C_k) \Longrightarrow (C_i \textit{ is-a}\ ^*\ C_k)).$$ **q.e.d.**

From Lemma 7, we can derive the lemma given below.

Lemma 8. *Given the schema G = (V,E) and a class VC. The search for the classes C_i in G that belong to the DIRECT-CHILDREN$_{VC}$ set has to consider only classes that are in subgraphs of G rooted at some class C_k that is a member of the DIRECT-PARENTS$_{VC}$ set of G.*

Lemma 8 follows directly from Lemma 7 since classes in the DIRECT-CHILDREN$_{VC}$ set are subclasses of VC and classes in the DIRECT-PARENTS$_{VC}$ set are superclasses of VC. From Lemma 8, we can conclude that the search for the DIRECT-CHILDREN$_{VC}$ set starts where the search for the DIRECT-PARENTS$_{VC}$ set ends. See the example below for a demonstration of this idea.

Example 17. *In Example 16 we have discussed the application of the Class-Placement algorithm given in Figure 19 to the schema graph shown in Figure 20. We now want to show how Lemma 8 can be used to limit the search space of the algorithm. The algorithm first searches for the DIRECT-PARENTS$_{VC}$ set of G by depth-first downwards traversal of G starting from root C0. Thereafter, it computes the DIRECT-CHILDREN$_{VC}$ set also by a depth-first downwards traversal of G. However, rather than starting again from the root node C0 of G, the search starts from the DIRECT-PARENTS$_{VC}$ set of G (Lemma 8). For instance, the search of direct subclasses continues with the subgraphs rooted at the class nodes C3 and C9. Other subgraphs, e.g., the ones rooted at the nodes C13 and C10, do not need to be explored at all. In Figure 19, we have encircled the parts of the graph G traversed for either the search for the DIRECT-PARENTS$_{VC}$ set or for the search for the DIRECT-CHILDREN$_{VC}$ set by dotted lines. Note that these two search spaces do not overlap.*

7.2 Computing The Direct Parents Set

In this section, we describe an algorithm for the computation of the DIRECT-PARENTS$_{VC}$ set. This then represents a solution for step (1) of the Class-Placement algorithm given in Figure 19. Based on Definition 20, we can make the following observation about the elements in the DIRECT-PARENTS$_{VC}$ set.

Lemma 9. *Given the schema G = (V,E) and a class VC, then the following properties hold for the classes C_i that are direct superclasses of VC in G, i.e., that are in the DIRECT-PARENTS$_{VC}$ set,*

I. All classes C_i in DIRECT-PARENTS$_{VC}$ are subclasses of the root class C0 of the schema:
$(\forall C_i \in$ DIRECT-PARENTS$_{VC})(subsumes(C0, C_i)$=true).

II. For all classes C_i in DIRECT-PARENTS$_{VC}$, none of its subclasses C_k in G subsumes VC:
$(\forall C_k \in V)$ $((C_k$ is-a $^* C_i) \Longrightarrow (subsumes(C_k, VC)$=false)).

III. For all classes C_i in DIRECT-PARENTS$_{VC}$, all of its superclasses C_k in G also subsume VC:
$(\forall C_k \in V)$ $((C_i$ is-a $^* C_k) \Longrightarrow (subsumes(C_k, VC)$=true)).

Proof:
Part I: Schema root.

Part I is true by default, since the root class of a schema is by definition a supertype and a superset of all classes in the schema. **q.e.d.**

Part II: Classes below the direct superclasses of VC.

By Definition 20, for a class C_i to be a direct superclass of VC, i.e., $C_i \in$ DIRECT-PARENTS$_{VC}$, means that the following holds:

(1) (VC *is-a* C_i), and

(2) $(\nexists C_k \in V)(k \neq i)((VC \textit{ is-a}^* C_k) \wedge (C_k \textit{ is-a}^* C_i))$.

For all classes C_k in G that are subclasses of the classes C_i in the DIRECT-PARENTS$_{VC}$ set in G, we have the *is-a* relationship:

$(C_k \textit{ is-a}^* C_i)$.

By Definition 20, this implies that none of the subclasses C_k of C_i can satisfy the subsumption relationship, i.e.,

((VC *is-a* * C_k):=false)

since, otherwise, there would exist classes C_k that contradict the definition of the DIRECT-PARENTS$_{VC}$ set and thus it contradicts the initial assumption that the classes C_i are members of the DIRECT-PARENTS$_{VC}$ set. We can conclude that $\forall C_k \in \textit{subclasses}(C_i)$ the condition *subsumes*(C_k,VC) fails. **q.e.d.**

Part III: Classes above the direct superclasses of VC.

By Definition 20, for all classes C_i in DIRECT-PARENTS$_{VC}$, we have

(VC *is-a* ^{d} C_i).

By the Definition 8 of a schema graph, the classes C_k above the direct superclasses C_i of VC in G have the following *is-a* relationships with their subclasses C_i:

$(C_i \textit{ is-a}^* C_k)$.

By transitivity, this implies the desired *is-a* relationship, namely,

($\forall C_i \in$ DIRECT-PARENTS$_{VC}$) ($\forall C_k \in \textit{superclasses}(C_i)$)

(((VC *is-a* ^{d} C_i) $\wedge$ (C_i *is-a* * C_k)) $\Longrightarrow$ (VC *is-a* * C_k)).

The later is equivalent to the desired subsumption relationship, namely,

subsumes(C_k,VC)=true. **q.e.d.**

Lemma 9 provides us with conclusive information on the shape of the search space for the DIRECT-PARENTS$_{VC}$ set. Namely, by **Part I** of Lemma 9, we know that the root class C0 of G qualifies as superclass for any VC. All other superclasses of VC are also located in the upper half of the schema graph (above all classes that are not superclasses of VC) (**Part II**). Once we find a class C_i which fulfills the condition *subsumes*(C_i,VC) but for which none of its children fulfill the superclass condition of VC, then the class C_i is a direct parent of VC (**Part III**). In term of the schema graph this means that the DIRECT-PARENTS$_{VC}$ set corresponds to all classes that are at the *lower* borderline of classes that still are subsumed by VC. In summary, we can conclude that the search for direct parents of VC should start with the root class C0 of G (**Part I**). It should continue downwards while the encountered classes are still superclasses of VC (**Part II**). Once we find a class C_i for which none of its children is a superclass of VC, then the class C_i is a direct parent of VC (**Part III**) and we are done with the search for this branch.

We associate the label *success* of type Boolean with each class C to delimit this search to the upper part of the schema. The label *success*() determines whether the class C (or any of its subclasses on this branch) has been identified as a member of the DIRECT-PARENTS$_{VC}$ set. This label is used to propagate upwards the fact whether or not a 'successful' class has been located in a given subgraph.

As explained in the previous section, this algorithm is based on the depth-first traversal of the schema graph. However, since we allow for schema graphs with multiple inheritance, we need to assure that a subgraph of G does not get processed more than one. For this, we use a labeling scheme that marks classes C that have been processed by the label *processed*(C)=true. These labeled classes are not processed again.

Algorithm outline: Compute Direct-Parents of A Class.

```
Input:
   A schema G = (V,E) with multiple inheritance with C0 the root.
   A class VC to be integrated into G.
Output:
   DIRECT-PARENTS$_{VC}$: a set of classes.
Data Structures:
   label processed(class) : Boolean;
   label success(class) : Boolean;
   function subsumes(class1,class2) : Boolean;
Algorithm:
   procedure Find-Direct-Parents (G,VC,DIRECT-PARENTS$_{VC}$)
   begin
      DIRECT-PARENTS$_{VC}$ := $\emptyset$;
      for all classes $C_i$ do processed($C_i$) := false; end for
      for all classes $C_i$ do success($C_i$) := false; end for
      Process-Node (root-of-schema);
   end procedure
   procedure Process-Node (C)
   begin
      processed(C) := true;
      for all K in children(C) do begin
         if ((processed(K)=false) and (subsumes(K,VC)))
         then Process-Node (K);
         endif
         if (success(K)) then success(C) := true endif;
      end for
      if (success(C)=false)
      then begin
         DIRECT-PARENTS$_{VC}$ := DIRECT-PARENTS$_{VC}$ $\cup$ C;
         success(C) := true;
      endif
   end procedure
```

Figure 23: The Algorithm For Computing the DIRECT-PARENTS$_{VC}$ Set.

The algorithm for computing the DIRECT-PARENTS$_{VC}$ set of VC, called the Find-Direct-Parents algorithm, is depicted in Figure 23. It proceeds as follows. The algorithm traverses the graph G depth-first starting from schema root. It goes down as far as possible while the function *subsumes(C_i, VC)* holds. If a class K is found that has already been processed, then the algorithm backtracks since the given branch needs to be explored but once. The search will also backtrack if the condition *subsumes*(K,VC) does not hold, since then neither the current class (nor any of its subclasses) will qualify as DIRECT-PARENTS$_{VC}$ (Lemma 9). After all children of a class C have been processed, the algorithm proceeds as follows. If one or more of its children (or their subclasses) have been found to be 'successful', i.e., they were added to the DIRECT-PARENTS$_{VC}$ set, then the class C need no longer be added to the DIRECT-PARENTS$_{VC}$ set. If however none of the children did qualify as DIRECT-PARENTS$_{VC}$, then C is the lowest class on this subtree that still subsumes the virtual class VC. Hence, C is added to the DIRECT-PARENTS$_{VC}$ set.

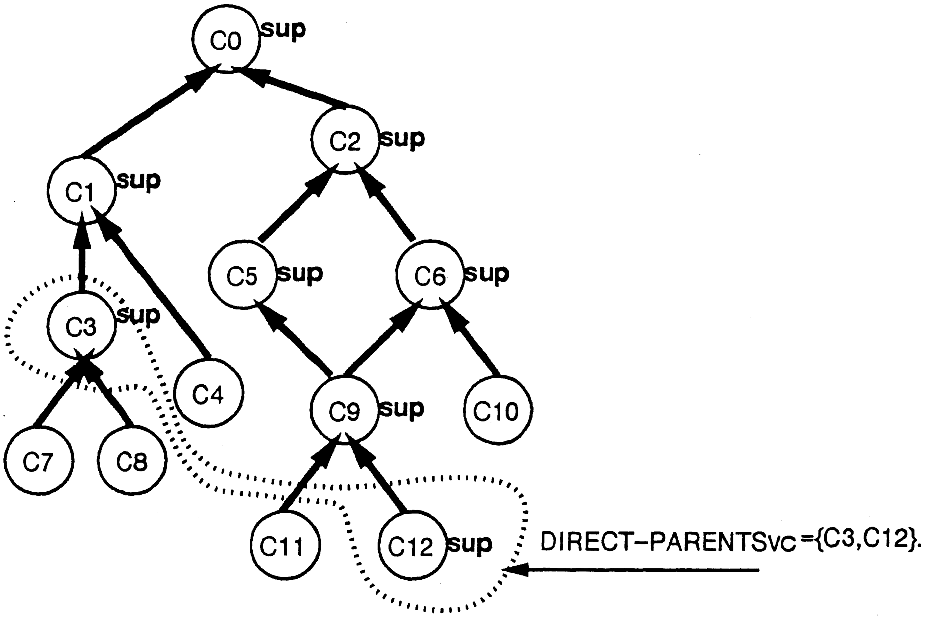


Figure 24: An Example of Applying the Find-Direct-Parents Algorithm.

