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.

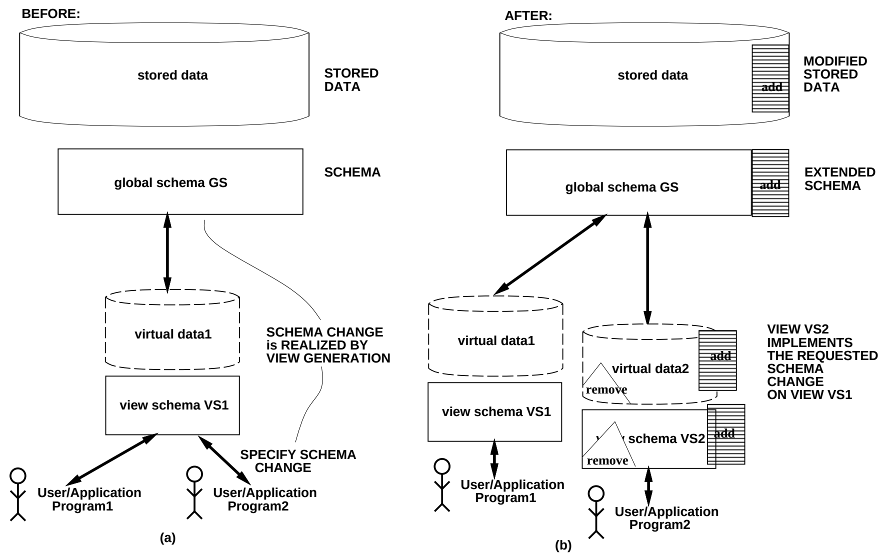


Figure 1: The TSE system approach

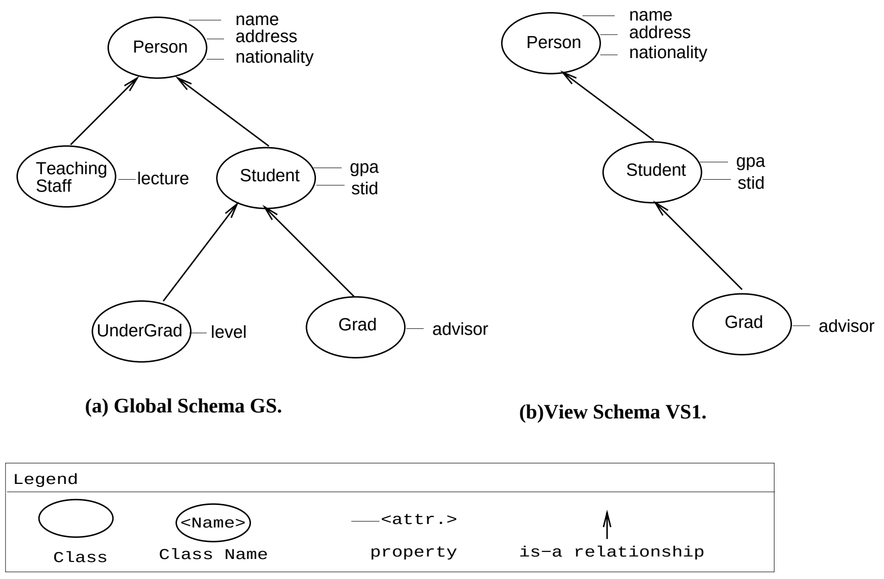
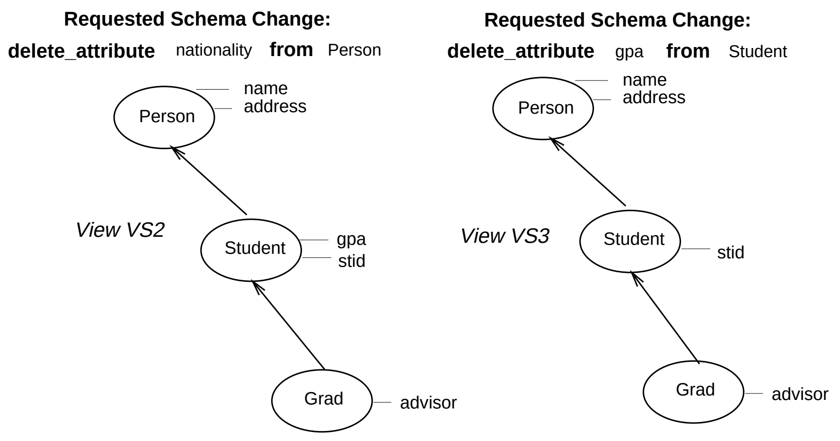
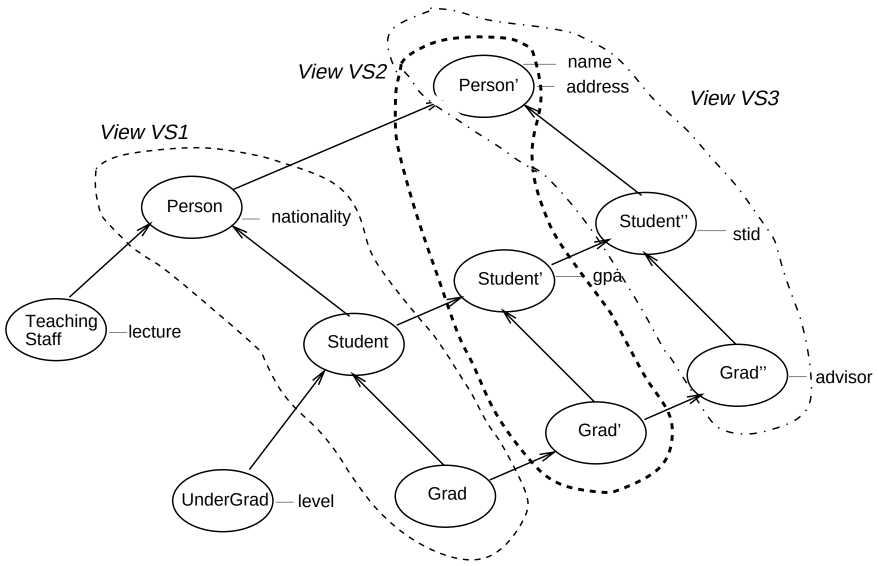


Figure 2: The University Database Example.



(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.

