Consistent Schema Version Removal: An Optimization Technique for Object-Oriented Views *

Viviane M. Crestana
Department of EECS
University of Michigan
Ann Arbor, MI 48109-2122
viviane@eecs.umich.edu

Amy J. Lee
Department of I. & S. Mgmt.
HK Univ. of Science and Technology
Kowloon, Hong Kong
imalee@ust.hk

Elke A. Rundensteiner
Department of CS
Worcester Polytechnic Institute
Worcester, MA 01609-2280
rundenst@cs.wpi.edu

Abstract

Powerful solutions enabling interoperability must allow applications to evolve and requirements of shared databases to change, while minimizing such changes on other integrated applications. Several approaches, such as the transparent schema evolution system (TSE) by Ra et al., schema versions by Lautemann, and integrated views by Bertino, have been proposed to make interoperability possible by using object-oriented techniques. These approaches may generate a large number of schema versions over time resulting in an excessive build-up of classes and underlying object instances, not all being necessarily still in use. This results in degradation of system performance due to the view maintenance and the storage overhead costs. In this paper, we address the problem of removing obsolete view schemas. We characterize four potential problems of schema consistency that could be caused by removal of a single derived class. We demonstrate that schema version removal is sensitive to the order in which individual classes are processed, and present a formal dependency model that captures all dependencies between classes as logic clauses and manipulates them to make decisions on class deletions and non-deletions while guaranteeing the consistency of the schema. We have also developed and proven consistent a dependency graph (DG) representation of the formal model. Lastly, we present a cost model for evaluating alternative removal patterns on DG to assure selection of the optimal solution. The proposed techniques have been implemented in our Schema View Removal (SVR) tool. Lastly, we report experimental findings for applying our techniques for consistent schema version removal on the MultiView/TSE system.

KEYWORDS: Object-Oriented Views, Schema Version, Transparent Schema Evolution, Interoperability, Evolving Software Applications, Performance Evaluation.

*This work was supported in part by the NSF RIA grant #IRI-9309076, NSF NYI grant #IRI 94-57609, and the University of Michigan Faculty Award Program. We would also like to thank our industrial sponsors, in particular, AT&T, Illustra, IBM, and Intel. Viviane M. Crestana is also grateful for support for her graduate studies from the Conselho Nacional de Pesquisa CNPq and from the IBM Toronto, The Center for Advanced Studies.

1 Introduction

1.1 Motivation

Systems that enable interoperability for multiple software applications in dynamic environments must allow applications to evolve and flexibly change their data requirements, while minimizing or even eliminating the impact of such change on other existing applications. Several approaches [3, 11, 15, 17, 18] have been proposed to support schema evolution and/or data integration needs by using object-oriented views. Nevertheless, these approaches run into a potential problem of generating a large number of schema versions over time resulting in an excessive build-up of classes and underlying object instances, not all being necessarily still in use. Shortcomings accompanied by this problem are degradation of system performance due to the maintenance of the materialized views and the storage overhead costs.

In this paper, we thus address this problem of excessive build-up of schema versions. The main goal of this work is the effective schema version removal (SVR) of redundant or obsolete view schemas without impacting existing applications. An additional advantage of reducing the number of potential classes is that it makes it easier for application developers to determine which view schema to run against. As our experimental studies presented in Section 7 confirm, performance improvements will be achieved for the propagation of updates from a base class to its derived classes [1] given fewer materialized derived classes. In this paper, we demonstrate our solution of the SVR problem on the MultiView [13] and the TSE systems [18]. However, while we base our experiments on given systems, we believe the proposed methods also benefit other similar systems [1, 22, 24]. Removing obsolete view schemas will also shorten the forward/backward conversion function search for Lautemann's work [15] on schema versions, and a similar technique could be used for view maintenance purpose for alternate view mechanisms proposed by others [11, 23]. For a further discussion of benefits see Section 8.

1.2 Our Schema Version Removal (SVR) Approach

In this paper, we first characterize several potential schema consistency problems that could be caused by removal of a *single* derived class, such as type-effect, derivation-dependency, etc [9]. Then we outline our strategies for solving each of these problems. Key ideas here include strategies for the redefinition of derived classes and for the migration of properties for preservation [7]. Furthermore, we demonstrate that *schema version removal*, i.e., removing multiple derived classes, is sensitive to the order in which classes are processed. Our solution for this multiple class removal problem includes the development of a formal model of capturing all dependencies between class deletions and non-deletions as logic clauses. The consistency of the resulting schema can then be guaranteed as long as at least one valid variable assignment exists for all clauses. In this paper we now present the complete formal model, called the Dependency Model, which represents the formal basis of our solution approach.

[1] The terms *derived* classes and *virtual* classes are used interchangeably in this paper.

Based on the Dependency Model, we have developed a dependency graph (DG) representation capturing these class interdependencies [2]. Sets of associated rules for DG generation and transformation that reduce these interdependencies by making decisions about deletion and non-deletion of classes are presented. Using our dependency model, we prove those rules to be consistent. In the process of optimizing the DG, our system will also identify conditions of *mutually exclusive removal,* namely where removal of one derived class would prevent removal of others in the future. In other words, during this stage of DG manipulation, we would be generating alternate (non-unique) reduced DG solutions. Hence, some trade-off criteria are necessary to determine which among these solutions to select.

To address the problem that view schema removal is sensitive to the order in which individual classes are processed, in this paper we establish a cost model that guides the removal process by selecting among alternative removal patterns on DG in terms of their associated view maintenance costs. Our schema removal strategies assure consistency, in the sense that no other class and/or class relationships in the global schema as well as in all other view schemas are unexpectedly affected and that the resulting schema meets all schema invariants.

Based on these techniques, we have implemented a first prototype of the SVR tool on top of the MultiView/TSE system [13]. Lastly, we report on experimental studies we have conducted to validate our approach. Our evaluation supports on one hand the view removal assumptions underlying our SVR approach, and on the other hand it also demonstrates the impact of the SVR removal strategies on the performance of the MultiView/TSE system.

1.3 Related Work

Several other approaches towards transparent schema evolution using views have been recently presented in the literature [4, 5, 11]. However, none have considered the schema removal problem as characterized in this paper. Some of the view systems presented in the literature [11, 23] assume that the derived classes are not integrated in the global schema. Therefore, removal of a derived class in such a system does not impose any constraints regarding the consistency of the global schema, since the view schema and the global base schema are not integrated. In the approach described in [11], a view schema is composed of only derived classes, and therefore in order to have a base class in a view schema, they provide an *identity-derived class* that has the same type and extent as the base class. In other words, such an identity-derived class is effectly a full copy of the base class into the context of the view schema. It has not been looked at what happens to such identify-derived classes if the base classes they are derived from changes. This approach, while avoiding the integration and therefore possibly the removal consistency problem, has the drawback of duplication of information. In systems where the derived classes are integrated in the global schema [1, 19, 22, 24], code-reuse as well as sharing of data is assured. Our work addresses the removal of such integrated derived

[2]The graph representation is more suitable for implementation in our SVR tool, as it can closely be mapped to a data structure implementation.

classes.

In systems using conventional versioning rather than object-oriented view mechanisms, schema removal is likely to be less of an issue as well. Typically, complete copies of class versions are created without being integrated into one global schema. This increases system overheads, but interdependencies as mentioned above do not exist between derived and base classes. Similarly, the object instances associated with different schema versions are typically copies of their source data objects. This duplication of state would allow for the removal of one of these copies with less or no side-impact – however at the cost of additional storage overhead and potential problems of value inconsistencies between different object versions over time. Some versioning systems however provide code reuse and sharing of data. In the work by [16], class versions are placed in one global schema, where versions of a particular class form a sub-hierarchy in the global schema, enabling the sharing of object instances among different versions. In such an approach, removal of class versions has side-effects on the remaining schema and thus would have to be carefully considered as in our work. As far as we know, the issues of removal have not been studied in the context of [16].

Our work is also related to the issue of *schema consistency* as studied in the context of OODB schema evolution work, e.g., Orion [2] or O2 [10, 25]. Such work must also establish what constitutes a consistent schema, and must assure that transformation operations indeed result in a consistent final schema. However, the objectives as well as the strategies of achieving these goals are rather distinct. Namely, while the purpose of traditional schema evolution systems is to actually perform the schema evolution operation exactly as specified by the user and to propagate it along the class hierarchy, the SVR system would only perform version cleanup updates assuring that they are non-intrusive and are not propagated to (affect) other classes.

1.4 Outline for Remainder of the Paper

The remainder of the paper is structured as follows. In Section 2, we give an overview of the underlying schea versioning and object-oriented view technology that drive the development of our solution approach. A discussion on the issues related to schema consistency when removing a class from the schema and solutions for each one of the issues is given in Section 3. In Sections 4 and 5, we describe the interdependency problem when we consider deletion of not only one but several classes in the schema, and we present present solutions for the multiple class removal, namely the dependency model and the dependency graph approach, respectively A cost model for measuring efficiency of the schema is proposed in Section 6, while our performance studies are presented in Section 7. Section 8 summarizes our contributions and presents possible future work.

2 The TSE System: Problem Description

In transparent schema evolution systems [17, 18, 21], each user [3] constructs a customized interface by selecting a subset of classes from the shared database. All the schema change requests made by the developer are issued against her customized view schema with which she is familiar (Figure 1(a)). The key idea of the TSE technology is to compute a new view schema (see VS2 in Figure 1(b)) that reflects the desired change instead of modifying the existing view schema in place (see VS1 in Figure 1(b)). This approach allows old application programs (program1) to continue to run against the schema (VS1) they were designed for.

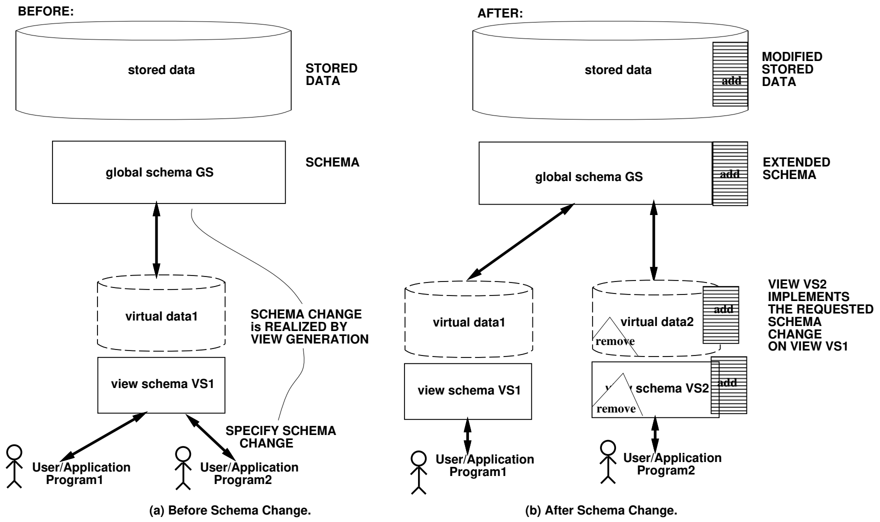


Figure 1: The TSE System Approach.

Ra et al. have built one instance of a TSE system on top of the MultiView object-oriented (OO) view system [13, 19], which is one of the first fully implemented OO view systems providing updatable and incrementally maintained materialized views [13, 14]. MultiView supports a full range of derived classes defined by a single object algebra operator, such as *select, hide, union, intersect*, etc. [19]. Base and derived classes are integrated into a global schema GS, and derived classes are treated as first-class database citizens such that properties shared among base and derived classes are defined exactly once in the global schema. A view schema (VS) is composed of a set of classes that are is-a connected. Each class V in a view schema VS corresponds to a class C (either base or derived) in the global schema GS, with C denoted by GS-class(V), that is, V is the "name" of class C in the view schema VS. This not only assures code-reuse but also allows the values of properties shared among a base object and all its derived instances to be stored only once.

The invariants of the global schema (GS) are summarized here:

[3]The user here means an experienced user, i.e., a developer, but not an end user.