Example 18. *In Figure 24, we demonstrate the Find-Direct-Parents algorithm given in Figure 23. In the figure, the label* **sup** *for a class C_i indicates that the condition subsumes(C_i, VC) evaluates to true, i.e., C_i is a* **sup***erclass of VC. After initialization, the algorithm starts the traversal of the schema graph G with the root node C0. It processes the first child of C0, which is C1. Since the first if-statement is true, it then recursively processes the first child of C1, which is C3. For both children of C3, which are C7 and C8, the if-statement evaluates to false. Hence, the label success(C3) remains false. The fourth if-statement then adds C3 into the DIRECT-PARENTS$_{VC}$ set. The search now backtracks to complete the processing of C1. Since the child C3 of C1 carries the label success(C3)=true, the class C1 is not added to the DIRECT-PARENTS$_{VC}$ set. The search now backtracks to continue processing of C0, which corresponds to exploring the right subgraph in the just described manner.*

Theorem 6. *(***Correctness***) Given the schema G = (V,E) and a class VC. The Find-Direct-Parents algorithm in Figure 23 will find all classes of G, called DIRECT-PARENTS$_{VC}$, that are direct superclasses of VC.*

Proof: We prove Theorem 6 in several steps.

Part I: The Find-Direct-Parents algorithm in Figure 23 will call the Process-Node function on each superclass of VC exactly once.

A schema G = (V,E) is a *connected* directed acyclic graph. This assures that a general depth-first search starting from the root will meet all leaf nodes exactly once (See a standard algorithm book, e.g., [2], pg. 176 - 178). By Lemma 9, the shape of the search space for superclasses of VC in G is such that all indirect superclasses of VC are above all direct parents of VC which are above all other classes in the schema graph G. The Find-Direct-Parents algorithm traverses G in a depth-first manner starting from the root of G until it reaches class C_i for which the condition subsumes(C_i,VC) no longer holds, i.e., it will process all indirect and all direct parents of C_i and it backtracks as soon as it reaches other undesirable classes. The Find-Direct-Parents algorithm in Figure 23 recursively continues the search downwards if and only if the first if-statement evaluates to true, i.e., the condition "((processed(K)=false) and (subsumes(K,VC)))" is true. This means that the search continues only if the class K has not been processed before and if it is indeed a superclass of VC. This assures that the search stops after all superclasses of VC have been found and also that the same class is processed at most once.

Part II: When traversing downwards the graph, the algorithm encounters a class C with processed(C)=true, then success(C)=true must also hold.

First, the label *processed*(C) is set to true for classes C that have been processed before. Second, once a class C_i is found to be a direct parent of VC, then the label *success*(C_i) is set to true by the third if-statement. In addition, the second if-statement sets the *success* label of parents to true whenever the *success* label of a child is true. Hence, the *success* label of all parents of the class C is also changed to true when backtracking. The directed depth-first search will completely finish the processing of a class C (and all its successors) before possibly encountering the same class again via an alternative path (due to multiple inheritance). This can easily be shown by using the facts that (1) there are no directed cycles in the schema graph and (2) the search starting at a class C will recursively process only its subclasses in the schema graph but no other class until terminating the processing of class C. Thus, the search encounters a class with the label *processed*(C)=true if and only if its label *success*(C)=true either due to being a direct parent or due to having backtracked passed this class.

Part III: The algorithm adds the class C_i of V into the DIRECT-PARENTS$_{VC}$ set if and only if the class C_i is a direct superclass of VC.

Part III.a: If a class C_i of V is a direct superclass of VC then the algorithm will add C_i into the DIRECT-PARENTS$_{VC}$ set.

Assume that the class C_i of V is a direct superclass of VC. Note that a class C_i is added into the DIRECT-PARENTS$_{VC}$ set only by the last if-statement of the algorithm. For this if-statement to be executed for a class C_i, we must show that the label *success*(C_i) must be false.

First, if the class C_i has no children, then the for-loop over all children K of C_i will not be executed. Hence, the label *success*(C_i), which is modified within this for-loop, is not changed and remains 'false'. If *success*(C_i)=false, then the last if-statement evaluates to true and adds C_i to the DIRECT-PARENTS$_{VC}$ set.

Second, if the class C_i has some children K, then the for-loop over all children K of C_i will be executed. However, none of the children K of C_i will be a superclass of VC (by Lemma 9) and hence the function *subsumes*(K,VC) will fail for all of them. Therefore, the first if-statement within the for-loop will be skipped for all K. Hence, the label *success*(K)=false will not be modified, and the second if-statement within the for-loop will also be skipped for all K. This again implies that the label *success*(C_i) is not modified and remains 'false'. In this case, the last if-statement will add C_i to the DIRECT-PARENTS$_{VC}$ set.

In summary, we have demonstrated that a class C_i that is a direct superclass of VC will always be added to DIRECT-PARENTS$_{VC}$ set by the algorithm.

Part III.b: If the algorithm adds the class C_i of V into the DIRECT-PARENTS$_{VC}$ set then the class C_i is a direct superclass of VC.

Assume that the algorithm has added the class C_i of V to the DIRECT-PARENTS$_{VC}$ set. Note that there is only one statement, namely, the last if-statement of the algorithm, that would add a class C_i into the DIRECT-PARENTS$_{VC}$ set. For this if-statement to be executed, the label *success*(C_i) must have been false. For the label *success*(C_i) to stay false means that the following circumstances must be true for C_i.

First, if the class C_i has no children, then the for-loop over all children K of C_i will not be executed and the label *success*(C_i) will remain false. Recall that the function Process-Node() is only called for class C_i, for which the function *subsumes*(C_i,VC) is true. And since the function Process-Node() has been called for the class C_i, we can deduce that the class C_i must be a superclass of VC. C_i being a superclass of VC and not having any children implies that C_i is by default a direct parent of VC.

Second, if the class C_i has children, then the for-loop over all children K of C_i is executed. However, if the label *success*(K) were changed for any of the children K of C_i to true, then it would also be changed for C_i. Consequently, the label *success*(K) must have remained 'false' for all children K of C_i. This can only occur, if the first if-statement evaluates to false for all children K of C_i (Part I). Since if the first part of the if-statement condition, which is processed(K)=false, is false, then K has been processed before and the algorithm has already backtracked over K (since the schema G has no loops). By Part I, this implies that success(K):=true. This would be a contradiction to the fact that the label *success*(K) must have remained 'false' for all children K of C_i. Hence, the first part of the if-statement condition must evaluate to true, and by this we can conclude that the second part of the if-statement condition, which is *subsumes*(K,VC), must evaluate to false for all K. By Lemma 9, *subsumes*(C_i,VC):=true and ($\forall K \in children(C_i)$) (*subsumes*(K,VC):=false) implies that C_i is a direct parent of VC. **q.e.d.**

Theorem 7. **(Complexity)** *Given the schema $G = (V,E)$ and a class VC. The Find-Direct-Parents algorithm in Figure 23 has a worst case complexity of $O(|E|)$ with $|E|$ the number of edges in the schema.*

Proof: The proof is similar to those for the standard depth-first search (See a standard algorithm book, e.g., [2], pg. 176 - 178). The initialize functions require O(|V|) steps if a list of class nodes is available. The time spend in the Process-Node (C) function is proportional to the number of out-going edges (or the number of children of C). The function Process-Node (C) is called at most once for a given class C, since the class are marked by "processed" the first time they are called. Therefore the worst case complexity is O(max(|V|,|E|)). Note that in a connected directed graph G the number of edges |E| in G is always larger or equal to the number of classes |V| in G. $|E| \geq |V|$ implies that complexity(Find-Direct-Children1) := O(max(|V|,|E|)) := O(|E|). **q.e.d.**

7.3 Computing The Direct Children Set

In this section, we present algorithms for computing the set of direct children of a new class VC in a schema graph G. An algorithm similar to the one for computing the direct parents set (Figure 23) can be used. The algorithm traverses the schema graph downwards in a depth-first manner, until it finds a class C_i that fulfills the condition *subsumes*(VC,C_i). This class C_i is a *direct* subclass of VC, since any class above it that is also subsumed by VC would have been found before reaching C_i. See Figure 25 for a precise formulation of the just outlined algorithm.

Algorithm outline: Compute The Direct-Children classes of A Class.

```
Input:
   A schema G=(V,E) with single inheritance and a class VC to be integrated into G.
   DIRECT-PARENTS$_{VC}$: a set of direct-parent classes of VC in G.
Output:
   DIRECT-CHILDREN$_{VC}$: a set of direct-children classes of VC in G.
Data Structures:
   function subsumes(class1,class2) : Boolean;
Algorithm:
   procedure Find-Direct-Children1(G,VC,DIRECT-PARENTS$_{VC}$) return DIRECT-CHILDREN$_{VC}$
   begin
      DIRECT-CHILDREN$_{VC}$ := $\emptyset$;
      for all classes $C_i$ in DIRECT-PARENTS$_{VC}$ do Process-Node ($C_i$); end for
   end procedure
   procedure Process-Node (class C)
   begin
      if (subsumes(VC,C))
      then DIRECT-CHILDREN$_{VC}$ := DIRECT-CHILDREN$_{VC}$ $\cup$ C;
      else for all classes K in children(C) do Process-Node (K); end for
      endif
   end procedure
```

Figure 25: The Algorithm For Computing the DIRECT-CHILDREN$_{VC}$ Set for Single Inheritance.

The Find-Direct-Children1 algorithm in Figure 25 works on a schema G = (V,E) with single inheritance. Such a schema G with single inheritance corresponds to a graph structure in the form of a tree, and hence a depth-first traversal of G will not traverse the same subgraph twice. Hence, the *processed* label introduced earlier to mark already processed classes is not needed for graphs G with single inheritance.

Example 19. *In this example, we apply the Find-Direct-Children1 algorithm given in Figure 25 to find the direct subclasses of the class VC in the schema graph G shown in Figure 26. We assume that the graph shown in Figure 26 corresponds to a subgraph of the complete schema graph G and that the DIRECT-PARENTS$_{VC}$ set has been determined to be { C1, C2 }. The search starts by traversing G downwards starting from the classes { C1, C2 }. The algorithm*

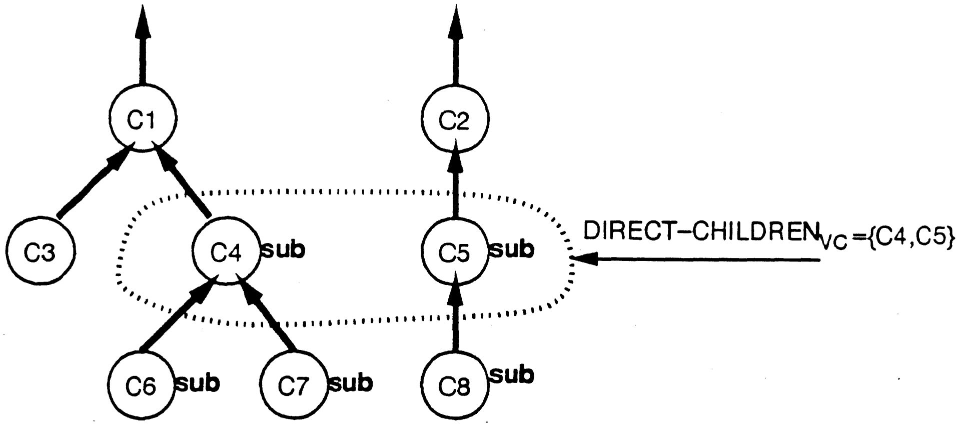


Figure 26: An Example of Applying the Find-Direct-Children1 Algorithm.

first processes the class C1 using the Process-Node() function. Since the first if-statement fails, the execution falls into the else-branch. Next, the for-loop iterates over the children of C1, which are C3 and C4. For the first iteration of the for-loop with K=C3, the if-statement fails and the execution continues with the else-branch. However, since C3 has no children, the for-loop in the else-branch is not executed. The algorithm then processes the second child of C1, which is K=C4. The first if-statement evaluates to true, i.e., C4 is indeed a subclass of VC, and the then-branch adds C4 to the DIRECT-CHILDREN$_{VC}$ set. The else-branch is skipped, and hence none of the classes below C4 will be processed. The search along this branch is completed. The algorithm backtracks to process the next class in the DIRECT-PARENTS$_{VC}$ set, C2, in the just described manner.