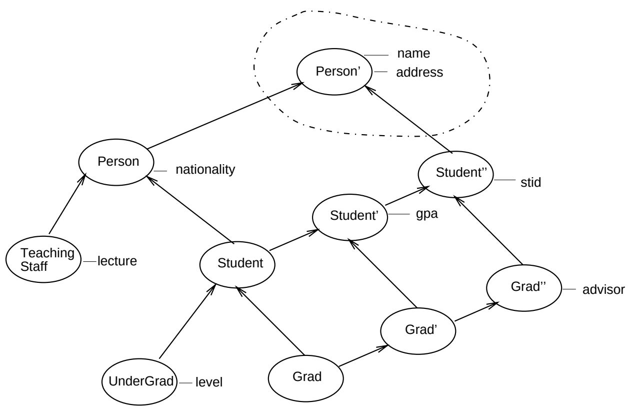


Figure 4: Example of Type-Effect Problem - Deleting *Person'* Class.

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.

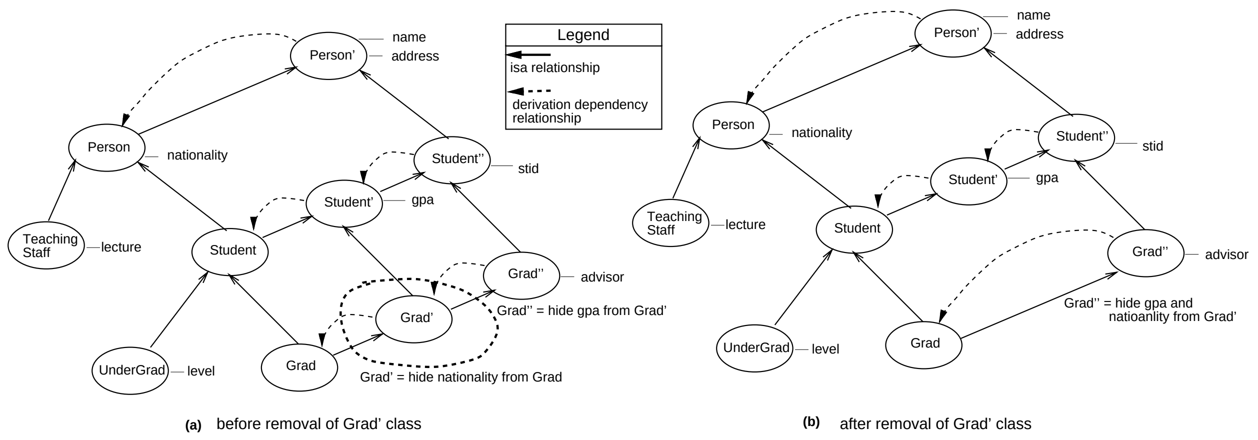
Now the problem we need to address is where to look for the new source. Potentially we could investigate all the classes in the global schema. However, due to the MultiView model of types of virtual classes, it is possible to significantly limit our search, as stated in Theorem 2.

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.



Legend
isa relationship
derivation dependency relationship
Person'
name
address
Person
nationality
Teaching Staff
lecture
Student
Student'
gpa
Student''
stid
Grad''
advisor
Grad'
Grad'' = hide gpa from Grad'
UnderGrad
level
Grad
Grad' = hide nationality from Grad
(a) before removal of Grad' class
Grad'' = hide gpa and natioanlity from Grad'
(b) after removal of Grad' class

Figure 5: Example - Deleting Class *Grad'* and Redefining *Grad''* Class.

4.6 Solution to the Undefined Reference Problem

The undefined reference problem 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 object in the schema. In [9] the approach recommended for solving the undefined reference problem (in particular, behavioral consistency) is to build a method-dependency graph that can be constructed by evaluating the code of each method. This is very expensive and may not be a feasible solution in practice.

In SVR the context is somewhat different because classes are not arbitrarily deleted from the schema. 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 [4], MultiView assumes closed view schemas [17]. 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 objects of 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.

4.7 Basic Algorithm to Remove a 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*)

```
01    if $C$ is shared then return
02    if $C$ has locally-defined properties and
        $C$ has more than one direct-subclass then return
03    for each $D \in$ derived-classes($C$) do
04        if cannot find new-source for $D$ then return
05    for each $D \in$ derived-classes($C$) do redefine $D$
06    if $C$ has locally-defined properties then
07        $C'$ := direct-subclass ($C$)
08        migrate properties of $C$ to $C'$
09    for each $SB \in$ direct-subclasses($C$) do
10        for each $SP \in$ direct-superclasses($C$) do
11            if not ($SB$ IS-A* $SP$) then
12        add-edge $SB$ IS-A $SP$
13    remove class $C$
```

When the **RemoveClass** algorithm is applied to a consistent schema, 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 6-8). 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 9 through 12. 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.

5 Interdependencies Problems of Multiple Class Removal

In the previous section, we presented the 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 one might attempt to address this problem 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. This problem can be easily illustrated with an example (refer back to Figure 5 (a)). Suppose we want to delete classes *Student'* and *Grad'*. If the sequence followed is *Student'*, *Grad'*, then the deletion of class *Student'* would be prohibited. If the sequence followed is *Grad'*, *Student'*, the type-effect problem will be resolved because by the time class *Student'* is checked, the *Grad'* class does not exist anymore. Of course that is not the only condition to be checked when trying to delete a class, but assuming that the tests on lines 01 and 04 are not satisfied, the type-effect problem would be the only condition to be met. It thus appears that if we only had the type-effect problem to deal with, then the best solution is to start from the leaves of the global schema DAG.

The derivation-dependency problem is analogous. You might not be able to delete a source class because it has some derived class that cannot be redefined; however, later it may be discovered that you can delete the derived class. In this case the best sequence would be to start from the derived classes and then look for the source classes. So for the derivation-dependency problem the best approach could be top-down for hide classes or bottom-up for refine classes. If we consider these problems together there appears to be not one best sequence (top-down vs. bottom-up) to follow in all situations.

In both these cases, by deleting one class, we were able to delete another class that we could not have deleted previously. One might argue that since we want to delete as many classes as possible, no matter which class we start from, we just have to keep iterating until no more changes are made. However, there exist another kind of pairwise dependency between two classes. Namely, 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.

