Schema Version Removal: Optimizing Transparent Schema Evolution Systems *

Viviane M. Crestana, Amy J. Lee
Dept. of Elect. Eng. and Computer Science
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122
{viviane,amylee}@eecs.umich.edu

Elke A. Rundensteiner †
Computer Science Department
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609
rundenst@cs.wpi.edu

Powerful interoperability-enabling solutions for software application integration must allow applications to evolve and data requirements to change, while minimizing such changes on other integrated applications. The transparent schema evolution system (TSE), that we developed at the University of Michigan, accomplishes evolution through schema versioning, where a schema version is a dynamic object-oriented view. When the TSE system receives a view schema change request, the system computes a new view schema that reflects the desired change instead of modifying the old view schema in place, therefore preserving existing view schemas for old applications. This generation of a potentially large number of schema versions over time results in an excessive build-up of classes and underlying object instances, not all being necessarily still in use. Since our view system provides us with materialized views, a degradation of system performance due to update progagation is expected in a situation where we have no-longer utilized view schemas in the system. Also, a larger number of view schemas will result in storage overhead costs. In this paper, we address this problem using consistent schema removal techniques. First, we characterize four potential problems of schema consistency that could be caused by removal of a single virtual class; and then outline our solution approach for each of these problems. Second, we demonstrate that view schema removal is sensitive to the order in which individual classes are processed. Our solution to this problem is based on a formal model, called the dependency model, of capturing all dependencies between classes as logic clauses and of manipulating them to make decisions on class deletions and nondeletions while guaranteeing the consistency of the schema. Third, based on this formal model, we have developed and proven consistent a dependency graph (DG) representation and an associated set of rules for DG generation, reduction, and transformation. A first preliminary version of the latter has been successfully implemented in our Schema Version Removal (SVR) tool. Fourth, we present a cost model for evaluating alternative removal patterns on DG. Lastly, we report our preliminary experimental studies that demonstrate the impact of our schema version removal on the performance of the TSE system.

KEYWORDS: Transparent Schema Evolution, Object-Oriented Views, Interoperability, Object Databases, 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.

[†] While this work was done, the author was at the University of Michigan.

1 Introduction

1.1 Motivation

The general goal of this work is to develop powerful interoperability-enabling solutions for the integration of software applications [12, 13, 17]. These mechanisms must allow applications to evolve and flexibly change their data requirements, while minimizing or even eliminating the impact of such change on other integrated applications. In this vein, we are developing a Transparent Schema Evolution system called TSE [15, 16], which simulates schema evolution (SE) using object-oriented views on top of GemStone [1]. TSE combines SE with schema versioning where a schema version is a dynamic (object-oriented) view of the integrated global schema. When TSE receives a schema change request, the system computes a new view schema with the desired semantics; while preserving all existing view schemas (and thus old applications). This approach allows old application programs to continue to run against the schema they were designed for; and thus achieves interoperability of applications with diverse and even changing requirements.

While the TSE approach offers numerous advantages over traditional schema versioning systems, it may result over its lifetime in an excessive generation and revision of schema versions (view schemas). When a new view schema is computed it often adds new customized classes necessary for this view (this may cause the global schema and with it the set of underlying object instance representations to become larger and larger). In this sense, information is never deleted from the global schema. However, over time, old view interfaces may become obsolete for various reasons. One reason may be that we have successfully migrated all application code to a newer view. Another reason may be that a global error correction affected all existing views, and thus caused their replacement.

In this paper, we thus address this problem of excessive buildup of schema versions. The main goal of this work is the effective schema version removal (SVR) of redundant or out-of-use view schemas in TSE - without impacting existing applications. Removing faulty or non-current view schemas will make it easier for the application developer to determine which view schema to run against. As our experimental studies (Section 9) confirm, performance improvements will be achieved for the propagation of updates from a base class to its derived classes, given fewer materialized derived classes.

1.2 Problem Description

Removal of such view schemas – even if they are no longer in use – could have potential side effects on other view schemas and thus would not be *transparent* to the users. One difficulty here is caused by the fact that we allow definition of virtual classes based on other existing, possibly virtual, classes, and thus removal of one virtual class could cause other virtual classes to become undefined. Further, all view classes (whether base or virtual) from different view schemas are integrated together into a unified global schema [18]. This offers advantages, such as that virtual classes in the TSE system participate in the actual inheritance hierarchy and thus behave just like base classes. In fact, virtual classes in TSE can independently define additional attributes and methods. However, it now raises the issue that virtual classes may serve as source of inheritance for other classes, and thus their removal cannot be accomplished without first resolving possible side-effects caused this removal.