Theorem 8. *(***Correctness***) Given the schema G = (V,E) with single inheritance and a class VC, then the Find-Direct-Children1 algorithm in Figure 25 will find all classes of G, called DIRECT-CHILDREN$_{VC}$, that are direct subclasses of VC*[4].

Proof: A schema G = (V,E) is a *connected* directed acyclic graph. This assures that a general depth-first search starting from the root will meet all leaf nodes exactly once (See a standard algorithm book, e.g., [2], pg. 176 - 178). By Lemma 8, it is sufficient to start the the search for the DIRECT-CHILDREN$_{VC}$ set from the classes in the DIRECT-PARENTS$_{VC}$ set on downwards rather than from the root of the schema graph. By Definition 20, the DIRECT-CHILDREN$_{VC}$ set contains all classes that fulfill the following conditions: (1) they are subclasses of VC and (2) there are no other classes above them in the schema graph that are also subclasses of VC. The later means that they are the **highest** possible classes in the schema graph that are still subsumed by VC. Due to the depth-first traversal of the schema graph G, the Find-Direct-Children1 algorithm finds the 'highest' classes of G that are subsumed by VC first. In fact, the search terminates the search for a given branch of the tree G, whenever the condition "*subsumes*(VC,C)" evaluates to true. By Definition 20, these highest subclasses C of VC are indeed the direct children of VC. These classes are thus added to the DIRECT-CHILDREN$_{VC}$ set by

[4] We assume that the function *subsumes* is computable for our object model and by the terms 'all direct subclasses' we mean all direct subclasses recognizable by the *subsumes* function.

the then-branch of the if-statement, and the search along this branch of the schema graph is terminated. **q.e.d.**

Theorem 9. (**Complexity**) *Given a schema G = (V,E) with single inheritance and a class VC, then the Find-Direct-Children1 algorithm in Figure 25 has a worst case complexity of O(|E|) with |E| the number of edges in the schema.*

Proof: We assume a set representation (e.g., for the sets *DIRECT-CHILDREN*$_{VC}$ and *DIRECT-PARENTS*$_{VC}$) that allows for the manipulation of the set by adding or deleting elements and for the checking its membership for a particular element in O(1) time, e.g., a vector representation of G. The proof is then similar to those for the standard depth-first search (See a standard algorithm book, e.g., [2], pg. 176 - 178). The functions to initialize the search require O(|V|) steps if a list of class nodes is available. The time spend in the Process-Node (C) function for a given class C is proportional to the number of out-going edges (or the number of children of C). The function Process-Node (C) is called at most once for a given class C, since for a graph G with single inheritance, a depth-first traversal will by default encounter each node in the tree exactly once (See [2], pg. 176 - 178). Therefore the worst case complexity is O(max(|V|,|E|)). Note that in a connected directed graph the number of edges |E| in the schema is always larger or equal to the number of classes |V| in the schema. $|E| \geq |V|$ implies that complexity(Find-Direct-Children1) := O(max(|V|,|E|)) := O(|E|). **q.e.d.**

The Find-Direct-Children1 algorithm in Figure 25 works on a schema G = (V,E) with single inheritance. If we want to apply the algorithm to a schema G with multiple inheritance, which corresponds to a DAG structure rather than to a tree structure, then we need to assure that the depth-first traversal will not traverse the same subgraph of G more than once. Hence, we introduce the label *processed* into the Find-Direct-Children1 algorithm in Figure 25 to mark classes that have been processed before. The resulting algorithm is shown in Figure 27.

Theorem 10. (**Correctness**) *Given the schema G = (V,E) with possibly multiple inheritance and a class VC, then the Find-Direct-Children2 algorithm in Figure 27 will find* **all** *classes of G, called DIRECT-CHILDREN*$_{VC}$*, that are direct subclasses of VC. In addition, the algorithm might select some classes* C_i *that are indirect subclasses of VC*[5].

Proof:
Part I: The Find-Direct-Children2 algorithm finds **all** *DIRECT-CHILDREN*$_{VC}$ of VC.

Theorem 8 shows that the Find-Direct-Children1 algorithm finds **all** *DIRECT-CHILDREN*$_{VC}$ of VC for a graph G with single-inheritance. The same reasoning can be used here to argue that the Find-Direct-Children2 algorithm finds all direct subclasses of VC for a graph G with multiple-inheritance. Namely, by depth-first traversal, each relevant branch of G is explored (once) and all classes C_i for which the condition "*subsumes*(VC,C)" evaluates to true are collected in the set *DIRECT-CHILDREN*$_{VC}$.

Part II: The Find-Direct-Children2 algorithm may also find indirect subclasses of VC.

[5] Again, we assume that the function *subsumes* is computable.

Algorithm outline: Compute The Direct-Children classes of A Class.

Input:

A schema G=(V,E) with multiple inheritance and a class VC to be integrated into G.
DIRECT-PARENTS$_{VC}$: a set of direct-parent classes of VC in G.

Output:

DIRECT-CHILDREN$_{VC}$: direct- and possibly indirect-children of VC in G.

Data Structures:

label processed(class) : Boolean;
function *subsumes*(class1,class2) : Boolean;

Algorithm:

```
procedure Find-Direct-Children2(G,VC,DIRECT-PARENTS$_{VC}$) return DIRECT-CHILDREN$_{VC}$
begin
    DIRECT-CHILDREN$_{VC}$1 := $\emptyset$;
    for all classes $C_i$ in V do processed($C_i$) := false; end for
    for all classes $C_i$ in DIRECT-PARENTS$_{VC}$ do
        processed($C_i$) := true;
        Process-Node ($C_i$);
    end for
end procedure
procedure Process-Node (class C)
begin
    if (subsumes(VC,C))
    then DIRECT-CHILDREN$_{VC}$ := DIRECT-CHILDREN$_{VC}$ $\cup$ C;
    else begin
        for all classes K in children(C) do begin
            if (processed(K)=false)
            then begin
                processed(K):=true;
                Process-Node (K);
            endif
        end for
    endif
end procedure
```

Figure 27: An Algorithm for Computing the DIRECT-CHILDREN$_{VC}$ Set for Multiple Inheritance.

In a graph G with multiple inheritance, a class node C may have more than one parent. Hence, we may arrive at the same node C of G more than once during a depth-first traversal. If we arrive at such a node C_i from one parent while one of its other parents is also subsumed by VC, then the class C_i will be redundant in the *DIRECT-CHILDREN*$_{VC}$ set. An example of this situation is depicted in Figure 28. **q.e.d.**

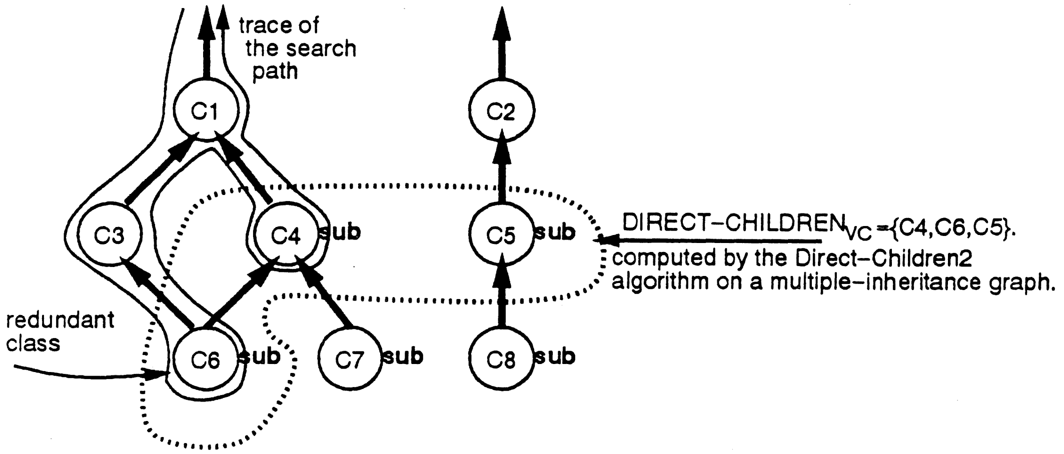

trace of
the search
path
C1
C2
C3
C4
sub
C5
sub
DIRECT-CHILDREN$_{VC}$={C4,C6,C5}.
computed by the Direct-Children2
algorithm on a multiple-inheritance graph.
redundant
class
C6
sub
C7
sub
C8
sub

Figure 28: Applying the Find-Direct-Children2 Algorithm to Multiple-Inheritance Schema.

Example 20. *The schema graph in Figure 28 is identical to the one in Figure 26, except for the addition of the arc e = <C6,C3>, which turns the schema into a graph with multiple inheritance. The Find-Direct-Children2 algorithm given in Figure 27 proceeds as described in Example 19. However, when processing the class C3, the algorithm now also processes C6, the new child of C3. Since C6 is the first class on this branch that fulfills the subsumes(VC,C6) condition, C6 is added to the DIRECT-CHILDREN*$_{VC}$ *set. Next, C4 and C5 are also added to the DIRECT-CHILDREN*$_{VC}$ *set as explained in Example 19, It can easily be seen that C6 is not a direct child of VC, since there is another class, C4, in the graph that is a direct child of VC and that is a superclass of C6. Hence, C6 is redundant in the DIRECT-CHILDREN*$_{VC}$ *set.*

Theorem 11. *(***Complexity***) Given a schema G = (V,E) with multiple inheritance and a class VC, then the Find-Direct-Children2 algorithm in Figure 27 has a worst case complexity of O(|E|) with |E| the number of edges in the schema.*

Proof: The Find-Direct-Children2 algorithm in Figure 27 is identical to the Find-Direct-Children1 algorithm in Figure 25, except for the introduction of the *processed* label to delimit the search on a graph with multiple inheritance. Hence, the proof proceeds similar to Theorem 9. The only difference is the reason for why the function Process-Node (C) is called at most once for a given class C given below. Namely, the Find-Direct-Children2 algorithm marks all classes by the label *processed* the first time they are called. When a class C is encountered for a second time, then its label *processed*(C) will be true. For a class C with the label *processed*(C)=true, the if-statement evaluates to false and the Process-Node (C) function call is skipped. Hence, the Process-Node (C) function is called but once for each class C. **q.e.d.**

Lemma 10. *Given the schema $G = (V,E)$ and a class VC, then the following properties hold for classes C_i that are either direct or indirect subclasses of VC in G:*

I. For all C_i that are direct subclasses of VC, i.e., they are true members of the DIRECT-CHILDREN$_{VC}$ set, the following holds:
$(\forall C_k \in parents(C_i))(subsumes(VC,C_k)=false)$.