To illustrate this idea, let us go back to the original University database example shown in Figure 2 (a). Suppose that in the process of customizing an interface to some application, we had the following virtual classes created:

- *Student2* as hide *stid* from *Student*
- *TA* as intersect *Student2* with *TeachingStaff*
- *Student3* as hide *nationality* from *Student2*
- *Student4* as hide *gpa* from *Student3*

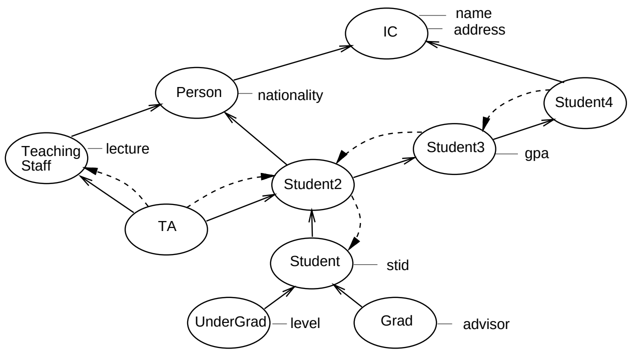


Figure 6: Example - Extension to Schema in Figure 2 (a).

The global schema that results from the addition of these virtual classes is shown in Figure 6[4]. In this case, suppose we detected that classes *Student2* and *Student3* are not being used by any view schema and therefore can be removed.

Conflict 1 (*type-effect problem*) : When we analyze the two classes *Student*2 and *Student*3 separately, both of them satisfy the criteria for deletion without causing the *type-effect problem*. Namely, *Student*2 has no locally defined properties, so its deletion will not change the type of its subclasses, and *Student*3 has locally defined properties but has only one subclass, so its properties can be migrated. However, if we delete *Student*2, its direct-subclasses, *TA* and *Student* will become direct-subclasses of *Student*3. Therefore, when we try to remove *Student*3, it does not satisfy the criteria for deletion anymore, because it has locally defined properties and two direct-subclasses. Conversely, if we delete *Student*3 first, its property *gpa* gets migrated to *Student*2 class. Consequently, when we try to delete *Student*2, it has locally defined properties and two direct-subclasses, so it can no longer be deleted.

Conflict 2 (*derivation-dependency problem*) : Removing *Student*2 by itself causes no problem since *TA* can be redefined from *Student*3. Removing *Student*3 by itself also does not cause any problem once *Student*4 can be redefined from *Student*2 or *Student*. However, if we want to remove both *Student*2 and *Student*3, 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 *Student*4, *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).

6 The Dependency Model

To overcome the difficulties presented in the previous section, we approach the search order problem by first detecting all possible conflicts between deletions of classes. For this purpose, we have developed a formal model based on Boolean logic, called the Dependency Model. Each dependency between 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). Each particular clause that represents dependency between 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 a class will be deleted or not. A clause that corresponds to the example illustrated in Section 5 (**Conflict 1** and **Conflict 2**), is: $(\overline{D_{Student2}} \vee \overline{D_{Student3}})$, where $D_{Student2}$ is 1 if *Student*2 is deleted and 0 otherwise, and $D_{Student3}$ is 1 if *Student*3 is deleted and 0 otherwise. In order to satisfy this clause, we need either $D_{Student2}$ or $D_{Student3}$ to be zero, i.e., we cannot have both equal to 1, or equivalently, we cannot delete both classes. We represent each such dependency with a clause. By making sure that all clauses are satisfied simultaneously, we are guaranteed to have a consistent final global schema.

[4]Note that some Intermediate Classes (namely, IC in this example) were created to provide schema consistency (as explained in [17]).

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 could experiment with different deletion patterns and check whether all consistency requirements are met. We can potentially 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, but later on we show how we can improve upon this potentially inefficient perfomance.

Since we want to "experiment" with the 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 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.

6.1 *Type-effect* problem

Recall from Section 4, 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 5, a conflict might arise between classes to be removed (see **Conflict 1**). A class may conflict with its direct-superclasses (case of *Student*2 conflicting with *Student*3) or with its direct-subclasses (case of *Student*3 conflicting with *Student*2). Furthermore, as classes are removed from the schema, the ISA-hierarchy changes and therefore the dependencies between classes change. So a class might conflict with other subclasses that were not its direct-subclasses in the original schema. Remember that we want to be able to detect whether the deletion of a class will affect the deletion of other classes in the schema. 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:

- **condition 1**: The class C will never have more than one direct-subclass. 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 deletion of *Student*2 causes the non-deletion of *Student*3). 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) 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. $\square$

  (b) 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. $\square$

- **condition 2**: 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 5 when deletion of *Student*3 causes the non-deletion of *Student*2). 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) 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, and 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. $\square$

  (b) 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. $\square$

- **condition 3**: The class C at any point in time has 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 1**.

In order to capture the *type-effect* dependency, we associate with each class C_i the following Boolean variables [5]:

- D_i indicates the label of the class C_i: 1, if it is marked *deletable*; 0, if it is marked *nondeletable*.
- 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 ($ST_i = 0$).

[5] 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.

We can guarantee **condition 1** (and **condition 3** 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 **condition 2** for class C_i if: C_i has no locally defined properties and will not get any property migrated to it. 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 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 the following:

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:

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

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 $C_{i_{j_1}} \ldots C_{i_{j_n}}$ are the direct-subclasses of C_{i_j}.

6.2 *Derivation-dependency* problem

Recall that if a derived class is not deleted, we have 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 $C_{i_2}, ..., C_{i_n}$ be the alternative sources. We express this requirement in our model as the following:

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

6.3 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 (also, 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 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 also 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, for instance, the example in Figure 6. 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 $D_{Student3} = 1$.

7 The Dependency Graph Approach

We have developed and implemented a hypergraph representation of this formal model, called the dependency graph (DG), that 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.

7.1 The Dependency Graph Model

In our SVR system, we thus solve the search order problems, outlined in Section 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 hypergraph $DG = (N, E)$, 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.

Recall that in the process of making decisions about which class to delete (or marking classes as *deletable* or *non-deletable*), we substitute the values of the variables D and we might end up with a different dependency between classes. 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 6 (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:

  $OSorNP(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$

  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 6.

- Since the link $OSorNP$ states that class C_i can be deleted if either OS_i or NP_i are 1, 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 6.

- The link $OSorNP$ states that class C_i can be deleted if either OS_i or NP_i are 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 6.

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 along our 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 are 1. If we determine that one subclass C_k will not be deleted (D_k is 0), we need all subtrees corresponding to the other subclasses to be deleted in order for OS_i to be 1. Therefore the consistency requirement is reduced down to: $(\overline{D_i} \vee (ST_{i_1} \wedge ST_{i_2} \wedge \cdots \wedge ST_{i_n}) \vee NP_i)$. 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 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 6 (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 $C_{i_2}, \ldots, C_{i_n}$ are the alternate sources. We represent this dependency by a link:

  $remainPropagate(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$

  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.