In traditional SE systems [3, 21], there is no schema version for individual users, but one schema is shared by all. In such systems, the semantics are to propagate all user-initiated changes to the whole schema. For example, a delete-class(C) command causes the deletion of properties from all of C's subclasses, if inherited from C. In many systems, e.g., O2 [21], we often have the additional precondition that the delete only proceeds if the class C is empty. In SVR, we are instead trying to achieve garbage-collection semantics, namely, our goal is to optimize the performance of the system without adversely impacting the rest of the schema, i.e., other applications and users. And, if we decide to proceed with this delete task, then we must remedy the undesired side-effect by, for instance, migrating methods from the to-be-deleted class C to its subclasses to make them local there, if needed, and redefining existing virtual classes derived from this class C, if any exist. These strategies for preservation have no correspondence in conventional SE systems, while they are critical for addressing our view removal problem.

Since we want the remaining schema to be unaffected by the schema version removal, the first issue that we have to deal with is consistency of the schema. Also, in our system, we want to delete a set of classes, and not only *undo* a virtual class. Conflicts arise when trying to remove multiple classes as we demonstrate in Section 5 and in this paper we develop several strategies for detection of those conflicts. Finally, if we

---

[1] Gemstone is a trademark of GemStone Systems, Inc.

detect a conflict between the removal of classes, we need a measure for the quality (cost) of a schema so that we can decide which is the best class to be removed, i.e., which class if removed will result in the best schema. *Best schema* here is used in the sense of allowing for efficient query or update processing.

1.3 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 virtual class, such as type-effect, derivation-dependency, etc. (these were problems we initially identified in an earlier workshop paper [8]). Then we outline our strategies for solving each of these problems. Key ideas here include strategies for the redefinition of virtual classes and for the migration of properties for preservation. Furthermore, we demonstrate that view schema removal 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 nondeletions 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. While a preliminary sketch of the formal model was first introduced in an earlier conference paper [7], this paper we now present the complete formal model, called the Dependency Model.

Based on the Dependency Model, we have developed a dependency graph (DG) representation capturing these class interdependencies. The graph representation is more suitable for implementation in our SVR tool. 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 and Boolean logic, we prove those rules to be consistent. In the process of operating with the DG, our system will also identify conditions of mutually exclusive removal, namely where removal of one virtual class will prevent removal of others in the future.

To address the problem that view schema removal is sensitive to the order in which individual classes are processed, in this paper we now 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 concepts, we have implemented a first simple prototype of the SVR tool on top of TSE system [15, 16].

Lastly, in this paper, we also report on experimental studies we have conducted to validate our approach. Our evaluation supports on the 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 TSE system.

1.4 Outline for Remainder of the Paper

The remainder of the paper is structured as follows. In Section 2, we present related work, while in Section 3 we give an overview of the underlying TSE and MultiView systems. A discussion on the issues related to schema consistency when removing a class from the schema, as well as solutions for each one of the issues is given in Section 4. In Section 5, we consider deletion of not only one, but several classes in the schema. Solutions for the multiple class removal, namely the dependency model and the dependency graph approach, are presented in Sections 6 and 7 respectively. A cost model for measuring efficiency of the schema is proposed in Section 8, while our performance studies are presented in Section 9. Section 10 summarizes our contributions and presents our future work.

2 Related Work

Several other approaches towards transparent schema evolution using views have been recently presented in the literature [4, 5, 10]. However, none have considered the schema removal problem as characterized in this paper. Like other view systems presented in the literature [19], in the formal model of views described in [10], the virtual classes are not integrated in the global schema. Therefore, removal of a virtual class in such a system does not impose any constraints regarding consistency of the global schema, since the view schema and the global base schema are not integrated. In the approach described in [10], a view schema is composed of only virtual classes, and therefore, in order to have a base class in a view schema, they provide a *identity virtual class* that has the same type and extent as the base class. This approach, while avoiding the integration, and therefore further removal consistency problem, has the serious drawback of duplication of information. In systems where the virtual classes are integrated in the global schema [2, 17, 20], code-reuse as well as sharing of data is assured. Our work addresses the removal of such integrated virtual classes.