II. For all C_i that are indirect subclasses of VC, i.e., they are redundant members of the DIRECT-CHILDREN$_{VC}$ set, the following holds:
$(\exists C_k \in parents(C_i))(subsumes(VC,C_k)=true)$.

Proof:
Part I: For direct subclasses of VC.

Assume that the class C_i is a direct subclass of VC. Then the following must hold by Definition 20:

(1) (C_i *is-a* VC), and

(2) ($\nexists C_k \in V$) with (C_i *is-a* * C_k) $\wedge$ (C_k *is-a* * VC).

The second condition can be rewritten as "$(\nexists C_k \in parents(C_i))$ with (C_k *is-a* * VC)", since if there is a class C_k with the above property then there is at least one direct superclass (parent) of C_i with the same property. This, of course, is equivalent to the condition that "$(\forall C_k \in parents(C_i))$ (*subsumes*(VC,C_k):=false)" given in Lemma 10.I.

Part II: For indirect subclasses of VC.

For a class C_i to be an indirect – but not a direct – subclass of VC means that the following holds:

(1) (C_i *is-a* VC), and

(2) ($\exists C_k \in V$) with (C_i *is-a* * C_k) $\wedge$ (C_k *is-a* * VC).

The first condition indicates that C_i is a subclass of VC and the second condition is a negation of the requirement that the subclass relationship must be *direct.* The second condition can be rewritten as "$(\exists C_k \in parents(C_i))$($C_k$ *is-a* * VC)", since if there is a class C_k with the above property then there is at least one direct superclass (parent) of C_i with the same property. This is equivalent to the condition listed in Lemma 10.II. **q.e.d.**

Next, we demonstrate these definitions by an example.

Example 21. As discussed in Example 20, the Find-Direct-Children2 algorithm finds the DIRECT-CHILDREN$_{VC}$ set := { C4, C6, C5 } for the schema graph in Figure 28. By Lemma 10.I, the class C4 is a true member of the DIRECT-CHILDREN$_{VC}$ set, since its only parent C1 is not subsumed by VC. Similarly, the class C5 is a true member of the DIRECT-CHILDREN$_{VC}$ set, since its parent C2 is not subsumed by VC. However, by Lemma 10.II, the class C6 is not a direct subclass of VC, since one of its parents, namely, C4, is also subsumed by VC. Since there is at least one class between VC and the class C6, namely, its parent C4, the class C6 is a redundant member of the DIRECT-CHILDREN$_{VC}$ set (Lemma 10.II).

Based on Lemma 10, we can derive an algorithm that removes all redundant classes (indirect subclasses of VC) from the *DIRECT-CHILDREN$_{VC}$* set generated by the Find-Direct-Children2 algorithm. Namely, for each class C_i in the *DIRECT-CHILDREN$_{VC}$* set, we test the condition outlined in Lemma 10. If the condition in Lemma 10.a is true, then the class C_i is indeed a *direct child* of VC and should remain in the *DIRECT-CHILDREN$_{VC}$* set. If the condition in Lemma 10.b is true, then the class C_i is not a *direct child* of VC and should be removed from the *DIRECT-CHILDREN$_{VC}$* set. This algorithm has been implemented by the Remove-Redundant-Classes procedure shown in Figure 29.

Algorithm outline: Remove all indirect subclasses of VC from DIRECT-CHILDREN$_{VC}$.

Input:

A schema G=(V,E) with multiple inheritance and a class VC to be integrated into G.
The set DIRECT-CHILDREN$_{VC}$ with all direct and possibly some indirect subclasses of VC.

Output:

DIRECT-CHILDREN$_{VC}$ with all direct children of VC.

Data Structures:

```
variable redundant: Boolean;
function subsumes(class1,class2) : Boolean;
function remove-from-set(class,set-of-classes);
```

Algorithm:

```
procedure Remove-Redundant-Classes(G,VC,DIRECT-CHILDREN$_{VC}$)
return DIRECT-CHILDREN$_{VC}$
begin
    for all $C_i$ in DIRECT-CHILDREN$_{VC}$ do begin
        redundant := false;
        for all K in parents($C_i$) do
            if (subsumes(VC,K)) then redundant:=true; endif
        end for
        if (redundant=true) then remove-from-set($C_i$,DIRECT-CHILDREN$_{VC}$); endif
    end for
end procedure
```

Figure 29: An Algorithm For Removing all Indirect Subclasses of VC from DIRECT-CHILDREN$_{VC}$.

Example 22. *The schema graph in Figure 30 is identical to the one in Figure 28. Example 20 explained how all direct and possibly some indirect subclasses of VC are found, i.e., DIRECT-CHILDREN$_{VC}$:= { C4, C6, C5 }, while in this example we show how all indirect classes are removed from this set. The algorithm proceeds as follows. The first iteration of the first for-loop with C_i=C4 checks whether the redundancy condition given in Lemma 10.I applies for C4. The second for-loop iterates over all parents of C4 to check whether any of them are subsumed by VC. Since C1, the only parent of C4, is not, the variable redundant:=false is not modified. C4 is found to be not redundant. The second iteration of the first for-loop with C_i=C6 checks whether C6 is redundant. The second for-loop iterates over all parents of C6, which are C3 and C4. For K=C3, the first if-statement fails. However, for K=C4, the first if-statement succeeds and the variable redundant is set to true. Consequently, the second if-statement evaluates to true and removes C6 from the DIRECT-CHILDREN$_{VC}$ set. C6 has been found to be redundant. Lastly, the third iteration of the for-loop with C_i=C5 determines that C5 is not redundant. In summary, the classes C4 and C5 remain in the DIRECT-CHILDREN$_{VC}$ set.*

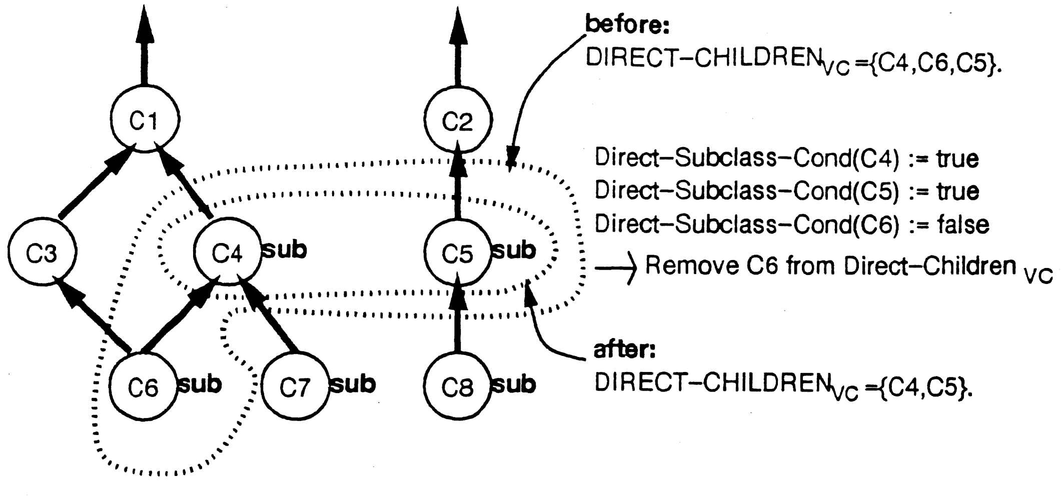

C1
C2
C3
C4
C5
C6
C7
C8
sub
sub
sub
sub
sub
before:
DIRECT-CHILDREN$_{VC}$={C4,C6,C5}.
Direct-Subclass-Cond(C4) := true
Direct-Subclass-Cond(C5) := true
Direct-Subclass-Cond(C6) := false
⟶ Remove C6 from Direct-Children $_{VC}$
after:
DIRECT-CHILDREN$_{VC}$={C4,C5}.

Figure 30: An Example of Applying the Remove-Redundant-Classes Algorithm.

Below, we show that the Remove-Redundant-Classes procedure in Figure 29 implements the algorithm for removing all redundant members (indirect subclasses) and none of the true members (direct subclasses) of the DIRECT-CHILDREN$_{VC}$ set as described in the previous paragraph.

Theorem 12. *(***Correctness***) Given the schema $G = (V,E)$ with possibly multiple inheritance, a class VC, and the set DIRECT-CHILDREN$_{VC}$ set composed of all direct subclasses of VC and possibly some indirect subclasses of VC. Then the Remove-Redundant-Classes algorithm in Figure 29 removes all of the indirect and none of the direct subclasses of VC from the set DIRECT-CHILDREN$_{VC}$.*

Proof: For a class C_i in the set DIRECT-CHILDREN$_{VC}$, the Remove-Redundant-Classes algorithm checks for the condition described in Lemma 10, namely, the condition $(\forall C_k \in$ *parents*$(C_i))($*subsumes*(VC,C_k)) for C_i a subclass of VC. The flag *redundant* remains false if and only if the condition of the if-statement never evaluates to true. This is equivalent to $(\forall C_k \in$ *parents*$(C_i))($*subsumes*(VC,C_k)=false), which is equal to the first condition listed in Lemma 10.I. Hence, by Lemma 10.I, the class C_i is a direct subclass of VC and is thus not removed from the DIRECT-CHILDREN$_{VC}$ set by the second if-statement. The flag *redundant* is set to true if and only if the condition of the if-statement evaluates to true at least once. This is equivalent to the second condition listed in Lemma 10.II, namely, there exists a class C_k in parents(C_i) for which (*subsumes*(VC,C_k)=true). By Lemma 10.II, the class C_i is an indirect subclass of VC and thus has to be removed from the DIRECT-CHILDREN$_{VC}$ set. It can easily be seen that this is done by the second if-statement. After running this test for all classes in the DIRECT-CHILDREN$_{VC}$ set, all indirect subclasses of VC have been removed and thus DIRECT-CHILDREN$_{VC}$ is left with the direct subclasses of VC. **q.e.d.**

Theorem 13. *(***Correctness***) Given the schema $G = (V,E)$ with possibly multiple inheritance, a class VC, and a set DIRECT-PARENTS$_{VC}$ consisting of all direct parents of VC in G. Then the algorithm Find-Direct-Children in Figure 31 finds all classes of G, called DIRECT-CHILDREN$_{VC}$, that are direct subclasses of VC.*

```
Algorithm outline: Find Direct-Children$_{VC}$ For a Graph G with Multiple Inheritance.
Input:
    A schema G = (V,E) with multiple inheritance and a class VC to be integrated into G.
    DIRECT-PARENTS$_{VC}$: set of all direct parents of VC.
Output:
    DIRECT-CHILDREN$_{VC}$: set of all direct children of VC.
Algorithm:
    procedure          Find-Direct-Children(G,VC,DIRECT-PARENTS$_{VC}$)          return
DIRECT-CHILDREN$_{VC}$
    begin
        DIRECT-CHILDREN$_{VC}$ := Find-Direct-Children2(G,VC,DIRECT-PARENTS$_{VC}$);
        DIRECT-CHILDREN$_{VC}$ := Remove-Redundant-Classes(G,VC,DIRECT-CHILDREN$_{VC}$);
    end procedure
```

Figure 31: An Algorithm for Finding All Direct Children for Multiple Inheritance.