- Since the link *remainPropagate* states that if class C_i remains in the schema, at least one of the source classes should also remain, if we know that class C_i will remain (i.e., it is non-deletable), at least one of the sources should remain. Therefore the requirement is: $(\overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_n}})$. 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).

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

7.2 The DG Generation Rules

As explained in Section 6, 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 ($GS$)
01    for each $C_i \in GS$ do
02        add a node $C_i$ to $DG$
03    for each $C_i \in GS$ do
04        if $C_i$ has subclasses
05            if $C_i$ has local properties
06                add link $OSonly$ to $DG$:
                    $OSonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ [6].
07            else
08                add link $OSorNP$ to $DG$:
                    $OSorNP(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$ [7]
09        if $C_i$ is a derived class
10            detect all classes in $GS$ that can be a source class for $C_i$
11            add link $remainPropagate$ to $DG$
                $remainPropagate(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ [8]
```

[6] $\{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}$ are the direct-subclasses of C_i

[7] $\{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}$ are the direct-subclasses of C_i and $\{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\}$ are the direct-superclasses of C_i.

[8] $\{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}$) are the classes found in step 10.

7.3 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 don't 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 6 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 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.

We proceed with this DG reduction process by first analyzing the classes and determining which ones will be deleted, which ones will 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.

Note that 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 further 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., $D_i = 1$.*

Proof: Consider a class C_i such that D_i does not appear in any clause negated. Let $c \in CR$ be a clause. If c does not involve D_i, it will not be affected. Suppose c involves D_i and suppose, by contradiction, that when $D_i = 0$, $c = 1$ and when $D_i = 1$, $c = 0$. If $c = 1$, by replacing D_i's the value from 0 to 1 we make a conjunction or disjunction of literals that was true become false. However, if D_i is not negated, if $V \vee D_i = 1$ when $D_i = 0$, then $V \vee D_i = 1$ when $D_i = 1$. Also, if $V \wedge D_i = 0$ when $D_i = 1$, then we must have $V \wedge D_i = 0$ when $D_i = 0$. Therefore, we cannot make a conjunction or disjunction of literals that was true to become false, and therefore if c is false when $D_i = 1$ it must be the case that c was false when $D_i = 0$. $\square$

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 $D_i = 1$. Since 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 "cheaper" 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, 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 $OSorNP$ link
- anywhere in a $OSonly$ link
- anywhere in a $NPonly$ link

- in the *origin* or second *destination set* of a $STorNP$ link
- in the *origin* of a $STonly$ link
- in the *origin* of a $remainPropagate$ link
- anywhere in a $minimalRemaining$ link

Therefore, in the preprocessing phase, we mark a class C' *deletable* if it does not appear in any of the situations described above.

7.4 The DG Transformation Rules

Now the question is: *what happens to the links in the graph once we have determined that particular classes will or will not be deleted?*. 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 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 Appendix A. We first give transformation rules for when a class is marked *non-deletable* and then transformation rules for when a class is marked *deletable*.

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: remove the link.

   Proof: By substituting D_i by 0 in the original clause: $(OS_i \vee NP_i \vee \overline{D_i}) = (OS_i \vee NP_i \vee 1) = 1$. That means that the clause is satisfied no matter what the values of OS_i and NP_i are. Therefore the link can be removed.

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:

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

   Proof: The original clause is: $(OS_i \vee NP_i \vee \overline{D_i})$, where
   $OS_i = (ST_{i_1} \wedge ST_{i_2} \wedge \cdots \wedge ST_{i_n}) \vee ((OS_{i_1} \vee \overline{D_{i_1}} \vee LP_{i_1}) \wedge ST_{i_2} \wedge \cdots \wedge ST_{i_n})$
   $\vee \cdots \vee (ST_{i_1} \wedge \cdots ST_{i_{j-1}} \wedge (OS_{i_j} \vee \overline{D_{i_j}} \vee LP_{i_j}) \wedge ST_{i_{j+1}} \wedge \cdots \wedge ST_{i_n})$
   $\vee \cdots \vee (ST_{i_1} \wedge ST_{i_2} \wedge \cdots \wedge (OS_{i_n} \vee \overline{D_{i_n}} \vee LP_{i_n}))$. If $D_{i_j} = 0$, then $ST_{i_j} = 0$. By substituting ST_{i_j} by 0 we get:
   $OS_i = (ST_{i_1} \wedge \cdots ST_{i_{j-1}} \wedge 0 \wedge ST_{i_{j+1}} \wedge \cdots ST_{i_n}) \vee \cdots \vee ((OS_{i_1} \vee \overline{D_{i_1}} \vee LP_{i_1}) \wedge \cdots ST_{i_{j-1}} \wedge 0 \wedge ST_{i_{j+1}} \wedge \cdots ST_{i_n})$