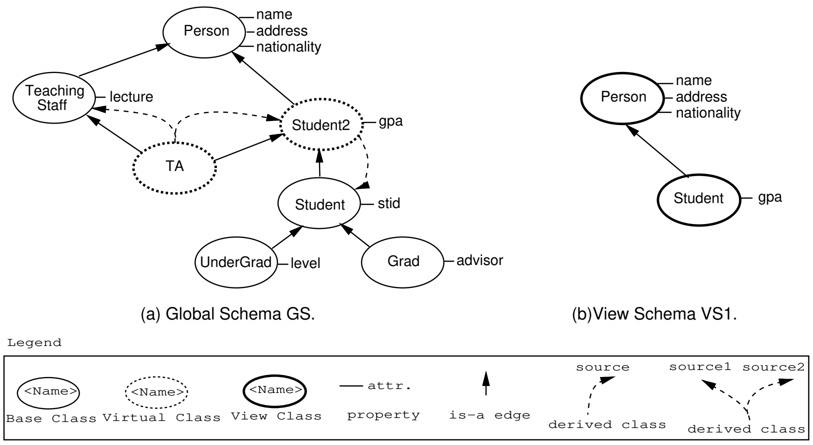


Figure 2: The University Database Example.

SI1. There is one class called **root** in the GS that is a superclass of all classes in the GS.

SI2. The IS-A hierarchy is a connected directed acyclic graph (DAG).

SI3. A class inherits all properties defined in its superclasses, except for the overridden properties.

SI4. Any property in the GS is defined only once as a local property of a class.

SI5. The derivation chain between derived classes and their source classes is acyclic.

SI6. A derived class is defined by a single object algebra operator.

Throughout the paper we use the university database schema depicted in Figure 2(a) as our running example. In this example we have the base classes *Person*, *Teaching Staff*, *Student*, *UnderGrad* and *Grad*, and two derived classes: *Student2* defined as hide *stid* from *Student* and *TA* as intersect *Student2* with *TeachingStaff*.

The initial view schema VS1 contains two classes *Person* and *Student* (see Figure 2(b)) that correspond to the base class *Person* and the derived class *Student2* in GS, respectively (see Figure 2(a)), i.e., class *Student* in the context of view schema VS1 is in reality class *Student2* from GS. The developer then issues two subsequent schema changes against this view schema VS1, namely, to delete the *nationality* property from the *Person* class and to delete the *gpa* property from the *Student* class. This would generate two view schemas VS2 and VS3 (see Figure 3(a)). New derived classes are created for realizing these schema changes and are integrated into the global schema (see Figure 3(b)). This example shows one of the schema change operators supported by the TSE system, namely the *delete-attribute* operator. A description of how view schemas are derived from the GS can be found in [13, 19], and a complete reference on the schema change operators supported by the TSE system can be found in [17, 18].

Assume the view schema VS1 is no longer in use, and we would like to remove it, i.e., we would like to remove all the classes that participate in that view schema, namely *Person* and *Student2*, from the global schema without affecting the other classes. Note that this is different than performing a schema evolution

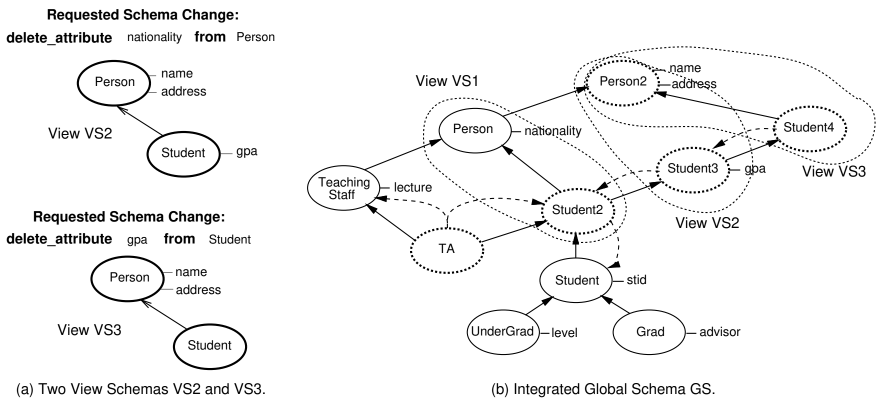


Figure 3: The University Database after TSE Operations.

operation, where the effects of removal would be reflected in the rest of the schema without regard to whether this affects existing applications or not. Removing VS1 could cause some problems, for example, it could violate the global schema invariant **SI4**. If *Person* is removed, in order to avoid the undesirable modification of the types of *Person*'s subclasses, the attribute *nationality* has to be migrated to *TeachingStaff* and *Student2* (this violates **SI4**). Similarly, removing *Student2*, could cause the classes rooted at *Student* to "lose" the attribute *nationality* (if, for example, *Student* is made a subclass of *Student3*). Deciding which and how obsolete view schemas can be removed safely from the GS without jeopardizing the rest of the view schemas (and thus the applications running on top of them) is the main focus of this paper.

3 Consistent Schema Removal: Problems and Solutions

In this section we first describe the problems associated with removing a single class, then outline solution strategies for these problems, and present the class-removal algorithm that integrates the strategies. Lastly, we consider the deletion of multiple classes in a view schema. Note that when we talk about removing a class we are "not" referring to the delete-class schema change operation, which will have effects on the rest of the schema. Instead it is our intention to remove the class from the global schema without affecting the rest of the schema.

3.1 Problems of Schema Removal

When removing a view schema $VS_{obsolete}$, all classes $C \in VS_{obsolete}$, whether base or derived, are scheduled to be removed one at a time. A class removal process is said to be *transparent*, if other existing classes in the global schema (GS) besides the removed class, have the same class types and extents they had before the removal. The schema version removal process stops when no more class $C \in VS_{obsolete}$ can be removed transparently. To assure a *transparent removal* of a view schema, i.e., to assure that other schemas are not affected, SVR needs to guarantee:

TR1. Every class that participates in any other view schema should be maintained in the GS and should have exactly the same type and extent as before the removal.

TR2. Any two classes *C* and *D* in a view schema, besides the one under removal consideration, that are IS-A related before the removal, should still be IS-A related in the resulting GS.

By guaranteeing the two conditions stated above, we can guarantee that other view schemas are unaffected. Also, when deleting a class we have to make sure that the resulting GS is still in a consistent state after the removal, i.e., the resulting GS still satisfies all the schema invariants listed in Section 2. We classify the problems that could be caused by a schema version removal as follows:

P1. Shared-class problem: Classes shared by other view schemas should not be deleted.

P2. ISA-hierarchy problem: The DAG of the resulting GS should be consistent and the types and extents of all remaining classes should not be affected to guarantee invariants **SI1** through **SI4**.

P3. Derivation-dependency problem: All the derivation dependencies are still valid according to schema invariants **SI5** and **SI6**.

P4. Undefined-reference problem: No reference is made to a class that is going to be deleted.

A trivial solution for problem **P1** (the shared-class problem) is to keep a reference counter with each class that keeps track of how many view schemas the class participates in. This way, we would be able to tell which classes are being shared and therefore cannot be removed.

The undefined reference problem (**P4**) is a classical problem that occurs when deleting a class that is being referenced in some form (by using its methods, as domain, etc.) by some other class in the schema. In the SVR context, a class will only be deleted when this class is not used in any view schema (i.e., no user has access to this class). Like other view systems [3], MultiView assumes closed view schemas [19]. The closure criterion ensures that all classes that are being used by the type interface of any class in a view schema are also defined within the view. With this assumption, if a class A is not defined in any view schema, that means that no other class in any view schema references class A (otherwise class A would be in the view schema as well). Given this closed view assumption, we will thus never have an undefined reference after deleting a non-shared class.

Here we review our solutions to **P2** and **P3**, which represent basis for solving the more difficult multiple-class removed problem, which is the focus of this paper. In the remaining sections, we provide solutions to these two problems. As mentioned before, the problem of removing a view schema can be reduced to the problem of removing a particular class (base or derived) from the underlying global schema. Therefore, in the following discussions, we analyze the removal of a particular class, regardless of which view schema this class participates in.

3.2 Solution to the ISA-Hierarchy Problem

The ISA-hierarchy problem deals with the invariants **SI1** through **SI4**. When removing a class from the global schema, we have to make sure none of the other classes' type and extent is changed and that the global schema is still connected. Each class in the schema must maintain its set of super- and subclasses (except for the one being deleted) that it had before the deletion to assure the maintenance of its properties (both inherited or locally defined ones). To guarantee that the set of superclasses will be maintained, after we remove a given class, we have to make sure its subclasses are connected with its superclasses. This is done by adding an IS-A edge between each direct-subclass and each direct-superclass that is not indirectly connected through any other class (this is important or else we would violate **SI2**).

Guaranteeing that the type will be unaffected is more complex (we will refer to this as the *type-effect* problem). By **SI3,** the type is defined by its local properties and its direct-superclasses' type. Therefore, when removing a class from the schema we have to check first whether the class has some local properties that are being used to define its subclasses' type. One key observation to help us address the *type-effect* problem is given next.

Theorem 1 *If there exists a class in the global schema that has its type affected by the deletion of another class from the schema, this class is a subclass (not necessarily a direct-subclass) of the one being removed, and there exists at least one direct-subclass of the class we are deleting that has its type affected. Furthermore, if none of the direct-subclasses has its type affected, then no class in the schema has its type affected.*

The proof of Theorem 1 follows directly from **SI3** and **SI4**. We omit it here, due to space constraints, but refer the reader to [8]. By Theorem 1, to determine whether there will be classes that have their type affected, we simply need to check the direct-subclasses. In our earlier work [9], we identified three different situations that can happen due to the ISA-hierarchy problem when removing a class. This leads us to the following simple rule.

Strategy 1 *If the class C has more than one direct-subclass it cannot be deleted unless C has no locally defined properties. In case C has only one direct-subclass, say D, and a deletion occurs, the existing local properties of C should migrate to D (in order to maintain the type of D the same as before). If the class has no subclasses or it has no locally defined properties, it can be deleted without affecting subclasses. In case C is deleted, its direct-subclasses, if any, should still be subclasses of the superclasses of the original class C. For that we add IS-A edges when necessary.*

Going back to our university database example in Figure 3, the delete of the *Person* class must be rejected, since the property *nationality* is inherited by both *TeachingStaff* and *Student2* classes. On the other hand, the delete of the *Student2* class would not cause the ISA-hierarchy problem, since *Student2* does not have locally defined properties. Similarly, if *Student3* class, instead of *Sudent2*, were to be removed, the removal would not cause ISA-hierarchy problem either. The reason is that although it has a local property *gpa*, it has only one subclass. Therefore, we could remove *Student3*, and migrate the property *gpa* to *Student2*.

3.3 Solution to Derivation-Dependency Problem

The *derivation-dependency problem* is concerned with the issue that the removal of a source class C may cause all its dependent derived classes VC to be undefined – i.e., may leave them without a derivation function to correctly compute their derived extent. Also, any operation dealing with the extent of a derived class (e.g., create instance or update a value) is forwarded to the source class. In this sense, if a source class is deleted without proper care, an error may occur when a derived class forwards operations to be executed by its source class. To solve this problem, we have to redefine the derived classes (including possibly changing their source class) in accordance with schema invariants **SI5** and **SI6**. If the deletion cannot be performed without violating **SI5** and **SI6**, the delete operation must be rejected. We note that, while **SI5** is a very important schema invariant (the derivation chain cannot be cyclic), **SI6** could in principle be relaxed without affecting consistency in general. We inforce **SI6** because this is the way derived classes are defined in our system (see [19]), and the underlying MultiView code assume that a virtual class implements only one object algebra operator.

Strategy 2 *Let VC be a derived class derived from a source class C. Then, the solution we propose is to delete the class C as long as there exists another class C' in the global schema that could become the source for VC, i.e., to allow for redefinition of VC in terms of C'* [4].

The basic problem then is to determine whether and if so how we can redefine a derived class. One example of successfully redefining a derived class is shown in Figure 4. Here the hide class $Student4$ finds a new source in $Student2$ and thus a successful redefinition of $Student4$ can take place when the original source $Student3$ is deleted [5].

Now the problem we need to address is where to look for the new source. Potentially we could investigate the possible suitability of all the classes in the global schema. However, due to the schema invariants, it is possible to significantly limit our search, as stated in Theorem 2.