Proof: By Theorem 10, we know that the Find-Direct-Children2 algorithm will find **all** classes of G that are direct subclasses of VC plus possibly some classes C_i that are indirect subclasses of VC in G. By Theorem 12, we know that the Remove-Redundant-Classes algorithm removes all indirect subclasses of VC and none of the direct subclasses of VC from DIRECT-CHILDREN$_{VC}$. Hence, all direct subclasses of VC will remain in the DIRECT-CHILDREN$_{VC}$ set. **q.e.d.**

Theorem 14. (**Complexity**) *Given the schema G = (V,E) and a class VC, then the Find-Direct-Children algorithm in Figure 31 has a worst case complexity of O(|E|) with |E| the number of edges in the schema.*

Proof: By Theorem 9, the complexity of the first procedure, called Find-Direct-Children2, is O(max(|V|,|E|)) = O(|E|). The complexity of the second procedure, called Remove-Redundant-Classes, is $O(\sum_{C_i \in DIRECT-CHILDREN_{VC}}(\#parents(C_i)))$ with #parents(C_i) equal to the number of outgoing *is-a* edges of C_i in G, denoted by #outgoing-edges(C_i). The sum $\sum_{C_i \in DIRECT-CHILDREN_{VC}}$(#outgoing-edges($C_i$)) is of course smaller or equal to |E|. In summary, the complexity of the Find-Direct-Children algorithm is complexity(Find-Direct-Children) := complexity(Find-Direct-Children2) + complexity(Remove-Redundant-Classes) = O(|E| + |E|) = O(|E|). **q.e.d.**

7.4 Summary

Figure 19 in Section 7.1 presents the general algorithm for integrating a virtual class VC into a global schema G. In Sections 7.2 and 7.3, we present algorithms for subtasks (1) and (3) of the Class-Placement algorithm, respectively. The reader is referred to Example 16 for an example of the complete algorithm. The complexity of the class placement algorithm given in Figure 19 is linear, since both the computation of the direct parent and the direct children set can be done in linear time as shown in Sections 7.2 and 7.3, respectively.

8 THE COMPLETE CLASS INTEGRATION ALGORITHM

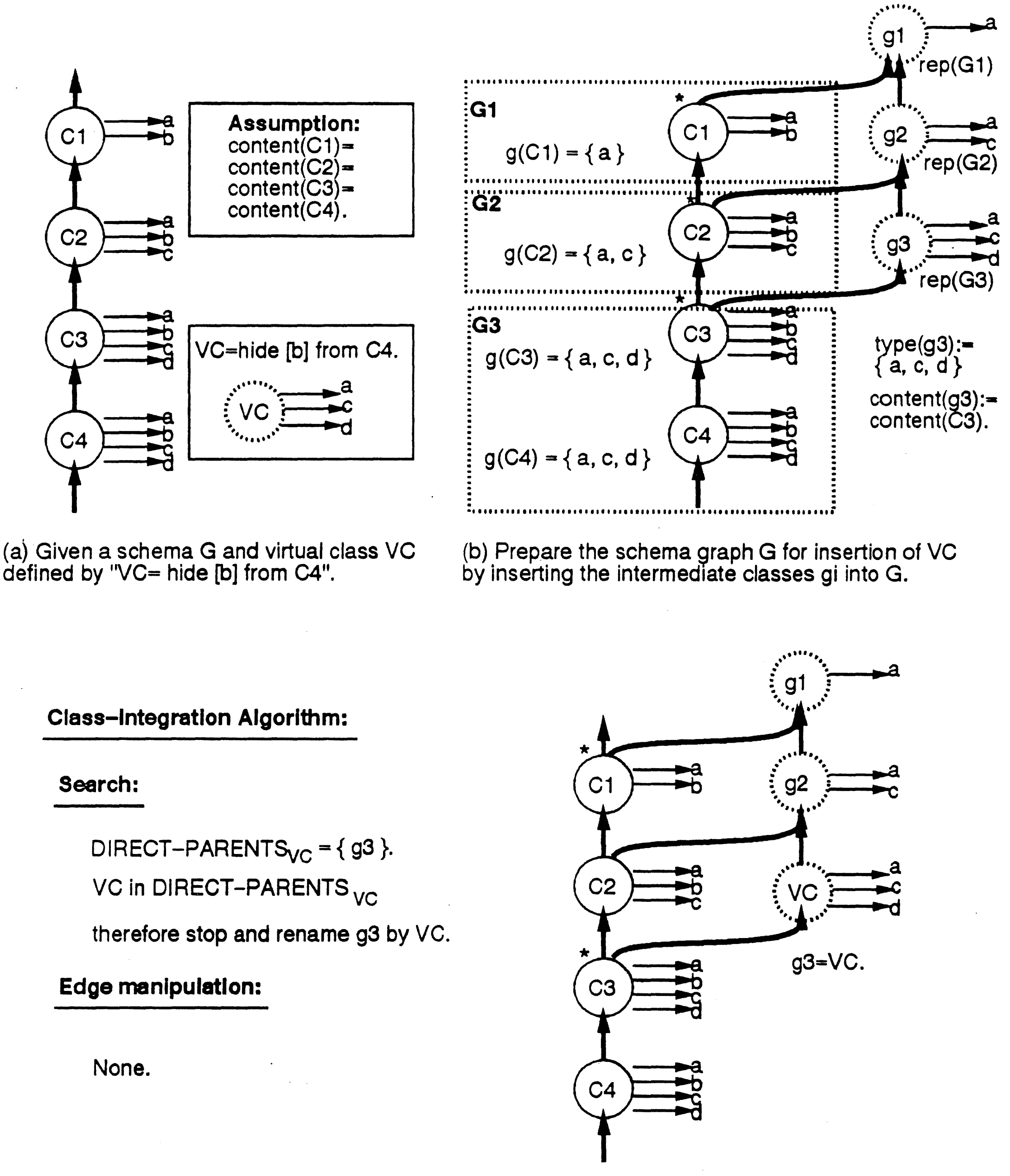
In this section, we present the general solution for classification based on the algorithms developed in previous sections (see also Figure 17). In Section 5, we have demonstrated that the introduction of a class with a new type into a schema graph may require the creation of yet additional classes. The Generate-Intermediate-Classes(G,VC) procedure addresses this problem by preparing the type hierarchy for insertion of a virtual class by adding all intermediate classes required to preserve the correct type lattice underlying the schema graph. Thereafter, we need to determine the correct location for VC in this prepared schema graph. The Class-Placement(G,VC) procedure presented in Section 7 solves exactly this problem. Finally, putting these two algorithms together as done in Figure 17 results in a class integration algorithm that solves the general classification problem (Theorem 6).

```
Algorithm outline: Integration of a Virtual Class VC into the Global Schema G.
Input:
  A schema G = (V,E) and a class VC.
Output:
  VC integrated into G (with possibly additional intermediate classes).
Algorithm:
  procedure Integration(G,VC)
  begin
    // Prepare G by adding all intermediate classes required for its closure with respect to VC.
    Generate-Intermediate-Classes(G,VC);
    // Insert VC into this modified schema graph G.
    Class-Placement(G,VC);
  end procedure
```

Figure 32: The Complete Class Integration Algorithm.

Below, we present two examples that demonstrate the classification algorithm given in Figure 32.

Example 23. *Figure 33 demonstrates how the classification algorithm given in Figure 32 inserts a class VC into a class hierarchy G. Figure 33.a shows G before type classification. For this example, we assume that all four classes have the same object instances as members, i.e., content(C1) = content(C2) = content(C3) = content(C4). The virtual class VC is derived by the view derivation "VC = hide [b] from C_4". Then the type of VC is defined by type(VC):=[a, c, d] and the object membership of VC is defined by content(VC) := content(C4). We first run the Generate-Intermediate-Classes(G, VC) procedure to modify the class hierarchy G such as to prepare its underlying type hierarchy for the insertion of the new class VC. As explained in Example 23 and shown in Figure 33.b, this procedure generated the three intermediate classes g_1, g_2, and g_3. Next, we apply the Class-Placement(G, VC) procedure to add VC to the now prepared schema graph G (Figure 33.c). By traversing G downwards starting from the root, the algorithm establishes the set DIRECT-PARENTS$_{VC}$:= \{g_3\}. g_3 is a direct parent of VC since (1) g_3 subsumes VC with type(g_3) = [a, c, d] = type(VC) and content(g_3) = content(C3) = content(C4) = content(VC), and (2) none of g_3's children does subsume VC. The algorithm checks whether VC is contained in the DIRECT-PARENTS$_{VC}$ set. Since type(g_3) = type(VC) and content(g_3) = content(VC), the two classes VC and g_3 are found to be equivalent and no*



(a) Given a schema G and virtual class VC defined by "VC= hide [b] from C4".

(b) Prepare the schema graph G for insertion of VC by inserting the intermediate classes gi into G.

Class-Integration Algorithm:

Search:

DIRECT-PARENTS$_{VC}$ = { g3 }.

VC in DIRECT-PARENTS$_{VC}$

therefore stop and rename g3 by VC.

Edge manipulation:

None.