   $\vee \cdots \vee (ST_{i_1} \wedge \cdots ST_{i_{j-1}} \wedge (OS_{i_j} \vee 1 \vee LP_{i_j}) \wedge ST_{i_{j+1}} \cdots \wedge ST_{i_n})$
   $\vee \cdots \vee (ST_{i_1} \wedge \cdots ST_{i_{j-1}} \wedge 0 \wedge ST_{i_{j+1}} \cdots \wedge (OS_{i_n} \vee \overline{D_{i_n}} \vee LP_{i_n})) = (ST_{i_1} \wedge \cdots ST_{i_{j-1}} \wedge ST_{i_{j+1}} \cdots \wedge ST_{i_n})$.
   So, the new clause is:
   $((ST_{i_1} \wedge \cdots ST_{i_{j-1}} \wedge ST_{i_{j+1}} \cdots \wedge ST_{i_n}) \vee NP_i \vee \overline{D_i})$ which is the link given above.

note: if in the original link there was only one subclass , i.e. $n = 1$, and so $j = 1$, then we could remove the link.

Proof: The original clause is: $(OS_i \vee NP_i \vee \overline{D_i})$, where
$OS_i = (ST_{i_1}) \vee (OS_{i_1} \vee \overline{D_{i_1}} \vee LP_{i_1}))$. By substituting ST_{i_1} by 0 and D_{i_1} by 0, we get:
$OS_i = (0 \vee (OS_{i_1} \vee 1 \vee LP_{i_1})) = 1$. Therefore, the clause $(OS_i \vee NP_i \vee \overline{D_i}) = (1 \vee NP_i \vee \overline{D_i}) = 1$. Therefore the clause is true no matter what the values of NP_i and D_i are. So we can remove the link.

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*:

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

    Proof: By substituting D_i by 0 in the original clause:

    $((ST_{i_1} \wedge ST_{i_2} \wedge \cdots \wedge ST_{i_n}) \vee NP_i \vee \overline{D_i}) = ((ST_{i_1} \wedge ST_{i_2} \wedge \cdots \wedge ST_{i_n}) \vee NP_i \vee 1) = 1$. That means that the clause is satisfied no matter what the values of $ST_{i_1}, ST_{i_2}, \ldots, ST_{i_n}$ and NP_i are. Therefore the link can be removed.

15. If $remainPropagate(C_i, \{C_{i_1}, C_{i_2}, \ldots C_{i_n}\})$ 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 $C_{i_1}, \ldots, C_{i_n}$ should remain. Therefore, this link is translated into a link that represent this dependency between classes $C_{i_1}, \ldots, C_{i_n}$ and does not include class C_i: $minimalRemaining(C_{i_1}, \{C_{i_2}, \ldots, C_{i_n}\})$

    Proof: By substituting D_i by 0 in the original clause:
    $(D_i \vee \overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_n}}) = (0 \vee \overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_n}}) = (\overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_n}})$ which is the *minimalRemaining* link given above. Note that, the *minimalRemaining* link corresponds to an or-ed clause of negated D_i's, and therefore, it would be also correct to transform the link to: $minimalRemaining(C_{i_2}, \{C_{i_1}, C_{i_3}, \ldots, C_{i_n}\})$, for example.

    note: a special case happens when there is only one class in the destination set (say C_{i_1}). Since at least one of the classes in the destination set should remain and C_{i_1} is the only one, C_{i_1} should remain. Therefore: C_{i_1} is *non-deletable*.

    Proof: By substituting D_i by 0 in the original clause: $(D_i \vee \overline{D_{i_1}}) = (0 \vee \overline{D_{i_1}}) = (\overline{D_{i_1}}) \Rightarrow D_{i_1} = 0$.

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

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

    Proof: by substituting D_{i_j} by 0 in the original clause:

    $(D_i \vee \overline{D_{i_1}} \vee \cdots \overline{D_{i_j}} \vee \cdots \vee \overline{D_{i_n}}) = (D_i \vee \overline{D_1} \vee \cdots \overline{D_{i_{j-1}}} \vee 1 \vee \overline{D_{i_{j+1}}} \vee \cdots \vee \overline{D_{i_n}}) = 1$. That means that the clause is satisfied no matter what the values of $D_i, D_{i_1}, \ldots, D_{i_{j-1}}, D_{i_{j+1}}, \ldots, D_{i_n}$ are. Therefore, we can remove the clause/link.

Transformation Rules in case a class is *deletable*:

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

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

    Proof: by substituting D_i by 1 in the original clause: $(D_i \vee \overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_n}}) = (1 \vee \overline{D_{i_1}} \vee \overline{D_{i_2}} \vee \cdots \vee \overline{D_{i_n}}) = 1$. That means that the clause is satisfied no matter what the values of $D_{i_1}, \ldots, D_{i_n}$ are. Therefore, we can remove the link.

33. If $minimalRemaining(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and C_i is *deletable*:

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

$minimalRemaining(C_{i_1}, \{C_{i_2}, \ldots, C_{i_n}\}$.

Proof: by substituting D_i by 0 in the original clause:

$(\overline{D_i} \vee \overline{D_{i_1}} \vee \cdots \vee \overline{D_{i_n}}) = (0 \vee \overline{D_{i_1}} \vee \cdots \vee \overline{D_{i_n}}) = (\overline{D_{i_1}} \vee \cdots \vee \overline{D_{i_n}})$ which is the *minimalRemaining* link given above.

7.5 The Decision-Making Phase

When there are no conflicts between which classes to remove (as explained in Section 5) we can completely determine during the preprocessing phase which classes to delete and which classes not to delete. When there exist interdependencies between multiple class removal options, 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 should 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, is corresponds to a class appearing in one of the following links:

- in the *origin* of a *OSonly* link, or
- in the *origin* of a *STonly* link, or
- 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 7.4, since marking a class might affect other classes. With that, we also reduce the set of combinations we have to examine.

7.6 The Overall View Class Removal Process