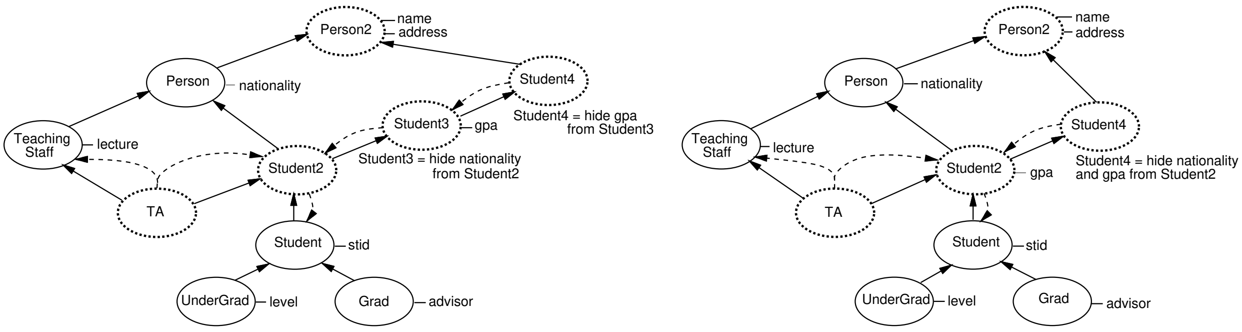
Theorem 2 *If there exists a class C' in the global schema that can be a new source for a derived class VC (in place of C), then there exists at least one (not necessarily direct) super- or subclass of the original source C that can also be a new source. And vice versa, if none of the super- or subclasses C' of C can be a new source for a derived class VC then there exists no class in the global schema that can.*

The proof of Theorem 2 can be found in [8], and is omitted here due to space constraints.

Strategy 3 *The strategy thus is to start looking for a new source C' for the VC redefinition from the direct-subclasses of the original source C and go downwards, and from the direct-superclasses of the original*

[4] A simplifying assumption we make is that a derived class, when being redefined, does not change its VC type. For example, if a VC class is defined as a SELECT class using a select predicate P over class C, then it will remain a SELECT class through possibly over a new class C' and a new select predicate P'

[5] Note that we had already determined that the class *Student3* could be deleted with respect to the type effect problem.



(a) Before Removal of Student3 Class.

(b) After Removal of Student3 Class.

Figure 4: Example - Deleting Class *Student3* and Redefining Class *Student4*.

source C and go upwards the is-a hierarchy. If we cannot find one, we should stop our search, because no other class in the schema will be a new source.

Finally note that, if we were to relax **SI6** (as mentioned before), our approach would not change much, except for the fact that finding a possible source could be easier.

3.4 Basic Algorithm to Remove an Individual Class

Incorporating the strategies to solve the problems pointed out in the previous sections, we now outline an algorithm that removes a single class from the global schema without violating the schema invariants.

Algorithm RemoveClass (*C*)

if *C* is shared **then** return
if *C* has locally-defined properties **and**
  C has more than one direct-subclass **then** return
for each $D \in$ derived-classes(*C*) **do**
    search for new sources for *D* (new-source(*D*))
    if new-source(*D*) $= \emptyset$ **then** return
for each $D \in$ derived-classes(*C*) **do** redefine *D* as a virtual class of a class in new-source(*D*)
if *C* has locally-defined properties **then**
    C' := direct-subclass (*C*)
    migrate properties of *C* to *C'*
for each $SB \in$ direct-subclasses(*C*) **do**
    for each $SP \in$ direct-superclasses(*C*) **do**
        if not (*SB* IS-A* *SP*) **then**
    add-edge *SB* IS-A *SP*
remove class *C*

When the **RemoveClass** algorithm is applied to a consistent schema (with consistency defined by the invariants **SI1** to **SI6** on Section 2), the resulting global schema and view schemas derived from it are all consistent. The first invariant (**SI1**) is not violated because the **root** class being included in all view schemas and therefore being a shared class will not be removed. The **SI4** invariant (single-source principle) could be violated when adding locally defined properties to a class. The only time we do this is when migrating properties from the class we are removing to its subclass (lines 7-9). However, the migration is done only if the class we are removing has *only one* direct-subclass (test at line 2). Therefore **SI4** is not violated. The IS-A hierarchy is still acyclic and connected (**SI2**) due to steps at lines 10 through 13. The step at line 11 guarantees that the subclasses are still connected to *C*'s superclasses. Since we are adding an IS-A edge from *SB* to *SP* and *SB* was already a subclass (indirectly) of *SP*, and the original schema was not acyclic, this edge cannot cause a cycle. **SI5** and **SI6** are not violated because of the way the redefinition is done. The detailed proof is omitted due to space constraints.

3.5 Interdependencies Problems of Multiple Class Removal

Above, we presented an approach used to delete a *single* class from the schema. Care is taken so the final schema will be consistent, and if the class cannot be deleted without violating consistency, the deletion is not performed. The assumption there was to attempt to delete a class assuming that all other classes in the schema would remain in the global schema. In this section, we now identify problems of multiple class removal, since in our system, we want to delete *all (or as many as possible)* classes from the to-be-removed view schema *VS* rather than just one. One simple solution is to just look at each class separately, and decide if it could be deleted given the state of the global schema at the time of deletion. The algorithm can be defined as: "**for** each class $V \in$ View Schema VS **do** RemoveClass (GS-class(V))", with the RemoveClass() function defined in the previous section.

The main drawback of this solution is that the configuration of the resulting global schema is dependent on the sequence in which the classes are examined. For example, it might happen that after you remove a class, you might not be able to remove another class that you could have removed before. This type of conflict happens in the derivation-dependency as well as in the type-effect problem. We show this with the following example.

Suppose we detected that classes *Student2* and *Student3* in Figure 3(b) are not being used by any view schema and therefore can be removed.

Conflict 1: type-effect problem: When we analyze the two classes *Student2* and *Student3* separately, both of them satisfy the criteria for deletion without causing the *type-effect problem*. Namely, *Student2* has no locally defined properties, so its deletion will not change the type of its subclasses, and *Student3* has locally defined properties but has only one subclass, so its properties can be migrated. However, if we delete *Student2*, its direct-subclasses *TA* and *Student* will become direct-subclasses of *Student3*. Therefore, when we try to remove *Student3* next, it does not satisfy the criteria for deletion anymore, because it now has locally defined properties and two direct-subclasses. Conversely, if we delete *Student3* first, its property gpa gets migrated to *Student2* class. Consequently, when we try to delete *Student2* afterwards, it has locally defined properties and two direct-subclasses, so it can no longer be deleted.

Conflict 2: derivation-dependency problem: Removing *Student2* by itself causes no problem since *TA* can be redefined from *Student3*. Removing *Student3* by itself also does not cause any problem since *Student4* can be redefined from *Student2* or *Student*. However, if we want to remove both *Student2* and *Student3*, there is no class from which we could redefine the class *TA*. This is due to the fact that if we redefine *TA* from *Student*, *TA* will have an extra property, *stid*, and if we redefine *TA* from *Student4*, *TA* will not inherit the *gpa* property. Therefore, removal of one of these two classes implies that the other cannot be removed (unless *TA* is also selected for removal).

4 The Dependency Model

To overcome the difficulties presented in the previous section of the removal ordering, we first must detect all possible conflicts between deletions of classes. For this purpose, we have developed a formal model based on Boolean logic, called the Dependency Model that characterizes all removal dependencies between all classes in the global schema. Each dependency between the deletion of classes is represented by a clause, where a clause consists of literals, negated or not, and the conjunction and disjunction operators ($\wedge, \vee$ respectively). All dependency classes must be satisfied to ensure the consistency of the global schema. The basic idea of the dependency model is to associate with each class C_i in the global schema a variable D_i that indicates whether the class C_i will be deleted or not. A clause that corresponds to the example illustrated in Section 3.5 (**Conflict 1** and **Conflict 2**) is: $(\overline{D_{Student2}} \vee \overline{D_{Student3}})$, where $D_{Student2}$ is 1 if $Student2$ is deleted and 0 otherwise, and $D_{Student3}$ is 1 if $Student3$ is deleted and 0 otherwise [6]. In order to satisfy this clause, we need either $D_{Student2}$ or $D_{Student3}$ or both to be zero. But we cannot have both equal to 1, or equivalently, we cannot delete both classes. We represent each such dependency of classes in GS with a clause. By making sure that all clauses are satisfied simultaneously, we are guaranteed to have a consistent final global schema.

The advantage of having the dependencies encoded is that we can analyze the impact of removing classes in the final schema without actually modifying it. We can experiment with different deletion patterns and check whether all consistency requirements are met for a given scenario. One possible approach would be to explore all possible configurations of the final schema and decide among them which one is the best (given some cost function that would minimize the cost of our schema). This approach is exponential in the size of the schema, and hence later on we show how we can improve upon this potentially inefficient performance.

Since we want to "experiment" with different possible deletion patterns, by setting a variable $D_i = 1$, our model indicates the effects that this deletion would have on the remaining schema. Therefore, in the

[6] $\overline{D_i} = 1$ iff $D_i = 0$ and $\overline{D_i} = 0$ iff $D_i = 1$.

process of “experimenting” with one deletion pattern, we mark classes as:

- *non-deletable:* if C_i cannot be deleted without violating the consistency requirements, or if C_i is shared. In this case, its corresponding variable D_i is set to 0.
- *deletable:* if C_i can be deleted without violating the consistency requirements and it is not shared. In this case, its corresponding variable D_i is set to 1.

We now explain the types of dependencies between classes captured by the dependency model.

4.1 The *Type-effect* Problem

Recall from Section 3, that a class cannot be deleted if it has more than one subclass *and* it has locally defined properties. As we have shown in Section 3.5, a conflict might arise between a class to be removed and its direct-superclasses (case of $Student2$ conflicting with $Student3$) or its direct-subclasses (case of $Student3$ conflicting with $Student2$). Furthermore, as classes are removed from the schema, the ISA-hierarchy changes and therefore the dependencies between classes may change. So a class might conflict with other subclasses that were not its direct-subclasses in the original schema. As we want to detect whether the deletion of a class will affect the deletion of other classes in the schema, we now establish the following observation.

Observation. A class C can be deleted *without* affecting the deletion of other classes *regarding the type-effect problem* if the class satisfies one of the following conditions:

• **Condition C1:** *The class C will never have more than one direct-subclass* (even after a sequence of several other deletes). If it is possible that the class C has more than one direct-subclass, its deletion may cause the non-deletion of a direct-superclass (as in the case where the deletion of $Student2$ causes the non-deletion of $Student3$). If the class will always have at most one direct-subclass no matter what classes are removed from the schema, it will not affect its superclasses, SP_i, or subclasses, SB_i.

Proof: (a) Will not affect SP_i: when class C is deleted, it will link at most one subclass to the superclass SP_i. So the number of direct-subclasses of the superclass SP_i will not increase. Suppose, by contradiction, that we were able to delete SP_i before the deletion of C, and that the deletion of C caused SP_i to be non-deletable. If SP_i is non-deletable due to the *type-effect* problem, it is because it has locally defined properties and it has more than one direct-subclass. But if it has more than one direct-subclass, it already had more than one direct-subclass *before* the deletion of C, and therefore it could not have been deletable before the deletion of C. So, either SP_i had more than one direct-subclass *before* the deletion of C, and therefore was non-deletable before the deletion of C, or SP_i is still deletable. □

(b) Will not affect SB_i: suppose, by contradiction, that we were able to delete SB_i before the deletion of C, and that the deletion of C caused SB_i to be non-deletable. If SB_i is non-deletable due to the *type-effect* problem, it is because it has locally defined properties and it has more than one direct-subclass. However, if it has more than one direct-subclass, considering that SB_i is the only direct-subclass of C (since by hypothesis C will never have more than one direct-subclass), if SB_i is removed, C will have as direct-subclasses SB_i's direct-subclasses which are more than one. But that violates our initial hypothesis that C will never have more than one direct-subclass no matter what classes are removed from the schema. So, either SB_i was non-deletable before the deletion of C, or SB_i has only one direct-subclass, and therefore it is still deletable. □

• **Condition C2:** *The class C will never have locally defined properties.* If it is possible that the class C has locally defined properties, its deletion may cause the non-deletion of a direct-subclass (as shown in Section 3.5 when deletion of $Student3$ causes the non-deletion of $Student2$). If the class C will never have locally defined properties no matter what classes are removed from the schema, it will not affect its superclasses, SP_i or subclasses, SB_i.