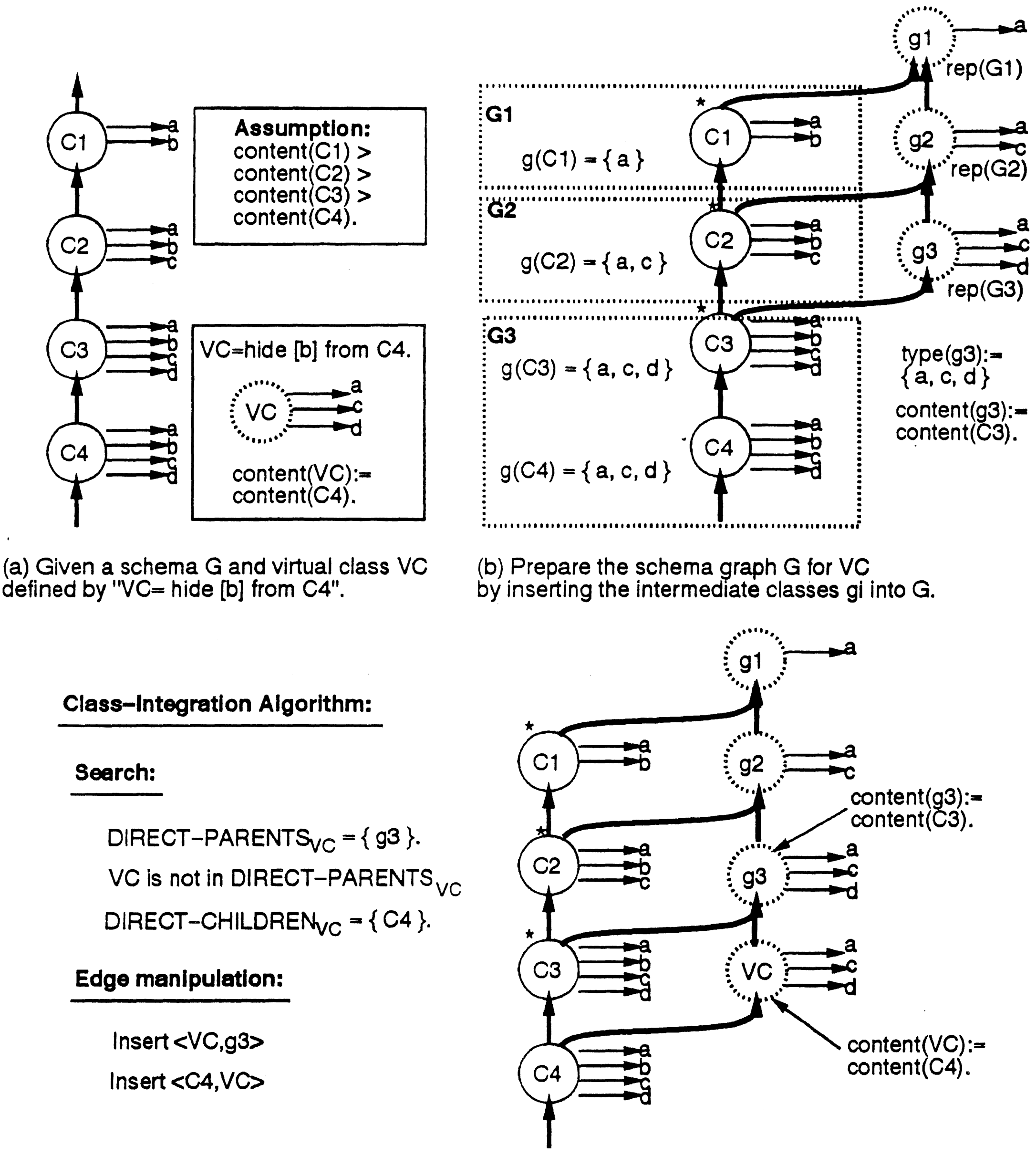
(c) Class Integration of VC into G: tracing through the algorithm.

(d) Class Integration of VC into G: the resulting schema graph G.

Figure 33: An Example of Complete Classification (Identical Set Contents).

new class needs to be added. The intermediate class g_3 is simply renamed by the name of VC (Figure 33.d). Since the class g_3 is already properly integrated in the schema graph, no further manipulation of the graph edges is needed.

Example 24. *Figure 34 presents another example of the classification algorithm given in Figure 32. The example schema graph in Figure 34.a is identical to the one in Figure 33.a with the only exception that all four classes C_i now have distinct object memberships, that is, content(C1) $\supset$ content(C2) $\supset$ content(C3) $\supset$ content(C4). VC is again derived by the view derivation "VC = hide [b] from C_4". First, the type hierarchy preparation task done by the Generate-Intermediate-Classes(G, VC) procedure proceeds as already explained in Example 23. Then, when applying the Class-Placement(G, VC) algorithm to add VC to the prepared schema graph G, the DIRECT-PARENTS$_{VC}$ set is again set equal to $\{g_3\}$. The algorithm checks whether VC is contained in the DIRECT-PARENTS$_{VC}$ set, which this time is not true for the following reason. While the type of VC is equal to the type of g_3 with type(g_3) = [a, c, d] = type(VC), the two contents are distinct with content(g_3) = content(C3) $\supset$ content(C4) = content(VC), in short, content(g_3) $\neq$ content(VC). The algorithm thus initiates the search for the DIRECT-CHILDREN$_{VC}$ set starting from g_3 on downwards the class hierarchy. C3 is not subsumed by VC, since content(C3) $\supset$ content(C4) = content(VC). However, since content(C4) = content(VC), the class C4 is subsumed by VC and we have DIRECT-CHILDREN$_{VC}$:= $\{C4\}$. Lastly, we need to adjust the edges of the schema graph in order to insert VC directly below the DIRECT-PARENTS$_{VC}$ set and directly above the DIRECT-CHILDREN$_{VC}$ set. For this reason, the edges $< VC, g_3 >$ and $< C4, VC >$ are added between VC and its direct parent and its child, respectively (Figures 34.c and 34.d). The resulting schema graph is shown in Figure 34.d.*



(a) Given a schema G and virtual class VC defined by "VC= hide [b] from C4".

(b) Prepare the schema graph G for VC by inserting the intermediate classes gi into G.

Class-Integration Algorithm:

Search:

DIRECT-PARENTS$_{VC}$ = { g3 }.

VC is not in DIRECT-PARENTS$_{VC}$

DIRECT-CHILDREN$_{VC}$ = { C4 }.

Edge manipulation:

Insert <VC,g3>

Insert <C4,VC>

(c) Determine the correct position for VC in G.

(d) Insert VC into G.

Figure 34: An Example of Complete Classification (Distinct Set Contents).

9 SOLVING THE SUBSET/SUBTYPE INCOMPATIBILITY PROBLEM

In Section 5, we have identified a class integration problem that arises due to the conflict between subset and subtype relationship requirements of a class. We have also shown an example of how this problem may be solved (Example 11). In this example, the *is-a* incompatibility problem was solved by creating additional intermediate classes (besides those required for a closed type hierarchy) that restructure the schema graph such as to allow for the correct insertion of the virtual class. In the following, we will present a general solution to the *is-a* incompatibility problem. More precisely, we will show that the algorithm that solves the type inheritance mismatch problem (Section 6) also successfully addresses the *is-a* incompatibility problem for the object algebra proposed in this paper.

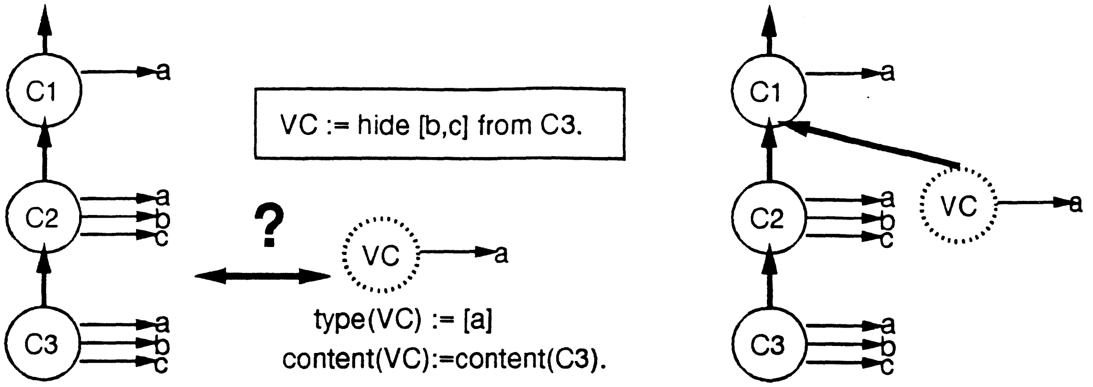
It is straightforward to see that virtual classes derived by most of the object algebra operators suggested in this paper are not subject to the subset/subtype incompatibility problem. This can intuitively be explained by the fact that the object algebra operators generate virtual classes that are compatible with the existing *is-a* hierarchy, meaning, either the derived class is both a subset and a subtype of the source class, or it is both a superset and a supertype of the source class. We leave it as an exercise to the reader to verify this observation.

Note that there is one exception to this, namely, the **hide** operator may indeed generate an *is-a* incompatible class as shown in Figure 36. The **hide** operator creates a virtual class VC with the same content as the source class C and a more generalized type than C. Due to its type, VC must be placed higher in the schema graph above C. In order to develop a solution for the *is-a* incompatibility problem in the context of the **hide** operator we consider the following three cases: (1) a class with a type equal to type(VC) already exists, (2) no class with a type equal to type(VC) exists but VC can consistently be integrated, and (3) no class with a type equal to type(VC) exists and none can be created.

In the first case, if a class C_k with type(C_k)=type(VC) already exists in the graph G, then VC can always be integrated - without conflict - by simply making it a subclass of C_k. This is so because the **hide** operator guarantees that the content of VC is smaller or equal to the content of any of the classes above the source class C of VC. Hence, $C_k \supseteq$ VC. This then implies VC *is-a* C_k, i.e., VC can always consistently be integrated into G by making it a subclass of C_k. We explain this situation with the example below.

Example 25. *In Figure 35, the virtual class VC is derived from the class C_3 using the* **hide** *operator. By the definition of the* **hide** *operator, content(VC)=content(C_3), and thus the set content of VC is always smaller or equal to the set content of any of the classes above C_3. Since VC $\succ$ C_3, VC needs to be integrated into G above C_3. There is one class in G above C_3 that has the same type as VC, namely, C_1. Therefore, VC can is integrated into G by making it a subclass of C_1.*

In the second case, a class C_k with type(C_k)=type(VC) does not exist in the graph G but VC can be consistently integrated. By assumption, this case does not represent a problem. An example of this situation is discussed below.



(a) Class C with type(C)=type(VC) exists in G.

(b) Successful integration of VC.

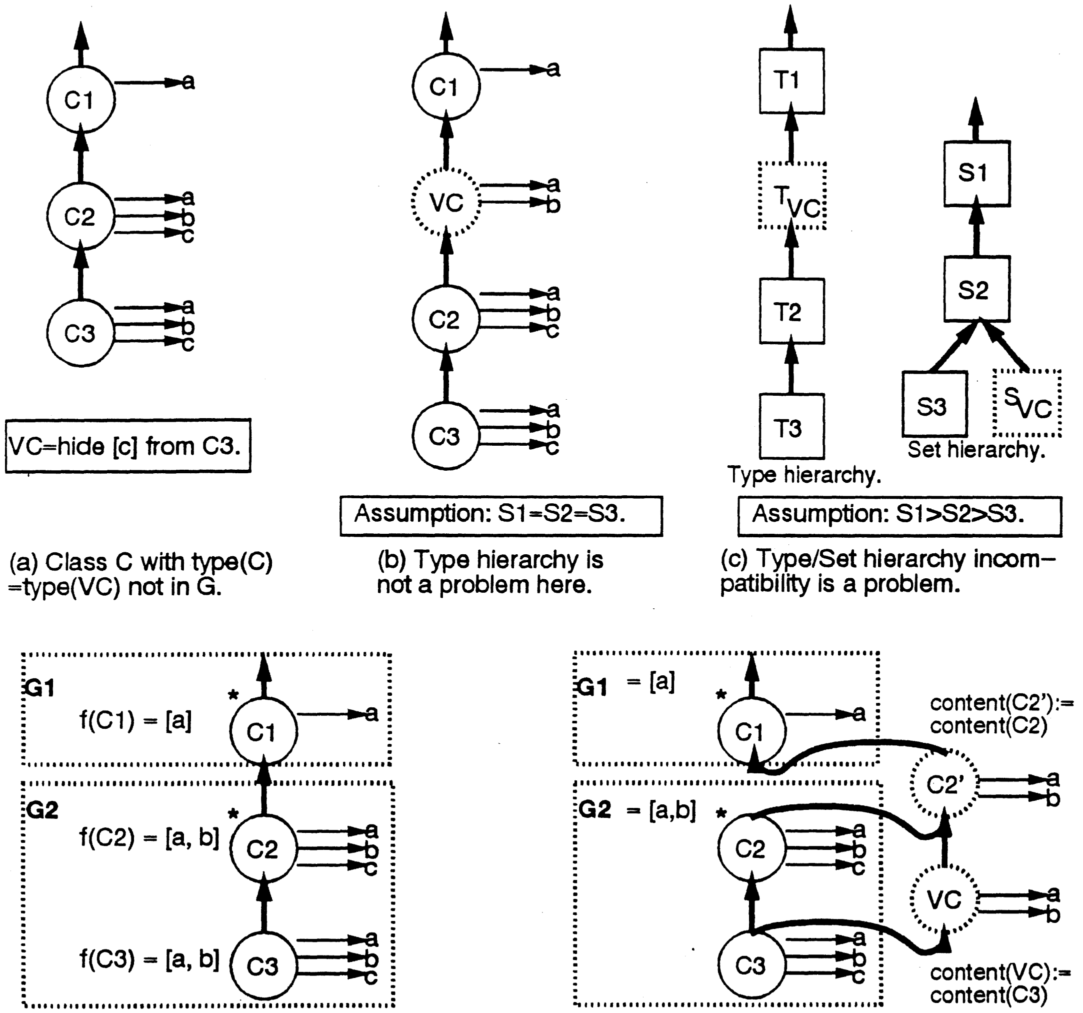
Figure 35: The *is-a* Incompatibility Problem: A Class C with VC's Type Exists in Schema.