Figure 7 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 7.2 we start the preprocessing phase discussed in Section 7.3. The preprocessing phase finishes when we exit the test "any rule applied" with a NO answer for the first time.

After the preprocessing phase, we have to make decisions of which classes to mark (See the bottom box of the flowchart in Figure 7 marked "Select a class ...".). In Section 7.5, we presented different options of how to select classes to mark. This selection strategy in the most simplest case could be to randomly pick a class to delete – which obviously is not 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

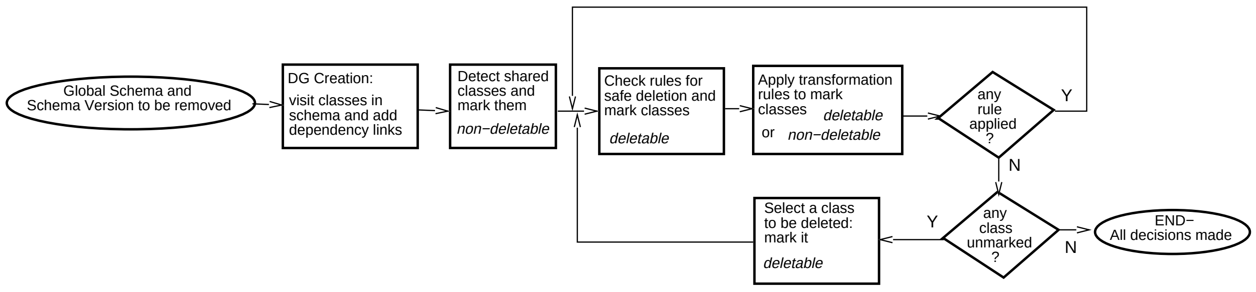

Global Schema and Schema Version to be removed
DG Creation: visit classes in schema and add dependency links
Detect shared classes and mark them non−deletable
Check rules for safe deletion and mark classes deletable
Apply transformation rules to mark classes deletable or non−deletable
any rule applied ?
Y
N
any class unmarked ?
Y
N
Select a class to be deleted: mark it deletable
END− All decisions made

Figure 7: The Overall Process for Determining Classes to Delete.

of the "smallest cost" of a schema graph as determined by the cost model introduced in the Section 8. We explain the overall process with the following 2 examples.

Example1: Figure 8 illustrates the overall process with the schema given in Figure 6. Suppose the classes $Student2$ and $Student3$ in Figure 6 are no longer used in views and we thus want to remove them. Following the algorithm given in Section 7.2, 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 *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 8 (a) shows the $remainPropagate$ links generated. We omit the $OSorNP$ and $OSonly$ links for simplicity of the figure.

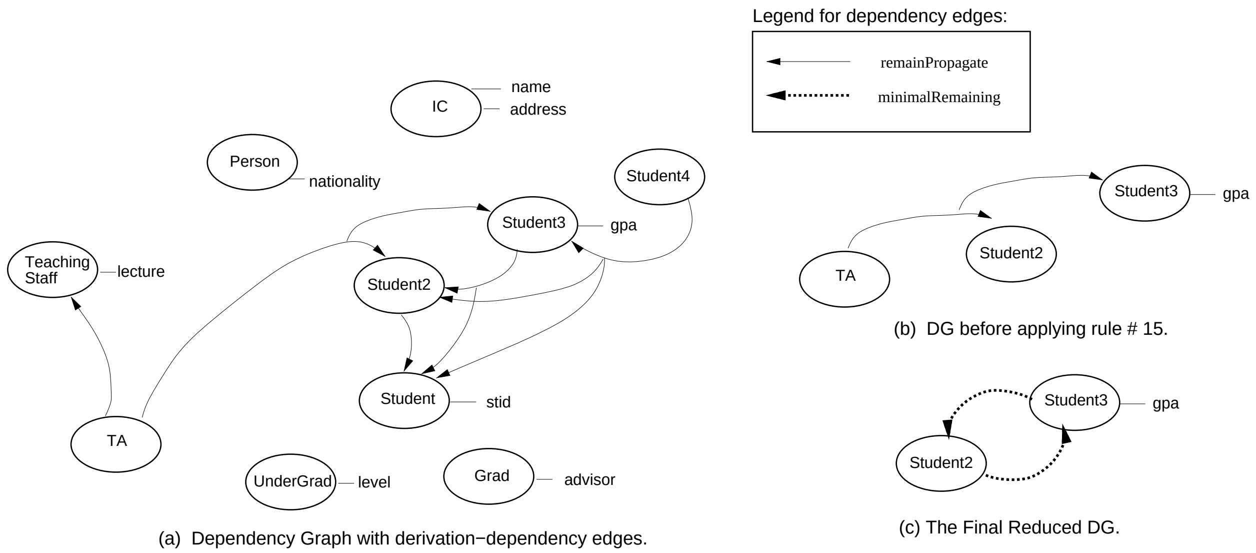


Figure 8: Initial and Reduced Dependency Graph Representations for the Schema in Figure 6.

The preprocessing phase starts by marking every class other than $Student2$ and $Student3$ as *non-deletable*. Then, 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*:

- $OSorNP(Student2, \{TA, Student\}, \{Student3, Person\})$

  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:

  $minimalRemaining(Student3, \{Student2\})$.

Now, with regards to the $remainPropagate$ links, since $Student$ and $Teaching-Staff$ are *non-deletable* by applying rule # 16, we are left with only one link (see Figure 8 (b)). Lastly, by applying rule # 15, and the fact that TA is *non-deletable*, we are left with the link: $minimalRemaining(Student2, \{Student3\})$. Therefore, we get the final reduced graph shown in Figure 8 (c). This DG indicates that removing $Student2$ will preclude us from removing $Student3$, and vice versa.

Example2: In the example in Figure 9(a), AtomicPart is a base class; APSel1, APSel2, APSel3, and Inter4 are virtual classes of type Select (the first three) and Intersect (the last one). Suppose that APSel1, APSel2 and APSel3 become obsolete and we wish to remove them from the global schema. In the DG generation stage, we create the following links:

1. $OSonly(AtomicPart, \{APSel1, APSel3\})$
2. $OSorNP(APSel1, \{APSel2\}, \{AtomicPart\})$
3. $OSorNP(APSel3, \{Inter4\}, \{AtomicPart\})$
4. $OSorNP(APSel2, \{Inter4\}, \{APSel1\})$
5. $remainPropagate(APSel1, \{AtomicPart\})$
6. $remainPropagate(APSel2, \{APSel1, AtomicPart\})$
7. $remainPropagate(APSel3, \{AtomicPart\})$
8. $remainPropagate(Inter4, \{APSel3\})$
9. $remainPropagate(Inter4, \{APSel1, APSel2\})$

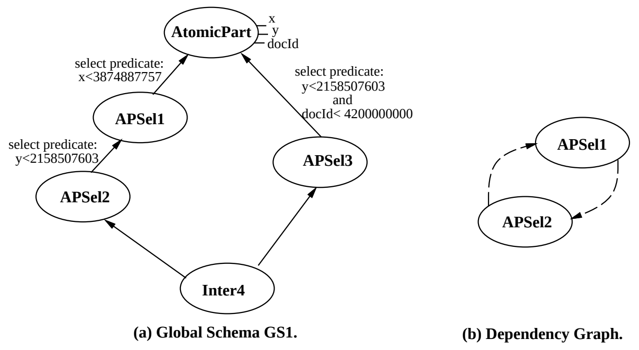

AtomicPart
x
y
docId
select predicate: x<3874887757
select predicate: y<2158507603 and docId< 4200000000
APSel1
select predicate: y<2158507603
APSel3
APSel2
Inter4
(a) Global Schema GS1.
APSel1
APSel2
(b) Dependency Graph.

Figure 9: Example of Global Schema and Associated Dependency Graph.

In the preprocessing phase, we first mark the classes $AtomicPart$ and $Inter4$ *non-deletable.* Then by using rule #4, we remove the 1st link, by using rule #3 we remove the 2nd and 3rd links, and rule #2 allows us to remove the 4th link. Then using the fact that *AtomicPart* is *non-deletable*, we can use rule #16 to remove links 5, 6 and 7. Since *Inter*4 is *non-deletable*, we use rule #15 (the special case) and mark *APSel*3 as *non-deletable*. Finally, we can use rule #15 and the fact that *Inter*4 is *non-deletable* to change link 9 to: $minimalRemaining(APSel1, \{APSel2\})$[9]. The resulting DG graph (Figure 9(b)) shows the interdependency between the removal of these two classes. That is, if APSel1 is removed, then APSel2 cannot be removed, and vice versa.

8 Cost Model for Guiding the Class Removal Process

We can easily see that for the examples given in the previous section, a simple decision criterion of selecting the deletion that results in a schema with the minimal number of classes is useless, since both resulting schemas would have the same number of classes (for both examples). Thus, we propose to use a cost model that models the quality of the schema to guide our decision about which removal to execute.

8.1 The Overall Cost Evaluation Process