In systems using conventional versioning rather than object-oriented view mechanisms [1], schema removal is likely to be less of an issue. Typically, complete copies of class versions are created, without being integrated into one global schema, resulting in a duplication of methods and thus no true interdependencies such as we have in our case. 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 an enormous 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 [14], class versions are placed in one global schema, where versions of a particular class forms a sub-hierachy 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 done carefully. As far as we know, the issues of removal have not been considered in the context of [14].

Our work is also related to the issue of *schema consistency,* e.g., Orion [3] or O2 [21, 9]. We both must 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 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 other classes. The delete-class example discussed in Section 1.1 clearly brings out these differences. Consequently, techniques we introduce in this paper to optimize schema removal, such as establishing an ordering for class removal processing, have not been developed for conventional schema evolution.

3 The TSE System

In the TSE system [15, 16], each user [2] defines his own customized interface of the shared database implemented as a materialized object-oriented view schema in the underlying MultiView system [12, 17]. All the schema change requests are made by the developer against the customized view schema to which the developer has access (Figure 1 (a)). The TSE system computes a new view schema that reflects the desired change (Figure 1 (b)) instead of modifying the old view schema in place. This approach allows old application programs (program1) to continue to run against the schema they were designed for (VS1).

The TSE system is built on top of the MultiView OO view system [12, 17] – which is one of the first fully implemented object-oriented view systems to provide updatable and incrementally maintained materialized object-oriented views [12, 13]. MultiView includes the incorporation of virtual classes into the global schema as first-class database citizens, assuring that properties shared among base and virtual classes are defined exactly once in the global schema. This not only assures code-reuse but also allows the values of properties shared among a base object and all its derived virtual instances to be stored only once.

MultiView supports a full range of virtual classes defined by a single object algebra operator, such as *select, hide, union,* etc. [17]. MultiView's materialized views [12] allow for fast retrieval of all instances from TSE, regardless whether the class we are retrieving the instance from is a base or virtual class. This shields users from the overhead of interoperating via an object repository potentially shared by many other users. In MultiView, a view schema is composed of a set of view classes (both base or virtual) and relationships among them. Each view class *V* in a view schema *VS* corresponds to a class *C* (either base or virtual) in the global schema GS, with C denoted by GS-class(*V*).

Throughout the paper we will be using the example of the university database depicted in Figure 2 (a). Suppose we have an initial view schema (Figure 2(b)) against which a developer issues two subsequent schema change operations, namely to delete the *nationality* property from the *Person* class and to delete the *gpa* property from the *Student* class. These two schema change requests create the view schemas presented in Figure 3 (a). In order to generate these view schemas, new virtual classes were created and the global schema was restructured (Figure 3 (b)). This example shows one of the schema change operators that the TSE system supports, namely the *delete-attribute* operator. A complete reference on the schema change operators supported by the TSE system can be found in [15, 16].

---

[2] The user here is a sophisticated user, i.e., a developer, and not an end user.