Proof: (a) Will not affect SP_i: suppose, by contradiction, that we were able to delete SP_i before the deletion of C, and that the deletion of C caused SP_i to be non-deletable. If SP_i is non-deletable due to the *type-effect* problem, it is because it has locally defined properties and it has more than one direct-subclass. However, if it has locally defined properties and it was deletable before the deletion of C, its local properties will be migrated to C. This violates our initial hypothesis that C will never have locally defined properties no matter what classes are removed from schema. So, either SP_i was non-deletable before the deletion of C, or SP_i has no locally defined properties, and therefore it is still deletable. □

(b) Will not affect SB_i: if C will never have locally defined properties no matter what classes are removed from the schema, it cannot cause any conflict with a direct-subclass, SB_i. Suppose, by contradiction, that we were able to delete SB_i before the deletion of C, and that the deletion of C caused SB_i to be non-deletable. If SB_i is non-deletable due to the *type-effect* problem, it is because it has locally defined properties and it has more than one direct-subclass. However, if it has more than one direct-subclass, and it was deletable before the deletion of C, it could not have had locally defined properties. That means that properties were migrated from C. However, this violates our initial hypothesis that C will never have locally defined properties no matter what classes are removed from schema. So, either SB_i was non-deletable before the deletion of C, or SB_i is still deletable. □

• **Condition C3:** *The class C will always have only one direct-subclass and this direct-subclass has locally defined properties.* If the class C has only SB_i as its direct-subclass and SB_i has locally defined properties, SB_i can be deleted provided it has only one direct-subclass to migrate the properties to. So, in case SB_i is deleted, SB_i's direct-subclass will be the only direct-subclass of C and will also have locally defined properties (because SB_i's local properties were migrated to it). Therefore the same reasoning will apply. As a consequence, class C will never have more than one direct-subclass, and this is a special case of **Condition C1**.

In order to capture the *type-effect* dependency, we associate with each class C_i the Boolean variables [7]:

- D_i indicates the label of the class C_i: 1, if it is marked *deletable*; 0, if it is marked *non-deletable*.
- LP_i indicates whether C_i has locally defined properties. Its value is 1 if C_i has locally defined properties, and 0 otherwise.
- NP_i indicates whether C_i will ever have properties locally defined. $NP_i = 1$ if C_i will never have properties locally defined no matter what combinations of deletions take place, and $NP_i = 0$ otherwise (note that if $LP_i = 1$, then $NP_i = 0$).
- OS_i indicates whether C_i will ever have more than one direct-subclass. $OS_i = 1$ if C_i will always have at most one direct-subclass, no matter what combination of deletions takes place, and $OS_i = 0$ otherwise.
- ST_i indicates whether all the classes in the subtree rooted by C_i are marked *deletable* ($ST_i = 1$), or not ($ST_i = 0$).

We can guarantee **C1** (and **C3** which is a special case) for class C_i if $OS_i = 1$. Now, $OS_i = 1$ if and only if C_i has no subclass, or C_i has only one subclass C_j and ($OS_j = 1$ or $D_j = 0$ or $LP_j = 1$). We can guarantee **C2** for class C_i if C_i has no locally defined properties and no properties will be migrated to C_i. We can guarantee the last part if $\forall C_j$ direct-superclass of C_i, either C_j is non-deletable ($D_j = 0$) or $NP_j = 1$.

Note that, if $OS_i = 0$ and $NP_i = 0$ we cannot delete the class C_i without affecting the deletion of other classes (regarding the *type-effect problem*), and we might not even be able to delete C_i. We express this requirement in our model as given below.

Requirement R1: $\overline{OS_i} \wedge \overline{NP_i} \Rightarrow \overline{D_i}$, or equivalently, $(OS_i \vee NP_i \vee \overline{D_i})$, where:

- $OS_i = (ST_{i_1} \wedge ST_{i_2} \wedge \cdots \wedge ST_{i_n}) \vee ((OS_{i_1} \vee \overline{D_{i_1}} \vee LP_{i_1}) \wedge ST_{i_2} \wedge \cdots \wedge ST_{i_n})$
  $\vee (ST_{i_1} \wedge (OS_{i_2} \vee \overline{D_{i_2}} \vee LP_{i_2}) \wedge \cdots \wedge ST_{i_n}) \vee \cdots \vee (ST_{i_1} \wedge ST_{i_2} \wedge \cdots \wedge (OS_{i_n} \vee \overline{D_{i_n}} \vee LP_{i_n}))$
- $NP_i = \overline{LP_i} \wedge (\overline{D_{i_{n+1}}} \vee NP_{i_{n+1}}) \wedge (\overline{D_{i_{n+2}}} \vee NP_{i_{n+2}}) \wedge \cdots \wedge (\overline{D_{i_{n+m}}} \vee NP_{i_{n+m}})$
- $ST_{i_j} = (D_{i_j} \wedge ST_{i_{j_1}} \wedge ST_{i_{j_2}} \wedge \cdots \wedge ST_{i_{j_l}}), \forall j \in \{1..n\}$

where $C_{i_1} \ldots C_{i_n}$ are the direct-subclasses of C_i; $C_{i_{n+1}} \ldots C_{i_{n+m}}$ are the direct-superclasses of C_i; and $C_{i_{j_1}} \ldots C_{i_{j_l}}$ are the direct-subclasses of C_{i_j}.

[7] Note that the D's are the only true decision variables. The LP's are just characteristics of a class, and the NP's, OS's and ST's are computed from D's and LP's.

4.2 The *Derivation-dependency* Problem

Recall that if a derived class is not deleted, we are required to keep in the final schema graph a class that can be a source for this derived class. Let C_i be the derived class, C_{i_1} be the old source, and $C_{i_2}, ..., C_{i_s}$ be the alternative sources, i.e., other classes in the schema from which we could redefine C_i. We express this requirement in our model as follows:

Requirement R2: $(\overline{D_i} \Rightarrow \overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_n}})$, or equivalently, $(D_i \vee \overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_s}})$.

4.3 Discussion

Therefore, in order to represent all consistency requirements, we have a requirement of type **R1** for each class in the schema and a requirement of type **R2** for each derived class in the schema. A valid schema would be one that satisfies all the clauses. We know that such an assignment exists, namely, the one that forces all the classes to remain in the schema, because the schema is already in a consistent state. Notice that if every class is non-deletable, i.e. $D_i = 0, \forall i$, all clauses are satisfied, since there is a negated D_i variable in every clause. This assignment is obviously not the best one, since we would not be optimizing the schema. Another alternative is to try all possible assignments and verify whether all clauses are satisfied (exponential in the size of the schema). This is not optimal in terms of computational effort, since we know that some assignments are impossible, and therefore we do not have to take them into account.

Consider the example illustrated in Section 3.5. We know that we cannot remove *Student2* and *Student3*, so we do not have to consider the assignment $D_{Student2} = 1$ and $D_{Student3} = 1$. Our approach is then to assign one variable at a time and to substitute the value of this variable in all clauses. If we had, for example, a clause that expressed the dependency between $Student2$ and $Student3$, namely: $(\overline{D_{Student2}} \vee \overline{D_{Student3}})$, as soon as we decided to assign 1 to $D_{Student2}$, by substituting its value in the clause, we have: $(\overline{1} \vee \overline{D_{Student3}})$ = $(0 \vee \overline{D_{Student3}})$ = $(\overline{D_{Student3}})$. Therefore, the only assignment that would satisfy this clause is $D_{Student3} = 0$, and so we do not have to consider $D_{Student3} = 1$.

5 The Dependency Graph Approach: DG Generation, Transformation, and Reduction

We have developed and implemented a hyper-graph representation of this formal model, called the dependency graph (DG). This allows us to represent the dependencies between classes using different types of dependency edges, and incrementally decide which classes to delete, while recording the effect on the other classes of the schema. We have found this graph representation, while equivalent to the formal model, to be more suitable for implementation in our SVR tool.

5.1 The Dependency Graph Model

In our SVR system, we thus solve the search order problems outlined in Section 3.5 by first explicitly encoding all the information from the type-effect and derivation-dependency constraints in a compact representation called the dependency graph - DG. The DG is then analyzed to determine the best classes to delete. This approach allows different search strategies to be applied to maximize the number of classes that can be deleted (or any other such cost function). The final structure of DG is not dependent on the order that classes are visited because we are not changing the global schema in the process of building DG.

The dependency graph model is a hyper-graph $DG = (N, E)$, with N the set of nodes and E the set of dependency links. Our DG represents all constraints that must be satisfied in order to guarantee consistency of the schema, i.e., all clauses developed to describe the dependencies based on the formal model. Each node n in DG corresponds to a class C that we wish to delete (here we use the terms *node* and *class* interchangeably). We represent each consistency requirement between classes in a compact form using a link $e \in E$ between a class (*origin class*) and sets of classes (*destination set*); with the consistency requirement encoded in the link type. Each link is denoted as *link-type*$(C_1, \{C_2, C_3, \ldots, C_n\}, \{C_{n+1}, C_{n+2}, \ldots, C_{n+m}\}))$ where $C_i \in DG$ corresponds to the class $C_i \in GS$. The second *destination set* is empty in most types of links, and we omit it when this is the case.

We examine all possible dependencies, and we associate a name with each one of them, which identifies the link type in our dependency graph. Let us start by describing the dependencies that relate to the *type-effect* problem.

- The first dependency is the one described in Section 4 (requirement **R1**): $(OS_i \vee NP_i \vee \overline{D_i})$, where OS_i depends on the subclasses and NP_i depends on the superclasses. Therefore we represent this dependency by a link: $OSorNP(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$ [8]. This link states that the class C_i can be deleted if either OS_i or NP_i are 1, where OS_i and NP_i are computed by the formula given in Section 4.
- The link $OSorNP$ states that class C_i can be deleted if either OS_i or NP_i is 1. Hence if we determine that $NP_i = 0$, we need $OS_i = 1$ in order for the class C_i to be deletable. Therefore the consistency requirement is reduced down to: $(\overline{D_i} \vee OS_i)$. We represent this dependency by a link: $OSonly((C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$. This link states that class C_i can be deleted if $OS_i = 1$, where OS_i is computed by the formula given in Section 4.
- The link $OSorNP$ states that class C_i can be deleted if either OS_i or NP_i is 1. If we know that $OS_i = 0$, we need $NP_i = 1$ in order for the class C_i to be deletable. Therefore the consistency requirement is reduced down to: $(\overline{D_i} \vee NP_i)$. We represent this dependency by a link: $NPonly((C_i, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$. This link states that class C_i can be deleted if $NP_i = 1$, where NP_i is computed by the formula given in Section 4.

[8] where $C_{i_1} \ldots C_{i_n}$ are the direct-subclasses of C_i; $C_{i_{n+1}} \ldots C_{i_{n+m}}$ are the direct-superclasses of C_i.

The first two links ($OSorNP$ and $OSonly$) are both satisfied if $OS_i = 1$. Recall that OS_i is 1 if either the class has no subclass or if it will have only one subclass. If during our deletion analysis process, we determine that one of C_i's subclasses will not be deleted, we know that OS_i will be 1 only if all other subtrees are deleted. Therefore we might come across a dependency such as: a class can be deleted only if a set of subtrees is also deleted. We then have two other dependencies related to the *type-effect* problem:

- The link $OSorNP$ states that class C_i can be deleted if either OS_i or NP_i is 1. If we determine that one subclass C_k of C_i will not be deleted (D_k is 0), we need all subtrees corresponding to the other subclasses of C_i to be deleted in order for OS_i to be 1. Therefore the consistency requirement is reduced down to: $(\overline{D_i} \vee (ST_{i_1} \wedge ST_{i_2} \wedge \cdots \wedge ST_{i_n}) \vee NP_i)$. We represent this dependency by a link: $STorNP((C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$.
- The link $OSonly$ states that class C_i can be deleted if $OS_i = 1$. If at some point we determine that one subclass C_k will not be deleted (D_k is 0), we need all subtrees, correspondent to the other subclasses, to be deleted in order for OS_i to be 1. Therefore the consistency requirement is reduced down to: $(\overline{D_i} \vee (ST_{i_1} \wedge ST_{i_2} \wedge \cdots \wedge ST_{i_n}))$. We represent this dependency by a link: $STonly((C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$.

The above links cover all the possible dependencies that relate to the *type-effect* problem. Let us now examine the dependencies that refer to the *derivation-dependency* problem.

- The first dependency relating to the *derivation-dependency* problem is the one described in Section 4 (requirement **R2**): $(D_i \vee \overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_n}})$, where C_i is the derived class, C_{i_1} is the old source, and $C_{i_2}, ..., C_{i_s}$ are the alternate sources. We represent this dependency by a link: $remainPropagate(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_s}\})$. This link states that if class C_i remains in the schema, at least one of the classes in $\{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}$ should also remain.
- The link $remainPropagate$ states that if class C_i were to remain in the schema, at least one of the source classes should also remain. If we know that class C_i will remain (i.e., $D_i = 0$), then one of the sources should also (i.e., $\overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_s}} = 1$). Therefore the requirement is: $(\overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_s}})$. We represent this dependency by a link: $minimalRemaining(C_{i_1}, \{C_{i_2}, \ldots, C_{i_n}\})$. This link states that, to guarantee consistency, at a minimum at least one class among those in $\{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}$ should remain (i.e., should be non-deletable).