Let GS denote a global schema that consists of classes C = { C_1, C_2, . . . , C_n }, and let the resulting global schema after removing a class C_i be $GS|C_i$, where i ∈ [1,n]. Now, assume both C_i and C_j, where i, j ∈ [1,n] and i ≠ j, can be removed individually without violating the consistency requirements, but their removal is mutually exclusive. In these circumstances, a more sophisticated decision criterion for guiding the class removal order, namely a cost model for evaluating the goodness of the alternative global schemas $GS|C_i$ and $GS|C_j$ is needed. That is, we choose to remove a class that results in a global schema with lower costs. Since the focus of this paper is on materialized view systems, we consider the view maintenance costs [10]. Our MultiView system utilizes materialized virtual classes [12, 13], therefore updates on base or virtual classes have to be propagated to all dependent derived classes in order to keep the materialized view classes consistent. In this paper, we consider insert, delete, and change-attribute operations on base classes [11].

Let $COST_{GS}$ be the view maintenance costs associated with the global schema GS. Then our strategy is to remove C_i, iff $COST_{GS|C_i} < COST_{GS|C_j}$. Otherwise, we remove C_j. Let baseClasses(GS) be the base classes in the schema GS, and derivedClasses(C_i) be the classes in the derivation hierarchy rooted at C_i. The propagation cost for the operation x at a base class C_i, denoted as $propCost(x, C_i)$, is equal to the actual update cost at the base class C_i and the summation of the update costs of the classes in derivedClasses(C_i), where x can be insert, delete, or change-attribute operation. The view maintenance cost of a schema GS is defined as follows:

$$\begin{aligned} COST_{GS} = \textstyle\sum_{C_i \in baseClasses(GS)} \{ & (propCost(insert, C_i) * numInsert_i) + \\ & (propCost(delete, C_i) * numDelete_i) + \\ & \textstyle\sum_{a:\ attribute\ of\ C_i} (propCost(change, a, C_i) * numChange_{a,i}) \}. \end{aligned} \tag{1}$$

where $numInsert_i, numDelete_i$, and $numChange_{a,i}$ are the number of insert, delete, and change-attribute updates operated on C_i.

Note that one type of update operation may cause the same type and/or other types of update operations 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. Similarly, the delete propagation effects are summarized in

[9] This link is also equivalent to $minimalRemaining(APSel2, \{APSel1\})$.

[10] Retrieval itself from a class C_k can be considered to be identical in both $GS|C_i$ and $GS|C_j$ - independent from other classes in the respective schemas when all classes are materialized. We do not consider the costs of restructuring the global schema required to correctly remove a class C_i 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.

[11] While our MultiView system allows updates on virtual classes as well, such updates are delegated by our system to the underlying source classes for the updated view classes. Hence, updates on virtual classes can be reduced to updates on base classes, and hence their discussion is omitted here for simplicity.

Equation 3. Note that changing an object's attribute value on 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 cause the *attribute value change* on 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.

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

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

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

An individual insert, delete or change-attribute 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 an attribute a of an object in the class C_k, and $cost_{insert}, cost_{delete}$, and $cost_{change-attr}(a)$ represent the costs to insert a new object into a class, the costs to delete an object from a class, and the costs to change the value of attribute a of an object, respectively. These parameters $cost_{insert}, cost_{delete}$, and $cost_{change-attr}(a)$ are assumed to be given [12].

In order to compute the propagation costs for each type of update operation, we need to know how often an update on a base class is propagated to its derived class. For this, we need to know the parameters PI_k, PD_k, $PC_k(a)$, and $PM_{j|i}$ for each base class (which are assumed to be given by the developer). $PM_{j|i}$ is the probability that an object belongs to C_j given that the object belongs to C_i, and it is needed to calculate PI_k and PD_k for a derived class C_k assuming C_i and C_j are its source classes. The same set of parameters PI_k, PD_k, and $PC_k(a)$ is computed for each of the derived classes as detailed next.

8.2 Determination of Propagation Probabilities for Virtual Classes

In this section, we will show how the values of $PI_k, PD_k, PC_k(a)$, and $PM_{j|k}$ introduced above for base classes are calculated for virtual classes. These parameters are derived iteratively through the derivation chain(s), while updates are performed on its rooted base class(es). The table in Figure 10 summarizes the propagation probabilities; with each row capturing one virtual class type C_k and its four associated parameters $PI_k, PD_k, PC_k(a)$, and $PM_{j|k}$ in columns 2, 3, 4 and 5, respectively. If there are two source classes for the derived class, denoted by C_{i1} and C_{i2}, we assume for simplicity that only one of the source classes is operated upon by an insert/delete/change-attribute at each time.

[12]The parameters $cost_{insert}, cost_{delete}$, *and* $cost_{change-attr}(a)$ are implementation dependent.

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 10: Propagation Probabilities Parameters for Derived Classes.

For a Hide virtual 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 derived class (row 2 of the table).