Example 26. *In Figures 36.a and 36.b, we assume that the set contents of all classes are identical. Then the set relationships can be ignored; this reduces the* ***is-a*** *incompatibility problem to the type inheritance mismatch problem, which fortunately we already know how to solve. In Figure 36.b, VC is integrated into G by placing it directly below C1 and above C2.*

Lastly, we take a look at the third case in which no consistent integration of VC can take place. For instance, if we drop the simplifying assumption of identical sets in the previous example, then the fact that a class C_k with type(C_k)=type(VC) does not exist in G results in conflicts caused by a mismatch between the subset and the subtype hierarchy. In this case, a subset/subtype conflict may occur since VC's content may be smaller than the content of any of the other classes above its source class. VC should be **below** all of them in terms of the set relationships and **above** some of them due to the subtype relationships.

Example 27. *An example of this situation is given in Figure 36.c, which is equal to Figure 36.b except for the fact that the set contents of all classes are now distinct. In this example, VC is required to be* **above** *C2 due to their subtype relationship and* **below** *C2 due to their subset relationship. Clearly, there is no consistent position for VC in G. The problem in Figure 36.c can be solved by introducing an intermediate class C2' with type(C2')=type(VC) and content(C2')=content(C2). VC can then be integrated by placing it directly below C2' as depicted in Figure 36.e.*

It is interesting to note that the solution for the *is-a* incompatibility problem proposed in the above example is exactly what the type lattice preparation algorithm presented in Section 6 would do. This is also demonstrated in Figures 36.d and 36.e by displaying detailed steps of the type lattice preparation algorithm for the example. We now argue that this is not coincidence but rather that it is always the case. This can be explained as follows. Our solution to the type inheritance problem introduces intermediate classes, one for each equivalence group of G with respect to VC (Section 6). Since type(<**hide**-source-class> ⊓ VC) = type(VC), this then implies that a class with a type equal to VC is created. Consequently, application of the type hierarchy preparation will always generate a class with the required type of VC. In other words, the type hierarchy preparation reduces the problem to the situation shown in Figure 35. As



(a) Class C with type(C) =type(VC) not in G.

(b) Type hierarchy is not a problem here.

(c) Type/Set hierarchy incompatibility is a problem.

(d) Use type hierarchy algorithm to solve incompatibility problem.

(e) The final schema graph also solves the incompatibility problem.

Figure 36: An Example of Solving the *is-a* Incompatibility Problem

discussed earlier, in this case *is-a* incompatibility is no longer a problem. In short, we thus have shown that the type hierarchy preparation algorithm solves the *is-a* incompatibility problem for the general case (see also Figures 36.d and 36.e). For this reason, we need not develop an independent solution to address the *is-a* incompatibility problem. Lastly, we would like to note that if one were to use a different class derivation language than the object algebra proposed in this paper, then one might have to develop a general algorithm to solve the *is-a* incompatibility problem (See for example Figure 11). Further investigation of this problem is however beyond the scope of this paper.

10 CLASSIFICATION CUSTOMIZED FOR OBJECT ALGEBRA

In this section, we customize the general class integration algorithm presented in this paper for each of the six object algebra operators proposed in Section 3.2. We demonstrate that in the majority of cases the type hierarchy preparation procedure, which has quadratic complexity, need not be run. For the following, we refer to the simple class placement procedure defined in Section 7 by algorithm A and to the more complex algorithm defined in Section 8 that first generates the appropriate intermediate types and then places the virtual class in the modified schema graph by algorithm B. Recall that complexity(A) is linear and complexity(B) is quadratic, assuming a *subsumes()* function with constant complexity.

derivation operators	intermediate types	final location	algorithm	complexity
select	no	below SC	A	$O(\|E\|)$
refine	no	directly below SC, no children	direct	$O(1)$
hide	yes	above SC	B	$O(\|E^2\|)$
union	no	above SC	A	$O(\|E\|)$
intersect	yes	below SC	B	$O(\|E^2\|)$
diff	no	below SC	A	$O(\|E\|)$

Legend

algorithm A = class placement (given correct type hierarchy)
algorithm B = type hierarchy preparation and class placement

Figure 37: A Table of Class Integration Customized for Each Object Algebra Operator.

The table in Figure 37 presents a summary of the results of customizing class integration for each derivation operator, while a more detailed discussion is given next. The first column lists the object algebra operators, the second whether or not type generation may be required, the third indicates the final location of the virtual class in the schema graph, and the fourth and fifth columns indicate the algorithm and complexity needed to accomplish the class integration of the virtual class generated by the respective operator.

The **refine** operator defined in Section 3.2 generates a virtual class VC with a more specialized type description than a given source class SC by adding new property functions to the type description of SC. VC has the same set content as SC. Having a more specialized type description and the same set content as SC, VC becomes a subclass of SC and thus must be placed below SC in the schema graph. In spite of the generation of a new type (and a new property function), we do not need to add intermediate classes to the schema during class integration. This is so, since all property functions added to the **refined** class are distinct from existing functions. Hence the equivalence partition of G with respect to SC is equal to the equivalence partition of G with respect to VC (see Definition 18 for a definition of the equivalence partition). By Theorem 4, this implies that all necessary types do already exist in the schema graph. We thus have shown that algorithm B is not needed for the class integration of a **refined** class.

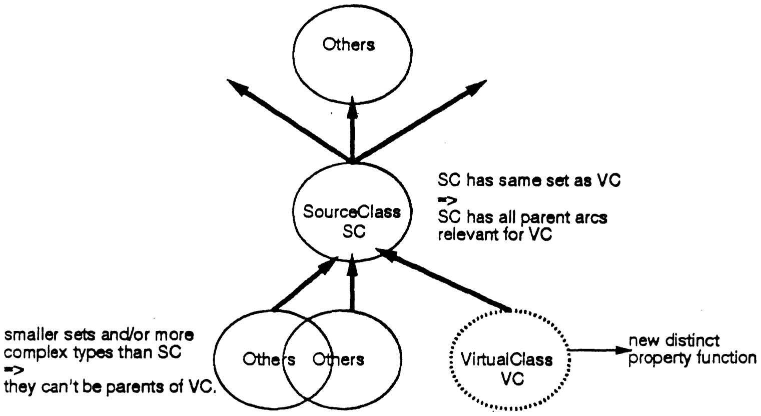

Others
SourceClass SC
SC has same set as VC
=>
SC has all parent arcs relevant for VC
smaller sets and/or more complex types than SC
=>
they can't be parents of VC.
Others
Others
VirtualClass VC
new distinct property function

Figure 38: Linear Class Integration for the **refine** Operator.

Next, we show that the integration of a virtual class VC generated by the **refine** operator can be reduced to a simple O(1) algorithm requiring no search – rather than having to apply the linear class-placement procedure in algorithm A. As shown in Section 3.2, the **refined** virtual class is always a (direct or indirect) subclass of its source class. Based on the example graph in Figure 38, we can explain why VC cannot be placed any lower in the class hierarchy. All classes below the source class SC have a restricted set membership and/or a refined type description of SC. Since the content of VC is equal to the content of SC, none of these children of SC can become a parent of VC. Based on Figure 38, we can also explain why the virtual class VC will not have any direct parents besides its source class SC. Since content(SC)=content(VC) and the types of SC and VC are equivalent with the exception of the new property function of VC, any parent of VC will already be a parent of SC. VC would consequently inherit these parent relationships through SC. Lastly, we argue that the virtual class VC cannot have any subclasses of its own. VC has a type description distinct from the types of all existing classes due to the newly defined property function and therefore it would be incorrect for any of the existing classes to inherit this new property. We can thus conclude that the **refined** virtual class VC always has to be placed as direct subclass of its source class SC with no children of its own, i.e., direct-parents(<**refined**-*virtual*-class>) := { <*source*-class> } and direct-children(<**refined**-*virtual*-class>) := { }.

The **hide** operator defined in Section 3.2 modifies the type description of a class SC by hiding some of its property functions. It thus generates a new class VC with a more general type description and the same set content as SC. As for instance demonstrated in Example 10, we are forced to generate intermediate classes since VC may have a new type description (composed of existing property functions) that does not yet exist in the object schema. Therefore, the integration of a virtual class generated by the **hide** operator may require the application of the full-blown algorithm B.

The **select** operator is a set-manipulating operator that generates a virtual class VC by selecting a subset of object instances from a given class SC. The final location of VC is below SC in the schema graph, but not necessarily as a direct subclass of SC. It is easy to see that we do not need to generate any intermediate classes, since VC has the same type as SC and SC is already properly integrated into the schema graph. Therefore, simple class placement using algorithm A is sufficient for the **select** operator.

The **union** operator defined in Section 3.2 builds a new class VC by combining the members of two source classes SC1 and SC2 into one set. The type description of the virtual class then is set equal to the lowest common supertype of the two sources classes as defined in Definition 4. Clearly, this makes VC a supertype and a superset of both its source classes, and thus VC is placed above them. Even though manipulating the type description of the two source classes, the integration of a class derived using the **union** operator does not require the introduction of intermediate classes. The reason for this can be explained as follows. The type of VC is equal to type(VC):=SC1 $\sqcap$ SC2, which corresponds to the lowest common supertype of SC1 and SC2. By Theorem 16, this lowest common supertype has to exist for all pairs of classes in a closed schema graph. Since SC1 and SC2 were members of the schema graph before the inserting of VC, a class with a type equal to the lowest common supertype of SC1 and SC2 must already have existed in the schema graph. Consequently, the simple class placement algorithm A with linear complexity is sufficient.