In summary, there are seven types of links, each representing a type of dependency between classes. Two of them relate to the *derivation-dependency* problem, and the other five to the *type-effect* problem.

5.2 The DG Generation Rules

As explained in Section 4, in order to represent all consistency requirements, we must have a requirement of type **R1** for each class in the schema and a requirement of type **R2** for each derived class in the schema.

Therefore, we generate DG by visiting each class in the schema and adding the corresponding links. The algorithm is as follows:

Algorithm GenerateDG (*GS*)

for each $C_i \in GS$ **do**
  add a node C_i to DG
for each $C_i \in GS$ **do**
  if C_i has subclasses
  let $\{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}$ be the direct-subclasses of C_i
    if C_i has local properties
      add link $OSonly$ to DG: $OSonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$
    else
      add link $OSorNP$ to DG:
      $OSorNP(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$
      where $\{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\}$ are the direct-superclasses of C_i
  if C_i is a derived class
    detect all classes, denoted as $\{C_{s_1}, C_{s_2}, \ldots, C_{s_s}\}$, in GS that can be a source class for C_i
    add link $remainPropagate$ to DG: $remainPropagate(C_i, \{C_{s_1}, C_{s_2}, \ldots, C_{s_s}\})$

Note that from the seven possible links, only three are generated at first. This is because these three links make no assumptions with regards to the value of the D variable of other classes, i.e., they do not depend on any knowledge about whether other classes will be deleted or not. This is the case at the beginning of the algorithm, at generation time, when we assume that all classes can "potentially" be deleted, but we have not yet determined which ones will actually be deleted, and which ones will not be deleted. The other four link types are generated when some classes are known to be non-deletable. Therefore, they appear later in the algorithm, as it will be discussed in Section 5.4.

5.3 Preprocessing Phase: Reducing the Dependency Graph

Once we have all the dependencies between classes in the schema encoded in DG, we need to determine a safe way to delete classes that will not violate the consistency of the schema, i.e., that will not violate any of the requirements represented in DG. Equivalently, in the formal model corresponding to DG, this means that we want to find a 'good' assignment for the variables D_i in such a way that all of our clauses are satisfied. As discussed in Section 4, one approach could be to try every possible assignment and verify if the consistency requirements are met. This approach is exponential in the size of the schema and therefore not practical. We can improve on this approach by noticing that some classes will not be deleted regardless of the dependency links in the graph, namely the shared classes. Similarly, some classes in our schema might not depend on the deletion or non-deletion of any other class, nor cause the non-deletion of any class, therefore we should be able to delete them regardless of the deletion of other classes. This process will reduce the size of our dependency graph and consequently the combinations of deletions and non-deletions that we have to consider later on.

We proceed with this DG reduction process by first analyzing the classes and determining which ones will definitely be deleted, which ones will definitely not, and how they affect the other classes in the schema. Initially, all classes are *unmarked*. After some investigation, a class may become *marked*. In case the class C_i is *marked*, its label may be:

- *non-deletable:* if C_i cannot be deleted without violating the consistency requirements, or if C_i is shared.
- *deletable:* if C_i can be deleted without violating the consistency requirements and it is not shared.

We determine the label of a class based on the global schema structure and consistency requirements, and based on the label of the other classes in the schema as we will describe below. If a class C_i is *unmarked*, then this is equivalent to no value having been assigned to D_i. When we label a class C_i *non-deletable*, we assign the value 0 to D_i, and when we label a class C_i *deletable*, we assign the value 1 to D_i.

The preprocessing phase begins initially with all classes being unmarked. After determining which classes C_i are shared, we can mark them *non-deletable* (i.e., we set D_i to 0). Furthermore, we can determine which classes do not depend on other classes in order to be deleted and do not affect deletion of other classes.

Theorem 3 *Consider a class C_i in a global schema GS. Let CR be the set of consistency requirements of the dependency model of GS. If CR is such that the decision variable D_i corresponding to C_i does not appear in any clause negated, and there exists an assignment A_1 of decision variables $D_j, \forall j$ such that $C_j \in GS$, that satisfies all clauses in CR and $D_i = 0$, then there exists an assignment A_2 of decision variables $D_j, \forall j$ such that $C_j \in GS$, that satisfies all clauses in CR and D_j in assignment A_2 is equal to D_j in assignment A_1, except for D_i, i.e., $D_i = 1$.*

The proof of Theorem 3 can be found in [8], and is omitted here due to space constraints.

According to Theorem 3, if we have an assignment A_1 that satisfies all clauses and $D_i = 0$, we can construct an assignment A_2 where all the variables are the same, except for $D_i = 1$. Deleting a class reduces the cost of the schema, and the schemas GS_1 and GS_2 which would be generated based on the assignments A_1 and A_2 respectively only differ by GS_2 not containing class C_i. Hence, GS_2 is "better" than GS_1. Since A_2 is also satisfied and therefore a valid assignment, it is preferable over A_1. Therefore if we detect a class C_i such that D_i does not appear in any clause negated, we mark C_i *deletable*.

Equivalently, from the definition of the links, for the dependency graph, a variable correspondent to a class does not appear negated in any clause provided it does not appear: anywhere in a $OSorNP$, $OSonly$, $NPonly$ or $minimalRemaining$ link; in the *origin* of a $STonly$ or $remainPropagate$ link; in the *origin* or second *destination set* of a $STorNP$ link. Therefore, in the preprocessing phase, we mark a class C' *deletable* if it does not appear in any of the situations described above.

5.4 The DG Transformation Rules

Now the question "*what happens to the links in DG once we have determined that particular classes will or will not be deleted?*" needs to be addressed. By deciding whether particular classes are deleted or not, we can reduce our set of consistency requirements. This is equivalent to reducing the set of classes for which a decision has to be made, and reducing the number of links, once some of them become redundant. Marking a class *deletable* or *non-deletable* might cause as a side-effect the marking of other classes. We represent this formally with transformation rules based on the labels of classes and existing links. Once we mark a class C_i, we are assigning a value to D_i. In this sense, the clauses of DG may be simplified by plugging into them the value of D_i in the place of the variable name D_i.

We provide transformation rules that reflect the decision about marking a class *non-deletable* or *deletable*. In order to cover all possible transformations, we have to consider what happens to the dependency between classes that is encoded in the link type, when the origin class is marked, or when a class in the destination set is marked. In case the link has two destination sets (e.g., $OSorNP$), we have separate rules for when a class from each destination set becomes marked.

Here we give a small subset of the rules to give the flavor of the type of rules, while the complete set can be found in [8]. [9]

Transformation Rules in case a class becomes *non-deletable*:

1. If $OSorNP(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$ and C_i is *non-deletable*:

   The link states a condition for C_i to be deleted. Since C_i is *non-deletable* the link is not needed. Therefore we remove the link.

2. If $OSorNP(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$ and C_{i_j} is *non-deletable* for some $j \in \{1..n\}$:

   The link states that C_i can be deleted (since it has no local properties, otherwise we would have generated $OSonly$ instead), but it can affect deletion of super and subclasses. If we know that one subclass will remain in the schema, if class C_i has any property migrated to it, it can only be deleted if all other subtrees are completed deleted. Therefore, we substitute the link by:

   $STorNP(C_i, \{C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$

10. If $STorNP(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$ and C_i is *non-deletable*:

[9] The numbering of rules corresponds to the numbering in [8] which includes the complete set of rules. Here we only have a subset of those, therefore, the numbers are not necessarily sequential.

The link states a condition for C_i to be deleted. Since C_i is *non-deletable* the link is not needed. Therefore we remove the link.

15. If $remainPropagate(C_i, \{C_{i_1}, C_{i_2}, \dots C_{i_s}\})$ and C_i is *non-deletable*:

    The link states that if C_i is not deleted (i.e., if C_i remains in the global schema), at least one of the classes $C_{i_1}, \dots, C_{i_s}$ should remain. Therefore, this link is translated into a link that represents this dependency between classes $C_{i_1}, \dots, C_{i_s}$ and does not include class C_i: $minimalRemaining(C_{i_1}, \{C_{i_2}, \dots, C_{i_s}\})$.

16. If $remainPropagate(C_i, \{C_{i_1}, C_{i_2}, \dots, C_{i_s}\})$ and there exists one $j, j \in \{1..s\}$ such that C_{i_s} is *non-deletable*:

    The link states that if C_i is not deleted, at least one of the classes $C_{i_1}, C_{i_2}, \dots, C_{i_s}$ should remain. Since C_{i_j} will remain, the link is not needed. Therefore we remove the link.

Transformation Rules in case a class is *deletable*:

31. If $remainPropagate(C_i, \{C_{i_1}, C_{i_2}, \dots, C_{i_s}\})$ and C_i is *deletable*:

    The link states that if C_i is not deleted, at least one of the classes $\{C_{i_1}, C_{i_2}, \dots, C_{i_s}\}$ should remain. Since C_i is *deletable*, the link is not needed. Therefore we remove link.

33. If $minimalRemaining(C_{i_1}, \{C_{i_2}, \dots, C_{i_s}\})$ and C_{i_1} is *deletable*:

    The link states that if C_{i_1} is deleted, at least one of the classes $\{C_{i_2}, \dots, C_{i_s}\}$ should remain. So, now the dependency is among the classes $\{C_{i_2}, \dots, C_{i_s}\}$. Therefore, this link is translated into another link that represents this dependency between classes $\{C_{i_2}, \dots, C_{i_s}\}$.

    $minimalRemaining(C_{i_2}, \{C_{i_3}, \dots, C_{i_s}\})$.

5.5 The Decision-Making Phase

When there are no conflicts between which classes to remove, we can completely determine during the preprocessing phase which classes to delete and which classes not to delete. However, when any interdependencies between multiple class removal options remain, i.e., several classes remain "unmarked", then a decision must be made about which class to give a higher preference for deletion over others. We note that, if we arbitrarily mark classes as *deletable*, we might be forcing other classes to be marked *non-deletable*. Even worse, it might happen that if we mark a class *deletable* we cannot find an assignment for the other variable that would satisfy all the constraints. Recall that we had the guarantee that at least one assignment would be possible, namely, marking all classes *non-deletable*. Therefore, in the decision-making phase, we can mark classes as *deletable* and still guarantee that there will be an assignment of variables that results in a consistent state, if when marking a class *deletable* (and therefore setting its corresponding D variable to 1), we are still left with a set of clauses that have at least one negated D variable in a disjunction. With that, we guarantee that, by marking all the classes that are left as *non-deletable*, we have the schema in a consistent state.

Equivalently, for the dependency graph, a variable that, if set equal to 1, will cause the remaining clause to have no negated D variable, corresponds to a class appearing in one of the following links: in the *origin* of a $OSonly$ or $STonly$ link; in the *destination set* of a $remainPropagate$, in case it is the only class in the *destination set*.

Therefore, if we want to guarantee that after marking a class *deletable* there will still be a feasible assignment, we should not mark classes that appear in the links given above. This way of marking classes is heuristic. If we have a way of measuring optimality, an issue which will be addressed in Section 8, we could compare two schemas and determine which one is best. In such a situation, we would want to examine all possible schemas to decide on the best one. Note that this approach would be exponential in the size of *unmarked* classes, and not in the size of the global schema. Given that the preprocessing phase would mark at least all the shared classes, the size of the problem that we want to examine is *at most* the size of the schema version we are removing. Also, once we mark one given class, we can apply the transformation rules given in Section 5.4, since marking a class might affect other classes. With that, we also reduce the set of combinations we have to examine.

5.6 The Overall Class Removal Process

Figure 5 depicts a flowchart of the overall process for determining classes to delete. It ties together the interactions among the different tasks outlined earlier in this section. After creating the DG using the DG-Generation rules detailed in Section 5.2, we start the preprocessing phase discussed in Section 5.3, followed by the transformation rules described in Section 5.4. The preprocessing phase finishes when we exit the test "any rule applied" with a NO answer for the first time.

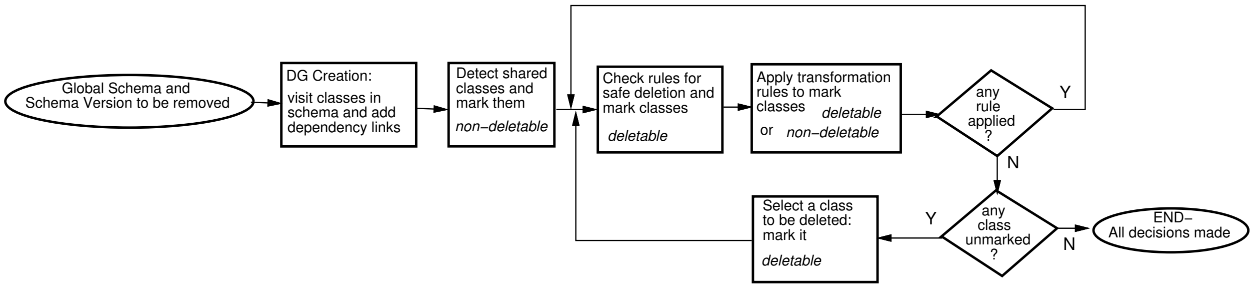


Figure 5: The Overall Process for Determining Classes to Delete.