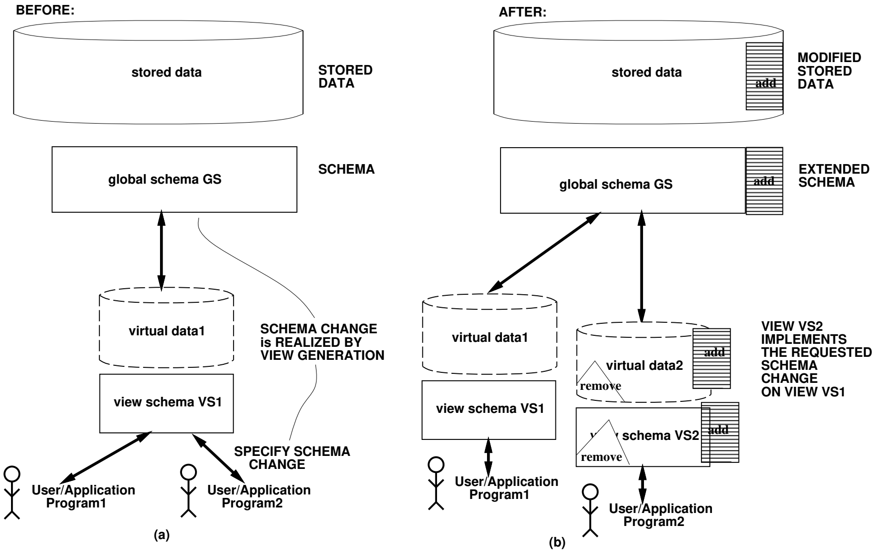
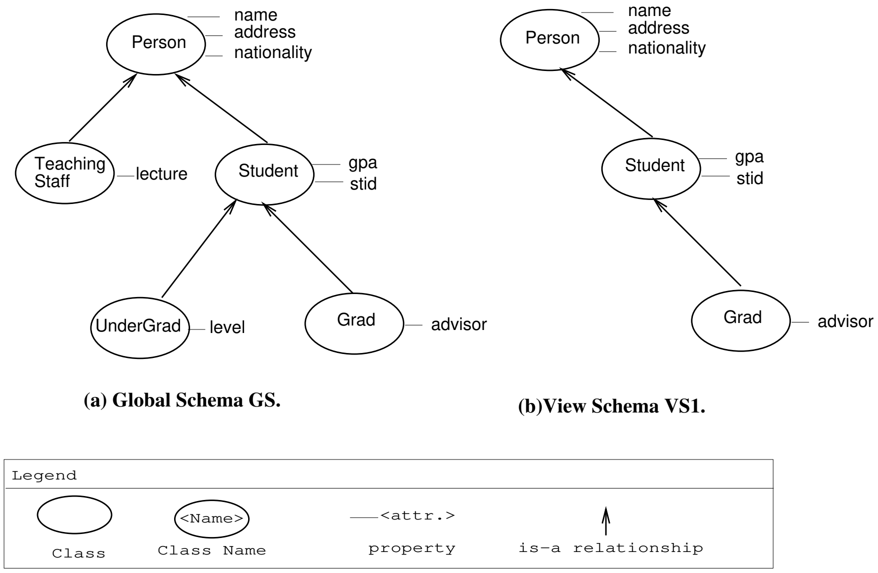


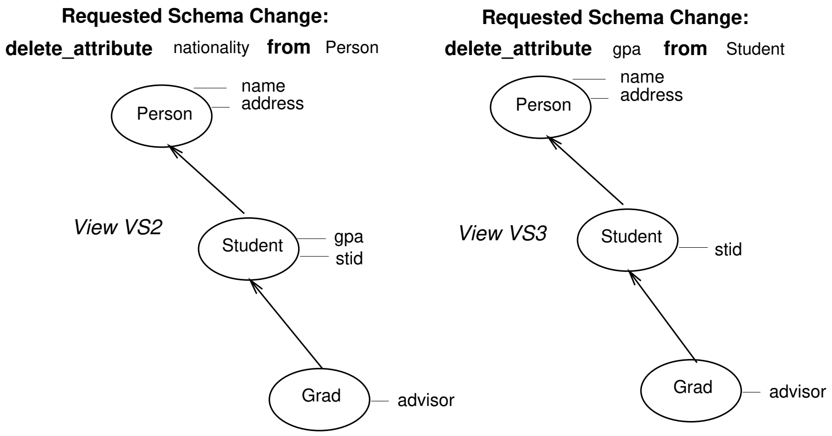
Figure 1: The TSE system approach



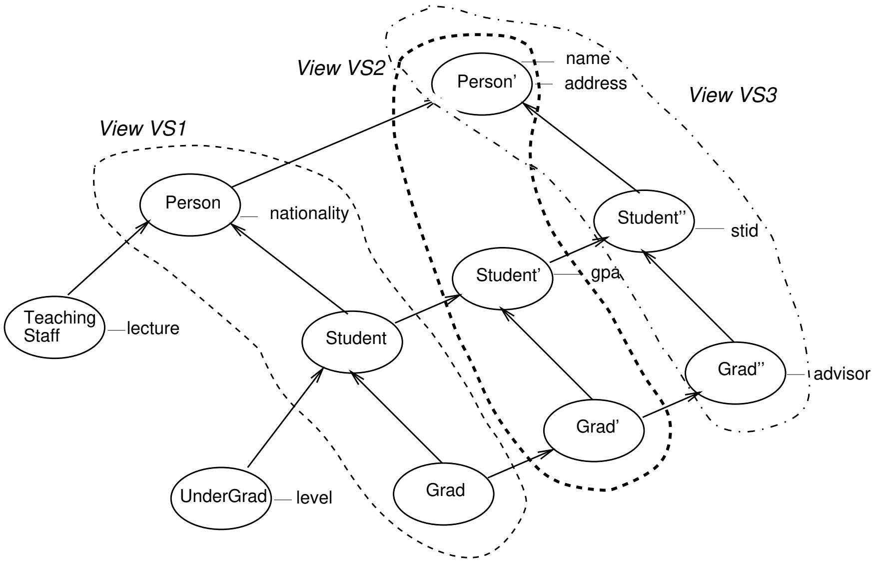
(a) Global Schema GS.