The **intersect** operator defined in Section 3.2 builds a new class VC by constructing a set of object instances that are members of two classes SC1 and SC2. The type description of the virtual class is equal to the greatest common subtype of the two sources classes as defined in Definition 3. VC is placed in the subgraph below both SC1 and SC2. The integration of a class derived using the **intersect** operator may require the introduction of intermediate classes since its new type did not necessarily exist before in the schema graph. We therefore need to use algorithm B with quadratic complexity.

The **difference** operator generates a virtual class VC consisting of a set of object instances that are members of the first but not of the second source class. This is effectively equivalent to the **select** operator, since "VC := **diff**(SC1,SC2)" is equivalent to "VC := **select** (e:SC1) **where** (e not in SC2)". Therefore, the virtual class has the same type as the first source class. And, since this type has already been properly integrated into the schema graph, no generation of intermediate classes is needed. In short, algorithm A is sufficient for the **difference** operator.

11 RELATED WORK

An immediate extension of the view mechanism from relational databases to OODB systems is to define a view to be equal to an object-oriented query. In fact, many efforts of defining views for OODBs follow this approach; that is, they suggest the use of the query language defined for their respective object model to derive a virtual class. Some examples are view mechanisms for the Fugue Model in [8], for the Orion model in [11], and for integrating databases in [9]. *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. In this sense, *MultiView* 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 [8], or they are attached directly as subclasses of the schema root class [11]. Scholl et al.'s recent work [21] is one of the exceptions. They sketch the class integration process for a selected subset of the operators of the query language COOL. This work is similar in flavor to what we present in Section 3.2 on object algebra. Namely, they determine whether a derived class should be placed lower or higher than their source classes. This localized class integration approach is directly guided by the derivation of a virtual class, and it is not, as we have shown in this paper, a solution to finding the globally most appropriate location in the schema graph. Scholl et al. [21] do not consider the problem of generating multiple view schemata, which is an integral part of *MultiView*. *MultiView* can thus be considered to be a compatible extension of their work.

Abiteboul and Bonner [1] present a view mechanism for the O_2 database system. In this context, they also discuss class integration as an important problem. However, their suggested solution is again a simplistic approach that results in partial rather in complete classification (See Section 4). No precise algorithm for class integration is presented.

Rundensteiner et al. [18] discuss the integration of virtual classes derived using set operators into a schema graph. Their work focuses on the semantics of set operators and the inheritance of property characteristics, such as, single- versus multi-valued or required versus optional. Again, they discuss the relative positioning of the virtual class with respect to its source classes without presenting a general solution for classification.

Tanaka et al.'s work on schema virtualization [24] does not distinguish between the task of integrating derived classes into a common schema and the task of generating view schemata. While recognizing the need for class integration, they do not present a general classification algorithm. They point out that work is needed for developing a systematic approach towards view specification in OODBs. 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 to a simple yet powerful mechanism. In summary, *MultiView* is a more systematic solution approach compared to their rather ad-hoc proposal.

Shilling and Sweeney's approach [22] for supporting object-oriented views is based on extending the concept of a class from having one type (one ADT interface) to having multiple type interfaces. We accomplish the same goal of specializing types by using the type refinement capability of the generalization hierarchy. Our work is simpler, however, since it does not require the extension of the traditional class concept. Shilling and Sweeney's work focuses on one class only, and the effects of multiple interfaces on the class generalization hierarchy are not addressed. Consequently, the question of whether a new type interface associated with an existing class is properly integrated with the complete schema remains open. Of course, class integration, which does not become an issue when dealing with an individual class only, is not addressed.

Algorithms for special forms of the classification problem have been proposed in the Artificial Intelligence literature. Schmolze and Lipkis [20], for instance, describe a classifier for 'concepts' in the KL-ONE Knowledge Representation System. The KL-ONE Knowledge Representation scheme does not include behavioral abstractions and abstract data types as done in an object-oriented model. Hence, the type inheritance mismatch problem is not addressed by their solution. Furthermore, the KL-ONE classifier deals with single-inheritance only, while our class placement algorithm can handle both multiple-inheritance schema graphs. More importantly, they do not assume that a 'concept' has an associated set of object members. Consequently, they do not have to tackle the problem of *is-a* incompatibility between the subset and the subtype hierarchies underlying the general representation scheme. Lastly, since the derivation of new classes is accomplished in *MultiView* using well-defined object algebra operators, we are able to reduce the complexity of classification depending on the operator used for derivation. This issue is not addressed in [20].

Our work on classification probably comes the closest to the research by Missikoff et al. [13] on inserting types into a lattice structure.. Rather than dealing with an object-oriented model, they assume a simple record-oriented type system. Our classification algorithm, on the other hand, is extended to be applicable to a class generalization hierarchy. Since a class represents both a type and a set, our classification algorithm solves the *is-a* incompatibility problem. This problem does not arise, and thus is not addressed in [13], when dealing with classification in a type structure rather than in a schema graph..

12 CONCLUSIONS AND FUTURE WORK

12.1 Summary and Contributions

MultiView is a novel approach for supporting multiple object-oriented views with an object-oriented view defined to be a *virtual, possibly restructured, subschema graph* of the global schema. 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.

MultiView breaks view specification and maintenance into the following three subtasks: (1) customization and derivation of virtual classes, (2) integration of virtual classes into *one* consistent global schema graph and (3) the specification of arbitrarily complex view schemata on this augmented global schema. In this paper, we have presented a solution to the second subtask, while solutions to the first and the third subtasks of *MultiView* are given in [17] and in [19], respectively. This second subtask, namely, the integration of all virtual classes into one global schema graph is the key feature of *MultiView* that ultimately supports the formation of arbitrarily complex view schema graphs composed of both base and virtual classes. In other words, the integrated global schema graph corresponds to the backbone of our view support system based on which all view schemata are being designed.

Rather than requiring manual placement of classes in the schema graph and then checking the entered information for consistency, *MultiView* supports the automatic integration of classes. Automatic classification does not only prevent the introduction of inconsistencies into the schema, but it also considerably simplifies the task of the view definer. This decreases the view creation time, and, more importantly, it allows a non-database expert to specify an application-specific view on his or her own without having to be concerned with the tedious class integration task.

For these reasons, we present in this paper an algorithm for the automatic classification of virtual classes into a global schema graph. We have identified two class integration problems, called the type inheritance mismatch and the *is-a* incompatible subset/subtype hierarchies problem. Our classification algorithm solves both problems. Both solutions are based on type lattice classification proposed in the literature, the essence of which is the creation of additional intermediate classes that restructure the schema graph. We present proofs of correctness and a complexity analysis for the classification algorithm.

Furthermore, we characterize classification requirements of virtual classes derived by different object algebra operators. This characterization helps us to reduce the complexity of the classification task for most cases. For instance, we are able to reduce classification from quadratic to linear complexity for classes derived by the Select, the Union, and the Difference operators and to constant complexity for those derived by the Refine operator.

To summarize, the contribution of our work is two-fold: (1) the idea of generating one global schema graph incorporating both virtual and base classes as foundation of view support and (2) the development of a general class integration algorithm for object-oriented data models that supports the generation of such a graph.

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 enrich these systems by allowing them to support a more powerful notion of views. A major contribution of the proposed approach lies in its simplicity compared to alternative proposals [22, 7], and hence the potential ease in implementing it with existing OODB technology.

12.2 Future Work

While the correctness of the algorithms presented in this paper has been tested in isolation, a prototype implementation of the complete *MultiView* system on top of some existing object-oriented database system would represent a good means for further evaluating our approach. We are in the process of evaluating available OODBs for their suitability. This evaluation includes the requirement for the OODB to support some basic features, such as that an object instance can be a member of many classes simultaneously and thus can take on different types [21]. Several implementation issues immediately arise from this requirement, such as the development of efficient strategies for method resolution. Other important issues, such as query processing techniques on views, materialized views, and view updates, which have been extensively studied in the relational model, must now be reexamined in the context of object-oriented data models.

In this paper, we have restricted the set of object algebra operators to be object-preserving [21]. We believe strongly that this is sufficient for many application domains, in particular, since the join operator can be simulated using the refine operator. However, it would be interesting to investigate whether, and if so how, *MultiView* could be extended to also handle object-generating algebra operators [11, 8].

As indicated in Section 4, the classification problem for object-oriented models is not decidable since it may involve the comparison of arbitrary functions and predicates. Hence, the development of a realistic *subsumes()* function for some of the emerging object models needs to be investigated. The goal of such a project would be not to restrict the expressive power of the model nor the constructs used for deriving new classes, while guaranteeing that the *subsumes()* function stays computable.

In this paper, we took the extreme stance of requiring complete classification, i.e., of automatically forcing the placement of a virtual class in its (syntacticly) most appropriate position. A more detailed discussion on this issue can be found in Section 4. If one were interested in dropping this stringent restriction and allow the user to place new classes in non-optimal but nontheless syntacticly correct positions, then our work on the classification algorithm would still be a useful component of classification. In particular, it could serve as core for a consistency checker that verifies the correctness of a manual class placement, once specified. The automatic classifier could also be utilized to guide the user during an interactive view specification phase.

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 our long-term goals is to apply *MultiView* to address this problem.

Lastly, we propose view optimization as a new open problem that arises from our research on object-oriented views. View optimization is concerned with restructuring the view specification such as to minimize the number of intermediate classes that need to be created. View optimization attempts to minimize the size of the resulting schema graph in terms of the number of (virtual) classes, whereas query processing on views is concerned with finding the best execution plan for a given view schema. View optimization is permanent since it influences the final schema structure whereas view query optimization is temporary since it is concerned with the performance of executing an individual query. As a matter of course, view optimization may have an influence on the effectiveness of query optimization, and thus should take the potential impact of its decisions on query processing into account.

References

[1] Abiteboul, S., and Bonner, A., "Objects and Views," in *Proc. SIGMOD'91*, pp. 238 – 247.

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

[3] 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.

[4] Bouzeghoub, M., "MORSE: a functional query language and its semantic data model," *Proc. of 84 Trends and Applications of Databases*, IEEE-NBS, Gaithersburg, 1984.

[5] Brachman, R. J., and Schmolze, J. G., "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, 9, 1985.

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

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

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

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

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

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

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

[13] Missikoff, M., and Scholl, M., "An Algorithm for Insertion into a Lattice: Application to Type Classification", *Foundations of Data Organization and Algorithms*, 3^{rd} Int. Conf., FODD'89, France, June 1989, pp. 64 – 82.

[14] Missikoff, M., "MOKA: A User-friendly Front-End for Knowledge Acquisition," *Int'l Workshop on Database Machines and Artificial Intelligence*, Minowbrook, N.Y., July 1987.

[15] 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.

[16] 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.

[17] Rundensteiner, E. A., "*MultiView:* A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases", *18th Int. Conference on Very Large Data Bases* (VLDB'92), Vancouver, Canada, Aug. 1992.

[18] Rundensteiner, E. A., and Bic, L., "Set Operations in Object-Based Data Models", in *IEEE Transaction on Data and Knowledge Engineering,* to appear in Volume 4, Issue 3, June 1992.

[19] Rundensteiner, E. A. and Bic, L., "Automatic View Schema Generation in Object-Oriented Databases", Dept. of Information and Computer Science, Univ. of Cal., Irvine, Tech. Rep. 92-15, Jan. 1992.

[20] 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.

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

[22] 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.

[23] 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.

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