After the preprocessing phase, we have to make decisions about which classes to mark (See the bottom box of the flowchart in Figure 5 marked "Select a class ..."). In Section 5.5, we presented different strategies of how to select classes to mark. This selection strategy in the simplest case could be to randomly pick a class to delete – which may not result in an optimal solution. As an alternative, one could employ an exhaustive search strategy – namely, to consider all possible combinations of deletions and pick the one with the "smallest cost". It is this second alternative that we propose to employ in our system with the notion of the "smallest cost" of a schema graph as determined by the cost model introduced in the Section 6. We explain the overall process with the following example.

Figure 6 illustrates the overall process with the schema given in Figure 3(b). Suppose the classes *Student2* and *Student3* in Figure 3(b) are no longer used in views and we thus want to remove them. Following the algorithm given in Section 5.2, we generate links of type $OSorNP$ to classes *Student4* and *Student2* and links of type $OSonly$ to classes *IC*, *Person*, *Student3*, *TeachingStaff* and *Student*. Those links go from each class to its direct-subclasses (and direct-superclasses in the case of a $OSorNP$ link). No links relating to the *type-effect* problem are added to classes *TA*, *UnderGrad* and *Grad* since they don't have subclasses. For each of the derived classes we add a $remainPropagate$ link after determining which classes are eligible for being a source class. Figure 6 (a) shows the $remainPropagate$ links generated. We omit the $OSorNP$ and $OSonly$ links for simplicity of the figure.

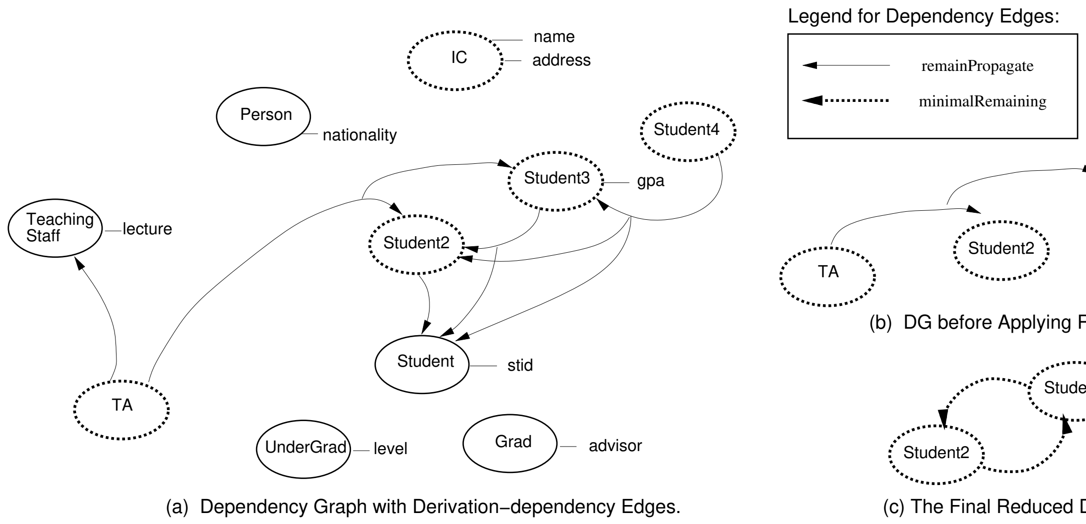


Figure 6: Initial and Reduced Dependency Graph (DG) Representations for the Schema in Figure 3(b).

The preprocessing phase starts by marking every class other than *Student2* and *Student3* as *non-deletable*. Now, with regards to the $remainPropagate$ links, since *Student* and *eachingStaff* are *non-deletable*, by applying rule # 16, we are left with only one link with regard to the derivation dependency problem (see Figure 6 (b)). Lastly, by applying rule # 15, and the fact that TA is *non-deletable*, we are left with the link: $minimalRemaining(Student2, \{Student3\})$.

With regard to using the fact that the classes *IC*, *Person*, *Student4*, *TeachingStaff*, and *Student* are *non-deletable* and rules # 1 and # 4, we are left with the following links regarding *type-effect* (remember that these links were omitted for simplicity of the figure):

- $OSorNP(Student2, \{TA, Student\}, \{Student3, Person\})$
  By using rule # 2 and the fact that *TA* and *Student* are *non-deletable*, we reduce the link to: $NPonly(Student2, \{Student3, Person\})$. Then, by using rule # 3 and the fact that *Person* is *non-deletable*, we reduce the link to: $NPonly(Student2, \{Student3\})$. Finally, by applying rule # 9, since $NP_{Student3} = 0$, we have: $minimalRemaining(Student2, \{Student3\})$.
- $OSonly(Student3, \{Student2\})$
  By using rule # 6 and the fact that $ST_{Student2} = 0, OS_{Student2} = 0, LP_{Student2} = 0$, we are left with: $minimalRemaining(Student3, \{Student2\})$.

Therefore, we get the final reduced graph shown in Figure 6 (c). This DG indicates that removing *Student2* will preclude us from removing *Student3*, and vice versa.

6 Cost Model for Guiding the Class Removal Process

When we encounter scenarios like the one given above, where one class (*Student2*) can be removed only if another class (*Student3*) stays and vice versa, a simple random selection for a deleting class that reduces the number of classes in the global schema is not very helpful (since both resulting schemas would have the same number of classes). Thus, we propose to use a cost model that models the quality of the resulting schema to guide our decision about which removal to execute.

6.1 The Overall Cost Evaluation Process

Let GS be a global schema that consists of classes C = $\{C_1, C_2, \ldots, C_n\}$, and let the resulting GS after removing a class C_i be $GS|C_i$, where i $\in$ [1,n]. When schema version removal is confronted with the interdependency problem, a class is chosen to be removed over another class if it results in a global schema with lower costs. Although the methodology presented in this paper is general enough and could be applied to different view systems, a cost model will definitely be system dependent. In this paper, we concentrate on a materialized view system, namely MultiView, and we consider the *view maintenance cost* of GS, denoted as $COST(GS)$ [10]. In this paper, we consider insert, delete, and change-attribute-value updates on base classes only. Updates on derived classes are delegated by our system to the underlying source classes then propagated back to all the relevant derived classes through the derivation chain. Hence updates on derived classes are omitted here for simplicity.

Let BC(GS) be the base classes in the schema GS, and DC(C_i) be the classes in the derivation hierarchy rooted at C_i. Let $pCost(x, C_i)$ be the propagation cost for an update *x*, where *x* is either insert, delete, or change-attribute-value at a base class C_i. $pCost(x, C_i)$ is equal to the update cost at the base class C_i plus

[10] Retrieval costs are negligible, since all derived classes are materialized. We do not consider the costs of restructuring the global schema required to correctly remove a class either, because such a schema version removal operation is a one-time cost that potentially could be done during non-peak hours or even off-line.

the summation of the update costs at the derived classes rooted at C_i. Given that NI_i, ND_i, and $NC_{a,i}$ are the number of insert, delete, and change-attribute-value updates issued to C_i, we have:

$$COST_{GS} = \sum_{C_i \in BC(GS)} \{ (pCost(insert, C_i) * NI_i) + (pCost(delete, C_i) * ND_i) + \sum_{a:\ attribute\ of\ C_i} (pCost(change, a, C_i) * NC_{a,i}) \}. \quad (1)$$

Note that one type of update may cause the same type and/or other types of updates to be triggered for its derived classes. For example, inserting an object into the first source class of a Difference class may cause the object to be inserted into the Difference class, if the object is not already in the second source class. On the other hand, inserting an object into the second source class of a Difference class may cause the object to be deleted from the Difference class, if the object is also in the first source class. However, an insert operation will never cause its derived classes to change their attribute values. We summarize these insert propagation effects in Equation 2 below.

$$pCost(insert, C_i) = cost(insert, C_i) + \sum_{C_k \in DC(C_i)} (cost(insert, C_k) + cost(delete, C_k)). \quad (2)$$

Similarly, the delete propagation effects are summarized in Equation 3.

$$pCost(delete, C_i) = cost(delete, C_i) + \sum_{C_k \in DC(C_i)} (cost(delete, C_k) + cost(insert, C_k)). \quad (3)$$

Note that changing an object's attribute value through a source class of a Select class may cause all three types of update operations to be triggered for the Select class. It may cause the object to be inserted into the Select class, if the select predicate was evaluated to be false before the attribute value change and to be true after the change. It may cause the object to be deleted from the Select class, if the select predicate was evaluated to be true before the attribute value change and to be false after the change. Finally, it may simply cause the attribute value to be changed also in the Select class, if the select predicate is evaluated to be true both before and after the attribute value change. If the select predicate is evaluated to be false both before and after the attribute value change, then it does not affect the Select class at all. These effects are summarized in Equation 4.

$$pCost(change, a, C_i) = cost(change, a, C_i) + \sum_{C_k \in DC(C_i)} \{cost(insert, C_k) + cost(delete, C_k) + cost(change, a, C_k)\}. \quad (4)$$

An individual insert, delete, or change-attribute-value operation at a given class C_k (without considering the propagation costs) is:

1. $cost(insert, C_k) = PI_k * cost_{insert}$
2. $cost(delete, C_k) = PD_k * cost_{delete}$
3. $cost(change, a, C_k) = PC_k(a) * cost_{change-attr}(a)$

where PI_k is the probability of adding a new type of the class C_k to an existing object, PD_k is the probability of deleting the class type C_k from an existing object, $PC_k(a)$ is the probability of changing the value of the attribute a of an object in the class C_k, and $cost_{insert}$ represents the costs to insert a new object into a class, $cost_{delete}$ the costs to delete an object from a class, and $cost_{change-attr}(a)$ the costs to change the value of an attribute a of an object. These parameters, being implementation dependent, are assumed to be given. To compute these parameters for the derived classes with two source classes C_i and C_j, i.e., for derived classes with types *union*, *difference*, or *intersection*, we also need to know the conditional probability of an object belonging to C_j given that the object belongs to C_i, denoted as $PM_{j|i}$.

6.2 Determination of Propagation Probabilities for Virtual Classes

In the remaining of this section, we show how to compute $PI_k, PD_k, PC_k(a)$, and $PM_{j|k}$ for each type of derived class, assuming the same set of parameters is given for each base class in GS. These parameters are derived iteratively through the derivation chain(s), while updates are performed on its rooted base class(es). The table in Figure 7 summarizes the propagation probabilities; with each row capturing one derived class type C_k and its four associated parameters $PI_k, PD_k, PC_k(a)$, and $PM_{j|k}$. If there are two source classes for a derived class, denoted by C_{i_1} and C_{i_2}, we assume for simplicity that only one of the source classes is operated upon by an insert/delete/change-attribute-value at each time.

VC type	Insert PI_k	Delete PD_k	Change-attribute $PC_k(a)$	Membership $PM_{j\|k}$
Hide	= PI_i	= PD_i	= $PC_i(a)$	= $PM_{j\|i}$
Refine	= PI_i	= PD_i	= $PC_i(a)$	= $PM_{j\|i}$
Select	= $PI_i * Sel_k +$ $(1 - Sel_k) * Sel_k * PC_i(a)$	= $PD_i * Sel_k +$ $(1 - Sel_k) * Sel_k * PC_i(a)$	= $PC_i(a) * Sel_k * Sel_k$	$\approx PM_{j\|i} * Sel_k$ or = 0 or 1
Union	= $PI_{i_1} * (1 - PM_{i_2\|i_1})$	= $PD_{i_1} * (1 - PM_{i_2\|i_1})$	= $PC_{i_1}(a)$	$\approx PM_{j\|i_1} + PM_{j\|i_2} - PM_{i_1\|i_2}$
Intersect	= $PI_{i_1} * PM_{i_2\|i_1}$	= $PD_{i_1} * PM_{i_2\|i_1}$	= $PC_{i_1}(a)$	$\approx PM_{j\|i_1} * PM_{j\|i_2}$
Diff (C_{i_1})	= $PI_{i_1} * (1 - PM_{i_2\|i_1})$	= $PD_{i_1} * (1 - PM_{i_2\|i_1})$	= $PC_{i_1}(a)$	$\approx PM_{j\|i_1} * (1 - PM_{i_2\|i_1})$
Diff (C_{i_2})	= $PD_{i_2} * PM_{i_1\|i_2}$	= $PI_{i_2} * PM_{i_1\|i_2}$	0	$\approx PM_{j\|i_1} * (1 - PM_{i_2\|i_1})$

Figure 7: Propagation Probabilities Parameters for Derived Classes.