(b)View Schema VS1.

Figure 2: The University Database Example.



(a) Two View Schemas VS1 and VS2.



(b) Integrated Global Schema GS.

Figure 3: The university database after TSE operations

4 Single Class 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 based on schema restructuring, and lastly present the class-removal algorithm that integrates the strategies.

4.1 Constraints for Schema Removal

The purpose of our TSE system is to provide schema evolution in a *transparent* manner so old schemas are not affected. Following the same idea, we want to achieve a *transparent* removal of unused schemas. By that, we mean that all other view schemas (and their associated applications) should be unaffected by the removal of a given view schema. The invariants of the global schema can be summarized by:

- **SI1** There is one class called **root** in the global schema that is a superclass of all classes in the schema.
- **SI2** The IS-A hierarchy is a connected DAG.
- **SI3** A class always inherits all properties defined in its superclasses, except for the overridden properties. Consequently, the type of a class is defined by its locally defined properties and the inherited properties.
- **SI4** By the single-source principle, a given property in the schema graph is defined only once in one class as a local property; and other classes must inherit it from this class.
- **SI5** The derivation chain between virtual classes and their source classes is always acyclic.
- **SI6** A virtual class is specified as (and thus *implements*) only one object-algebra operator.

To assure *transparent removal*, i.e., that other schemas are not affected, SVR also needs to guarantee:

- **TR1** Every class in the global schema that corresponds to a view class in any view schema should be maintained in the global schema and should have exactly the same type and extent as before the removal.
- **TR2** For any two classes C and D in a view schema for which there is an IS-A edge from C to D, the relationship C IS-A D is still valid in the global schema.

By guaranteeing the two conditions stated above, we can guarantee that the view schemas will be unaffected (for a description of how view schemas are derived from GS see [12, 17]).

4.2 Problems of Class Removal

When removing a view schema, we would like to remove every class that participates in the view schema from GS. We are referring here to the removal of base or virtual classes from the global schema, since the view classes by themselves are just proxy objects in the view schema. The first concern is however to assure that no other view schemas will be affected. Since different view schemas share classes in the global schema, we have to make sure that we do not delete those shared classes. Also, when deleting a class we have to make sure that the resulting global schema is still in a consistent state after the removal, i.e., it still satisfies all the schema invariants.

The problems can be classified as follows:

- **P1. shared-class problem**: Classes that are shared by different view schemas still in use should not be deleted.
- **P2. ISA-hierarchy problem**: The DAG of the remaining schema should be consistent as well as the type and extent 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 an object or method that belongs to the class you are deleting.

In the sections that follow, we will be discussing and providing solutions to each of these problems. As mentioned before, the problem of removing schemas can be reduced down to to the problem of removing a particular class (base or virtual) from the underlying global schema. Therefore, in the following discussions, we will be analyzing the removal of a particular class, regardless of which view schema this class used to participate in.

4.3 Solution to the Shared-Class Problem

Since classes could be shared by different view schemas, we need a way to identify whether a given class is incorporated in other view schemas. We have identified two alternatives as potential solutions :

- Visit all view schemas in the system to see which view schemas use the class of interest.
- Cache information with each class about its usage in different view schemas.

We avoid the first alternative because the number of view schemas can be potentially very large. The drawback of the second alternative is basically its space requirements and some maintenance costs. Two ways to encode the information for the second solution are: (1) For each class in the global schema, keep a backward reference to view schemas that the class participates in; (2) Keep a reference counter with each class that keeps track of how many view schemas the class participates in. We have selected the second approach for SVR, since it can save significant space and the detailed information of who uses which class is not being used by SVR.