For a Hide class (row 1 of the table), for example, the class extent of the Hide class is the same as its (direct) source class, and hence whenever an object is added into (deleted from) its source class, the object is also added into (deleted from) the Hide class. Thus the first row of the table is: $PI_k = PI_i$, $PD_k = PD_i$, $PC_k(a) = PI_i(a)$, and $PM_{j|k} = PM_{j|i}$. Similarly we get the propagation probabilities for the Refine class (row 2 of the table).

In order to compute the parameters for a Select class C_k, we need to know the selectivity (Sel_k) of the select predicate. Note that changing an attribute value of a source class of a Select class may cause all three kinds of updates to happen to the Select class.

1. $PI_k = (1 - Sel_k) * Sel_k * PC_i(a)$, where a is an attribute used in the select predicate
   Explanation: An object of the source class, originally having the select predicate evaluated to false, changes its attribute value, and the select predicate evaluates to true afterwards [11].

[11] When the select predicate is defined upon multiple attributes, changing the attribute value **a** of an object in the source class

2. $PD_k = (1 - Sel_k) * Sel_k * PC_i(a)$, where a is an attribute used in the select predicate
   Explanation: An object of the source class, originally having the select predicate evaluated to true, changes its attribute value, and the select predicate evaluates to false afterwards.

3. $PC_k(a) = PC_i(a) * Sel_k * Sel_k$
   Explanation: The change-attribute-value operation applied to an object needs to be propagated to the Select class only when the object is in the Select class before and after the change-attribute-value operation (with probability Sel_k).

(a) Relationship between C_i and C_j. (b) Case 1: $PM_{j|k} = 1$. (c) Case 2: $PM_{j|k} = 0$. (d) Case 3: $0 < PM_{j|k} < 1$.

Figure 8: Different Cases for Calculating Multiple Membership Probability PM.

There is no straightforward way to calculate the conditional probability PM, i.e., the multiple membership probability, for the Select class. Let's use Figure 8 to illustrate the intrinsic difficulties associated with deriving PM for the Select class. As shown in Figure 8, there are three situations that may occur. Figure 8(a) shows the relationship between C_i and C_j; while the Select class C_k defined on C_i will be represented by a smaller dashed circle inside C_i. We use the shaded area to represent the area that an object belongs to C_k and also belongs to C_j. Figure 8(b) shows the case where $PM_{j|k}$ = 1, i.e., C_k is totally contained in C_j (note $PM_{j|i}$ is less than one here). Figure 8(c) shows that $PM_{j|k}$ = 0, when C_k and C_j do not overlap, although $PM_{j|i}$ is greater than zero. Finally Figure 8(d) shows the situation when C_k and C_j partially overlap, hence $0 < PM_{j|k} < 1$. To simplify our work without knowing the exact relationship between $area_1 = C_i \cap C_j$ and $area_2$ = the location of C_k, we first check whether (1) $area_2$ is contained in $area_1$, (2) $area_1$ and $area_2$ are disjoint, or (3) $area_1$ and $area_2$ partially overlapped. For case 1, we set $PM_{j|k}$ to 1; and for case 2, we set $PM_{j|k}$ to 0. However, if it is case 3, we make the assumption that the locations of $area_1$ and $area_2$ are independent in order to be able to determine a value for $PM_{j|k}$. Given this assumption, we have $PM_{j|k} \approx PM_{j|i} * Sel_k$ for case 3.

Operating upon any of the source classes when the VC type is Union or Intersect gives us the same results of the parameters, because the order of the operands does not affect the resulting derived class. Hence, we show only one of the two symmetric cases in rows 5 and 6. However, updates upon the first or second source class of a Difference class differ, since Difference is not symmetric (row 7). Note there is no straightforward way to calculate the multiple membership probability PM for these derived classes.

and causing the object to be inserted into the Select class happens only when **a** is the only attribute causing the select predicate to evaluate to false.

Similar to the Select class case described above, we make a similar independence assumption for the Union, Intersect, and Difference classes.

7 Experimental Evaluation

7.1 Experimental Setup

In this section, we discuss experiments we have run to evaluate our SVR tool. Goals of this evaluation include to demonstrate that (1) removing obsolete materialized derived classes indeed reduces the overall incremental view maintenance costs, and (2) the cost model introduced in Section 6 guides us to choose the most effective class removal patterns. An initial version of SVR incorporating a simplified dependency graph model [9] was successfully implemented on top of MultiView. While planning on enhancing the implementation of the first SVR prototype tool, we ran simulation experiments on top of the earlier system. Our test cases are built upon the small OO7 benchmark [6], which we have extended with derived classes – as there is currently no available OO benchmarking testbed for object-oriented views [12]. As depicted in Figure 9(a), the $AtomicPart$ and $Document$ classes from the OO7 benchmark contain 10,000 and 500 object instances, respectively. For each experiment, we run the same set of update operations ten times on the global schema and report the average run time. All experiments are conducted on GemStone[12] running on SUN Sparc-10 workstation with a 48MB main memory.

7.2 Experiment One

In this experiment (Figure 9(a)), $AtomicPart$ and $Document$ are the base classes; $APSel1$ is a Select class defined upon $AtomicPart$; $APSel2$ is a Select class defined upon $APSel1$; and $DocSel$ is a Select class defined upon $Document$. The goal of this experiment is to determine whether decreasing the number of derived classes in the global schema indeed decreases the view maintenance costs – as assumed by the overall approach presented in this paper.

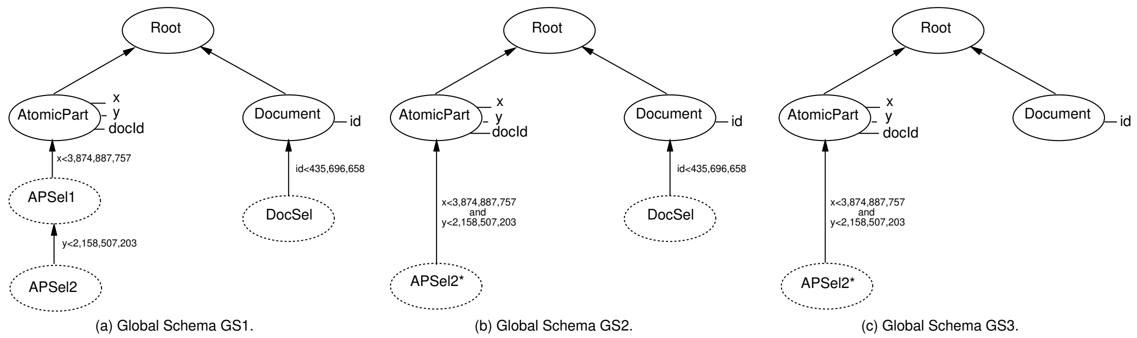


Figure 9: Global Schema Used in Test Case 1.

[12]The GemStone OODB is a registered trademark of GemStone Inc.

First, we ran change-attribute-value updates [13] on 1% of the extent of the $AtomicPart$ and $Document$ classes, respectively, and measured the run time to propagate the update effects to all their derived classes. Assume that $APSel1$ and $DocSel$ have become obsolete over time, and we wish to remove these out-of-date derived classes. Applying the class removal techniques introduced earlier, we conclude that removing these derived classes does not violate any of the consistency constraints and there exists no interdependency problems. Let us assume $APSel1$ is removed first and we run the same set of update operations on the resulting $GS2$ (Figure 9(b)), and measure the view update propagation time for it. Note in Figure 9(b), $APSel2$ is redefined with the name $APSel2*$, when its original source class $APSel1$ is removed from $GS1$, and with $AtomicPart$ as its new source as determined by our derived class redefinition strategy. Next we removed $DocSel$, and again measure the run time for the same set of updates for the resulting $GS3$ (Figure 9(c)).

	GS1	GS2	GS3
Num of total classes	6	5	4
Running time (milliseconds)	22566.7	6799.5	6406.2

Figure 10: The Results of Experiment 1.

Figure 10 shows the results of this test case with run times measured in milliseconds. The results confirm that the view maintenance costs decrease when the total number of classes decreases given the same workload. We have run various experiments with different types of derived classes, such as Hide, Select, and Intersect, all of which have the same findings. These results support our goal of removing as many obsolete materialized derived classes as possible, when their safe removal has been determined by our dependency model.

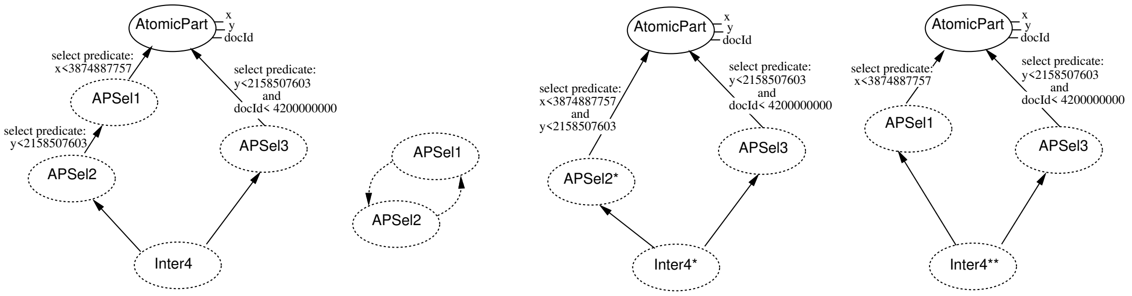
7.3 Experiment Two

This experiment is designed to explore the usage of our cost model. For Figure 11(a), let us assume that $APSel1$ and $APSel2$ have become obsolete over time and we wish to remove them from the global schema, if possible. After consulting the dependency graph DG (Figure 11(b)), we find that there exists a mutual interdependency between the removal of these two classes. We hence use our cost model to guide the class removal.

Let us assume the workload consists of only change-attribute-value operations on 1% of the $AtomicPart$ extent, i.e., the $NC_{a,AtomicPart}$ in Equation (1) is 100 change-attribute-value updates (1% of the 10000 objects), and NI_i and ND_i are both zero. We get:

$$COST_{GS2} = \sum_{\forall a:\ attribute\ of\ AtomicPart} pCost(change, a, AtomicPart) * NC_{a,AtomicPart}$$
$$= \sum_{\forall a:\ attribute\ of\ AtomicPart} \sum_{c \in GS2} \{cost(insert, c) + cost(delete, c) + cost(change, a, c)\}$$
$$COST_{GS3} = \sum_{\forall a:\ attribute\ of\ AtomicPart} pCost(change, a, AtomicPart) * NC_{a,AtomicPart}$$
$$= \sum_{\forall a:\ attribute\ of\ AtomicPart} \sum_{c \in GS3} \{cost(insert, c) + cost(delete, c) + cost(change, a, c)\}$$

[13]Note that in this case the change-attribute-value may also cause objects to be added to (or deleted from) a Select class.



(a) Global Schema GS1. (b) DG of APSel1 and APSel2. (c) Global Schema GS2. (d) Global Schema GS3.

Figure 11: Global Schema Used in Test Case 2.