4.4 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 are 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 whether the class has some local properties that may be 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.*

Proof: The type of a class is determined by the locally defined properties and the inherited properties (**SI3**). Since a class can only inherit properties from its superclasses, the only classes affected are the subclasses. Now, suppose that by removing a class C, none of its direct-subclasses ($SB_1, SB_2, \ldots, SB_n$) have their type changed. We know that the type of any class in the schema can be determined by its locally defined properties and the type of its direct-superclasses (since they can only inherit properties through the class hierarchy). Therefore any class in the schema other than $C, SB_1, SB_2, \ldots, SB_n$ have their type unaffected since their direct-superclasses have the same type (note our assumption that $SB_1, SB_2, \ldots, SB_n$ have their type unaffected), and since their locally defined properties are still the same. Now we have to prove that if there exists a class in the global schema that has its type affected, there exists at least one direct-subclass that has its type affected. We prove this by contradiction, using the previous result, i.e., if none of the direct-subclasses had their type affected, then no class in the schema would have their type affected. □

By Theorem 1, to determine whether there will be classes that have their type affected, we just need to check the direct-subclasses. The three different situations that can happen when removing a class are the following:

1. *The class we are deleting has no subclass*: in this case the *type-effect* problem will not occur when deleting the class, because there is no class that inherits the locally defined properties.
2. *The class does have subclasses, but does not have locally defined properties*: since the subclasses do not directly inherit any properties from the class we are deleting, deleting the class will not change the type of its subclasses, provided we link its subclasses to its superclasses as explained before.
3. *The class has subclasses and locally defined properties*: in this case, we have to make sure the type of the subclasses will not change after the deletion. Since the class will be removed and its local properties are not defined anywhere else, we should define them somewhere. We cannot migrate them to the superclasses or else it would change their type. The solution is to migrate the properties to the direct-subclasses. However, if the class has more than one direct-subclass we cannot migrate the properties to each one of them, otherwise we would violate **SI4**. Going back to our university database example, in Figure 4 the delete of class *Person'* must thus be rejected, since the properties *name* and *address* are inherited by both *Person* and *Student"* classes.

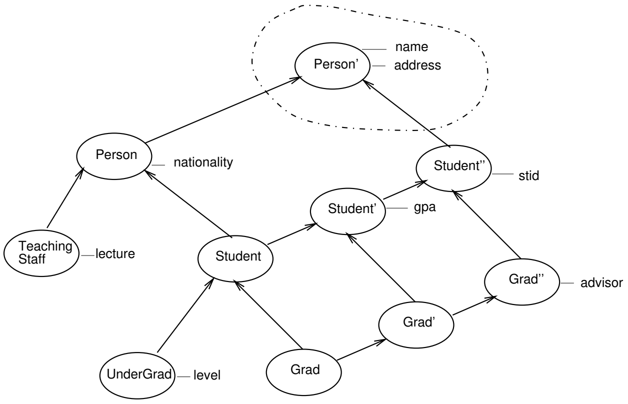


Figure 4: Example of Type-Effect Problem - Deleting *Person'* Class.

Therefore, we can establish a simple rule for removal of a class with regard to the ISA-hierarchy problem.

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. 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.*

4.5 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 analyzing the derived class, an error may occur when a derived class forwards the operation to its source class. To solve this problem we have to redefine the derived classes (including possibly changing their source class) in accordance with **SI5** and **SI6**. If the deletion cannot be performed without violating **SI5** and **SI6** the delete operation must be rejected.

Strategy 2 *Let VC be a virtual class derived from a source class C. Then, the solution we propose is to delete the class VC as long as there exists another class C' in the global schema that could become the source for VC to allow for redefinition of VC* [3]*.*

When specifying a virtual class by an object algebra operator such as hide, select, union, etc. [17], the type and extent of the derived class may or may not be the same as those of the source class. In order to determine the situations where the delete cannot be performed, we classify our object algebra operators used for view specification.

Definition 1 *A **type-preserving** object algebra operation is an operation where the new derived class preserves the type of its source class. A **type-augmenting (type-reducing)** object algebra operation is an operation where the the type of the derived class is a sub-type (super-type) of the source class' type.*

Definition 2 *An **extent-preserving** object algebra operation is an operation where the new derived class preserves the extent of its source class. An **extent-augmenting (extent-reducing)** object algebra operation is an operation where the extent of the derived class is a super-set (subset) of the source class' extent.*

The basic problem then is to determine whether and if so how we can redefine the derived classes. As an example if we want to redefine a hide class C' we first would have to find a new source that:

- is a subtype of type C' since hide is type-reducing; and
- have the same extent as C' since hide is extent-preserving.