Comparing the two cost functions listed above, the only difference is that $GS2$ has $APSel2*$ and $Inter4*$ and $GS3$ has $APSel1$ and $Inter4**$ instead (Figure 11(c) and (d)). Now, we compute the change-attribute-value propagation costs for these four classes. We begin with parameters $PI_{AtomicPart} = 0, PD_{AtomicPart} = 0$, $PC_{AtomicPart}(a) = 1, \forall\ a\ an\ attribute\ of\ AtomicPart$, $Sel_{APSel1} = 90\%$, and $Sel_{APSel2*} = 45\%$. Then we compute PI, PD andPC for $APSel1$, $APSel2*$, $Inter4*$, and *Inter*4**. The computed values are:

1. $PI_{APSel1} = 0 * 0.9 + (1 - 0.9) * (0.9) * 1 = 0.09$
2. $PD_{APSel1} = 0 * 0.9 + (1 - 0.9) * (0.9) * 1 = 0.09$
3. $PC_{APSel1}(a) = 1 * 0.9 * 0.9 = 0.81$
4. $PI_{APSel2*} = 0 * 0.45 + (1 - 0.45) * (0.45) * 1 = 0.2475$
5. $PD_{APSel2*} = 0 * 0.45 + (1 - 0.45) * (0.45) * 1 = 0.2475$
6. $PC_{APSel2*}(a) = 1 * 0.45 * 0.45 = 0.2025$
7. $PI_{Inter4*} = 0.2475 * 0.98 = 0.24255$
8. $PD_{Inter4*} = 0.2475 * 0.98 = 0.24255$
9. $PC_{Inter4*}(a) = 0.2025$
10. $PI_{Inter4**} = 0.09 * 0.49 = 0.0441$
11. $PD_{Inter4**} = 0.09 * 0.49 = 0.0441$
12. $PC_{Inter4**}(a) = 0.81$

Assuming that the values of $cost_{insert}, cost_{delete}$ and $cost_{change-attr}(a)$ are the same (say $cost_{access}$), we have: [14]

$$COST_{GS2} = [(0.2475+0.2475+0.2025)+(0.24255+0.24255+0.2025)]*cost_{access} = 1.3851*cost_{access}$$

$$COST_{GS3} = [(0.09 + 0.09 + 0.81) + (0.0441 + 0.0441 + 0.81)] * cost_{access} = 1.8882 * cost_{access}$$

Therefore, if we consider that the values of $cost_{insert}, cost_{delete}$ and $cost_{change-attr}(a)$ [15] are approximately the same, we will have $COST_{GS3} > COST_{GS2}$. Also, if we were to take into account insertions and deletions of objects in the $AtomicPart$ class, $PI_i \neq 0$ and $PD_i \neq 0$ the probability of inserting (deleting) an object in (from) $APSel1$ would be much higher than $APSel2*$, and this would make the difference between $COST_{GS3}$ and $COST_{GS2}$ even larger. Therefore, we choose to delete $APSel1$.

To confirm the validity of the finding achieved by the cost model, we now evaluate the costs on our real system. We first measure the view maintenance costs associated with $GS1$ while running a set of

[14] Note that we just compute part of the cost of the schema, namely, we did not consider operations on classes $AtomicPart$ and $APSel3$ since they belong to both schemas.

[15] At this point, we have not run experiments to give an approximate value for $cost_{insert}, cost_{delete}$ and $cost_{change-attr}(a)$ yet.

Global Schema with Class (Selectivity) Num of Total Classes	GS1 6	GS2 APSel2* (45.32 %) 5	GS3 APSel1 (90.16 %) 5
Running Time (milliseconds)	27023.5	10908.9	16793.4

Figure 12: The Results of Experiment 2.

change-attribute-value updates on the $AtomicPart$ base class (on 1% of the $AtomicPart$ extent). Then we measure the view maintenance costs for the case that $APSel1$ is removed and for the case that $APSel2$ is removed. As illustrated in Figure 12, the run time results show the same trend as the results we get from using our cost model.

8 Conclusions

In this paper, we identify the schema version removal problem which has importance for applications utilizing versioning and view approaches [3, 11, 13, 15]. In this paper, we focus our discussions in the context of MultiView/TSE system. Results of our work improve the efficiency of transparent schema evolution systems such as TSE [18], in the sense that it provides a mechanism for removal of schema versions in systems where the virtual classes are integrated in the class hierarchy [1, 19, 22]. It thus increases their utility as mechanisms for enabling interoperability.

We characterize four potential schema consistency problems for single class removal, and present a solution for each of these problems. Key ideas here are virtual class redefinition strategies to address the derivation-dependency problem and promotion of properties to address the type-effect problem. We demonstrate that view schema removal is sensitive to class ordering. Our solution for this multiple class removal problem is based on a formal model, called the Dependency Model, of capturing all dependencies between class deletions and non-deletions as logic clauses. This model allows us to guarantee the consistency of the resulting schema in the sense that a schema is consistent as long as the chosen variable assignment is valid – where a variable assignment corresponds to a decision for each class in the global schema as to whether it should be deleted or not. Based on this formal model, we have developed and proven consistent a Dependency Graph (DG) representation and associated set of rules for DG generation, reduction, and transformation. Once alternative removal patterns on the Dependency Graph are identified that cannot allow for the removal of a virtual class without preventing the removal of another, we require a strategy for deciding which mutually exclusive selection to make. To address this problem, we have presented a cost model for evaluating alternative removal patterns on the DG and thus for guiding the class removal process.

In this paper, we also present results from the experiments we have conducted on top of the MultiView/TSE system. In the small example in Section 7 (see Figure 10), we improved the efficiency of a sequence of update operations by 70% by removing 2 classes. The simulations also indicated that our cost model provides us with good guidance of which class to remove in order to optimize the schema. We have implemented a preliminary version of the SVR system as proof of concept of some of the ideas outlined in this paper that removes classes and guarantees that the final schema is still consistent.

In the future, a new version of the SVR implementation needs to be developed to incorporate the complete dependency graph with all the transformation rules. Another area that needs future work is with regard to the cost model. Extensive experimental studies would have to be run to get an approximation of the values for $cost_{insert}$, $cost_{delete}$ and $cost_{change-attr}(a)$ for the view model chosen as target of the study. Since these values are implementation dependent, these types of experiments would have to be re-run if our cost model were to be applied to some other view system. With a good estimate for those values, one can then integrate the proposed cost model into the SVR tool.

One capability that could be added to SVR is to allow a deleted class's operator to be combined with another class's operator. These multiple operator classes require relaxing assumption **SI6**. Combining operators will decrease the number of stored classes, and therefore, improvements in the efficiency of our schema are likely to occur, as shown in our preliminary test cases. Adding multiple operators will require changes to the classification system of MultiView [19, 20] so that the new multiple operator classes can be properly placed in the global schema as well as changes in the view maintenance algorithm [13, 14]. The work described in this paper still applies for a system where multiple operator classes are allowed. The only extensions we envision that would need to be done to our solution presented in this paper are with regard to the way new sources are found (the algorithm itself), and the way redefinition is done.

Finally, while the SVR tool was developed for the MultiView/TSE system, the general approach and many of its associated techniques are applicable to other systems where the virtual classes are integrated in the class hierarchy [1, 22]. The degree to which our approach will have to be changed depends on how much the schema invariants differ from the ones listed in Section 2 (the rules for class deletion are directly related to final consistency of the schema, which is dictated by the schema invariants). Here, we briefly discuss how our approach could be applied to two other systems. In the work by Scholl et al. [22], composite queries are allowed (which would invalidate **SI6**). However, intermediate results are built in the process, and therefore each class is implementing only one object algebra, so the schema invariant **SI6** is in fact the same. It is not clear whether schema invariant **SI4** (single point of inheritance) is also enforced, although in the examples shown, that happened to be the case. All other invariants are the same, which allows direct application of our approach. In the view system described by Abiteboul et al. [1], the schema invariant **SI4** is not enforced, so the migration of properties could be different if our approach were to be applied to this system. In this case, some of the deletion rules would change, but the overall approach, i.e., formal dependency model, DG, transformation rules, and the cost model, are still applicable.

Acknowledgements. Thank you to Harumi Kuno and Young-Gook Ra, who built the MultiView and the TSE systems, for providing us with a valuable platform for developing the SVR tool. We also thank them as well as the other members of our University of Michigan Database Group for discussions and suggestions on an early draft of this paper. Finally, we would like to thank the journal referees for their comments and suggestions which helped improve the quality of this paper.

References

[1] S. Abiteboul and A. Bonner. Objects and Views. *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 238–247, May 1991.

[2] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 311–322, May 1987.

[3] E. Bertino. A View Mechanism for Object-Oriented Databases. *Proc. of the 3rd Int'l Conference on Extending Database Technology (EDBT)*, pages 136–151, Mar. 1992.

[4] S. E. Bratsberg. Unified Class Evolution by Object-Oriented Views. *Proc. 12th Int'l. Conference on the Entity-Relationship Approach(ER)*, pages 423–439, Oct. 1992.

[5] P. Breche, F. Ferrandina, and M. Kuklok. Simulation of Schema Change using Views. *Proc. of the 6th Int'l Conference and Workshop on Database and Expert Systems Applications (DEXA)*, pages 247–258, Sept. 1995.

[6] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. *Proc. of ACM-SIGMOD Int'l Conference on Management of Data*, pages 12–21, May 1993.

[7] V. Crestana, A. Lee, and E. A. Rundensteiner. Sustaining Software Interoperability Via Shared, Evolving Object Repositories: System Optimization and Evaluation. *Proc. of Centre for Advanced Studies Conference (CASCON)*, Nov. 1996.

[8] V. Crestana, A. Lee, and E. A. Rundensteiner. Schema Version Removal: Optimizing Transparent Schema Evolution Systems. Technical Report CSE-TR-340-97, University of Michigan, Ann Arbor, 1997.

[9] V. Crestana and E. A. Rundensteiner. Consistent View Removal in Transparent Schema Evolution Systems. *Proc. of 6th Int'l Workshop on Research Issues on Data Eng.(RIDE), Interop. of Nontraditional DBMSs*, pages 138–147, Mar. 1996.

[10] C. Delcourt and R. Zicari. The Design of an Integrity Consistency Checker (ICC) for an Object Oriented-Database System. *Proc. of the 5th European Conference on Object-Oriented Programming (ECOOP)*, pages 97–117, July 1991.

[11] G.Guerrini, E.Bertino, B.Catania, and J.Garcia-Molina. A Formal View of Object-Oriented Database Systems. *Journal of Theory and Practice of Object Systems (TAPOS)*, 3(3):157–183, 1997.

[12] H. A. Kuno and E. A. Rundensteiner. New Benchmark Issues for Object-Oriented View Systems. *OOPSLA'95 Workshop on Object Database Behaviour, Benchmarks, and Performance*, Oct. 1995.

[13] H. A. Kuno and E. A. Rundensteiner. The *MultiView* OODB View System: Design and Implementation. *Journal of Theory and Practice of Object Systems (TAPOS), Special Issue on Subjectivity in Object-Oriented Systems*, 2(3):202–225, 1996.

[14] H. A. Kuno and E. A. Rundensteiner. Using Object-Oriented Principles to Optimize Update Propagation to Materialized Views. *Proc. of the 12th IEEE Int'l Conference on Data Engineering*, pages 310–317, Feb. 1996.

[15] S.-E. Lautemann. A Propagation Mechanism for Populated Schema Versions. *Proc. of the 13th IEEE Int'l Conference on Data Engineering*, pages 67–78, Apr. 1997.

[16] M. A. Morsi, S. B. Navathe, and H. J. Kim. A Schema Management and Prototyping Interface for an Object-Oriented Database Environment. *Proc. IFIP WG 8.1 Working Conference: The Object-oriented Approach in Information Systems*, pages 157–180, Oct. 1991.

[17] Y. G. Ra and E. A. Rundensteiner. Towards Supporting Hard Schema Changes in TSE. *Proc. of the 4th Int'l Conference on Information and Knowledge Management (CIKM)*, Nov. 1995.

[18] Y. G. Ra and E. A. Rundensteiner. A Transparent Schema Evolution System Based on Object-Oriented View Technology. *IEEE Transactions on Knowledge and Data Engineering*, pages 600–624, Sept. 1997.

[19] E. A. Rundensteiner. *MultiView*: Methodology for Supporting Multiple Views in Object-Oriented Databases. *Proc. of the 18th Int'l Conference on Very Large Data Bases (VLDB)*, pages 187–198, 1992.

[20] E. A. Rundensteiner. A Classification Algorithm for Supporting Object-Oriented Views. *Proc. of the 3th Int'l Conference on Information and Knowledge Management (CIKM)*, pages 18–25, Nov. 1994.

[21] E. A. Rundensteiner, A. J. Lee, and Y. G. Ra. Capacity-Augmenting Schema Changes on Object-Oriented Databases: Towards Increased Interoperability. *International Conference on Object-Oriented Information Systems*, Paris, France, Sept. 1998.

[22] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-oriented Databases. *Proc. of the 2th Int'l Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 189–207, Dec. 1991.

[23] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. *Proc. of the 5th Int'l Conference on Extending Database Technology (EDBT)*, pages 81–94, Mar. 1994.

[24] K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema Virtualization in Object-Oriented Databases. *Proc. of the 4th IEEE Int'l Conference on Data Engineering*, pages 23–30, Feb. 1988.

[25] R. Zicari. A Framework for O_2 Schema Updates. In *Proc. of the 7th IEEE Int'l Conference on Data Engineering*, pages 146–182, Apr. 1991.