One example of successfully redefining a hide class is shown in Figure 5. Here the hide class $Grad''$ finds a new source in *Grad* and thus a successful redefinition of $Grad''$ can take place when the original source $Grad'$ is deleted.

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

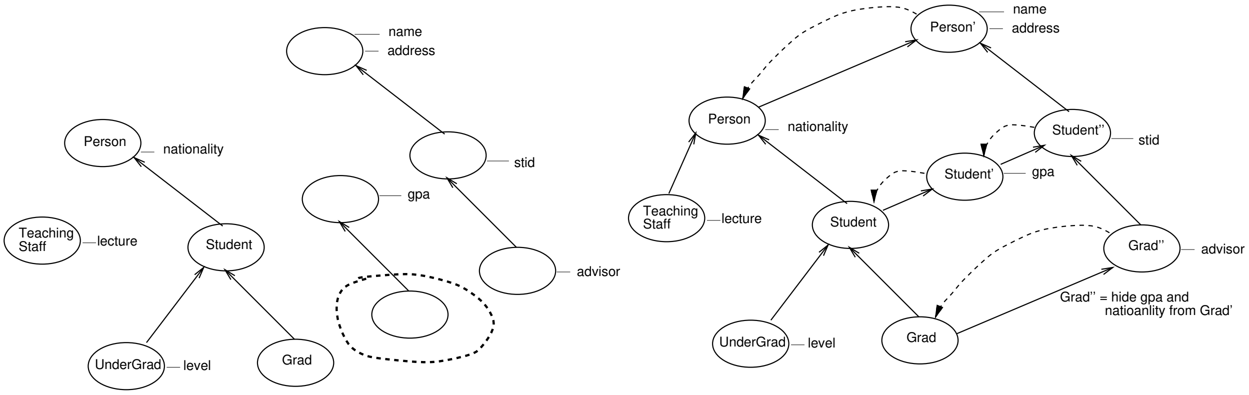
Theorem 2 *If there exists a class C' in the global schema that can be a new source for a virtual class VC (in place of C), then there exists at least one super- or subclass of the original source C that can also be a new source. Furthermore, if none of the super- or subclasses of C can be a new source for a virtual class VC then there exists no class in the global schema that can.*

The above theorem can intuitively be shown by the following argument given for the hide example. For the other virtual classes the reasoning is analogous. Suppose we want to redefine the hide class *HC*, where the original source class is *OS*. Let *SB* denote a subclass of *OS* and *SP* a superclass of *OS*. Hide is a type-reducing operation, therefore *OS* is a subtype of *HC*. All subclasses of *OS* are subtypes of it, and therefore, also subtypes of *HC*. Regarding the type, all subclasses of OS are potential new sources for *HC*. Assume then that *SB* is not a new source for *HC*, then extent(*SB*) is not the same as extent(*OS*). Since the extent of a subclass is always a subset of the extent of the superclass, extent(*SB*) and the extent of all subclasses of *SB* are smaller than the extent of *OS*. Therefore neither *SB* nor *SB*'s subclasses can be a new source hence the downward search can be terminated. In regard to the upward search there are two different reasons why *SP* cannot be a new source. First, if *SP* is not a subtype of the hide none of *SP*'s superclasses will be a subtype either. The second reason relating to the extent is analogous to the reasoning in the previous paragraph. If extent(*SP*) is not the same as the extent(*OS*), and *SP*, being a superclass, is a superset of *OS*, extent(*SP*) and the extent of all superclasses of *SP* are bigger than the extent of *OS*. Therefore neither *SP* nor *SP*'s superclasses can be a new source and the upward search can be terminated.

Strategy 3 *The strategy thus is to start looking for a new source for VC redefinition from the direct-subclasses of the original source and go downwards, and from the direct-superclasses of the original source and go upwards. If we cannot find one, we should stop our search, because no other class in the schema will be a new source.*

---

[3] a simplifying assumption we make is that a virtual class, when being redefined, does not change its VC type.



Grad'' = hide gpa and natioanlity from Grad'

(b)