In order to compute the parameters for a Select derived 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 update operations 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 evaluated to true afterwards [13].

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 operation operated on 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 operation (with probability Sel_k).

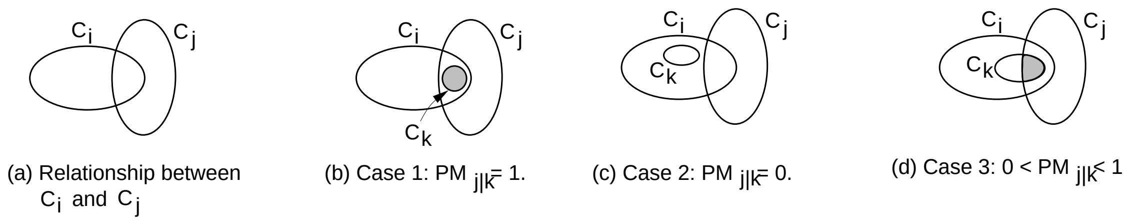


Figure 11: Different cases for Calculating Multiple Membership Probability PM.

There is no straightforward way to calculate the multiple membership probability PM for the Select class. Let's use Figure 11 to illustrate the intrinsic difficulties associated with deriving PM. As shown in Figure 11, there are three situations that may occur. Figure 11(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 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 11(b) shows

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

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 11(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 11(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 classes of a Difference class matter, since Difference is not symmetric (row 7). Note there is no straightforward way to calculate the multiple membership probability PM for these derived classes. We make a similar assumption about independence for the Union, Intersect, and Difference classes.

9 Experimental Evaluation

9.1 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 virtual classes indeed reduces the overall incremental view maintenance costs, and (2) the cost model introduced in the previous section guides us to choose the most effective class removal patterns. The initial version of SVR, which had a simpler dependency graph model [8], was successfully implemented on top of MultiView. While currently working 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 virtual classes – as there is currently no available OO benchmarking testbed for object-oriented views [11]. As depicted in Figure 12(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[14] running on SUN Sparc-10 workstation with a 48MB main memory.

9.2 Experiment One

In this experiment (Figure 12(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 virtual classes in the global schema indeed decreases the view maintenance costs of our TSE system - as assumed by the overall approach presented in this paper.

First, we ran change-attribute operations [15] on 1% of the extent of the *AtomicPart* and *Document* classes, 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 virtual classes. Applying the class removal techniques introduced earlier, we conclude that removing these virtual 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 12(b)), and measure the view update propagation time for it. Note in Figure 12(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 virtual class redefinition strategy. Next we removed $DocSel$, and again measured the run time for the same set of update operations for the resulting $GS3$ (Figure 12(c)).

Figure 13 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 virtual classes, such as Hide, Select, and Intersect, all of which have the same findings. These results support our goal of removing as many obsolete

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

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

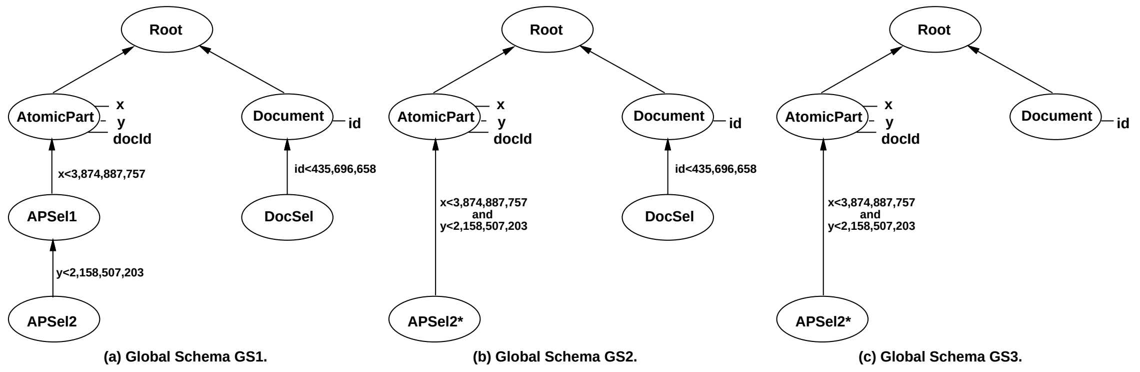


Figure 12: Global Schema Used in Test Case 1.

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

Figure 13: The Results of Experiment 1.

materialized virtual classes as possible, when their safe removal has been determined by our dependency model.

9.3 Experiment Two

In this experiment, we have a simple global schema with a base class, *AtomicPart*, and a Select virtual class. We vary the selectivity factor of the Select virtual class and run a random set of change-attribute updates (to guarantee that a certain percent of updates, approximately equal to the selectivity factors, propagates to the derived classes). This experiment is designed to demonstrate that our cost model is reliable, meaning that the quality of the global schemas measured by our cost model correctly reflects the runtime experimental results.

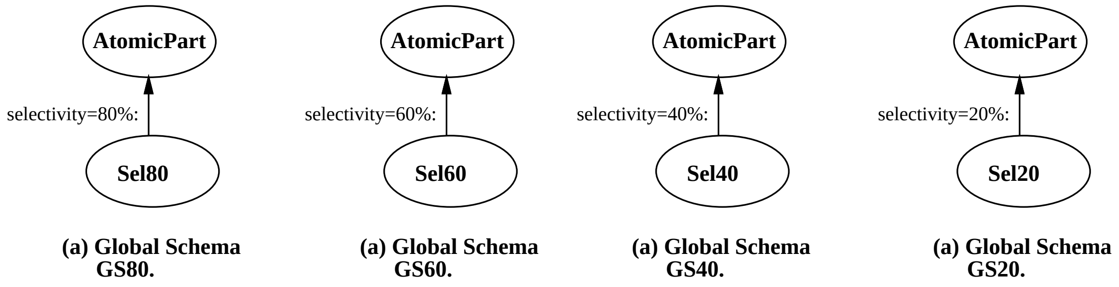


Figure 14: Global Schema Used in Test Case 2.

This experiment is run with four different selectivity factors: 80%, 60%, 40%, and 20%, and we name the corresponding global schema as $GS80$, $GS60$, $GS40$, and $GS20$ respectively (Figure 14). Let us now elaborate on the calculation of the change-attribute propagation costs. For this purpose, we need to compute the propagation probabilities of insert, delete, and change-attribute on the Select virtual class. We set the basic parameters $PI_{AtomicPart} = 0, PD_{AtomicPart} = 0, PC_{AtomicPart}(a) = 1, \forall\ a :\ \textit{an attribute of AtomicPart}$. Now, we compute PI, PD, and PC for Sel80, Sel60, Sel40, and Sel20 using the cost model introduced in Section 8. We get:

1. $PI_{Sel80} = 0 * 0.8 + (1 - 0.8) * (0.8) * 1 = 0.16$
2. $PD_{Sel80} = 0 * 0.8 + (1 - 0.8) * (0.8) * 1 = 0.16$
3. $PC_{Sel80}(a) = 1 * 0.8 * 0.8 = 0.64$
4. $PI_{Sel60} = 0 * 0.6 + (1 - 0.6) * (0.6) * 1 = 0.24$
5. $PD_{Sel60} = 0 * 0.6 + (1 - 0.6) * (0.6) * 1 = 0.24$
6. $PC_{Sel60}(a) = 1 * 0.6 * 0.6 = 0.36$
7. $PI_{Sel40} = 0 * 0.4 + (1 - 0.4) * (0.4) * 1 = 0.24$
8. $PD_{Sel40} = 0 * 0.4 + (1 - 0.4) * (0.4) * 1 = 0.24$
9. $PC_{Sel40}(a) = 1 * 0.4 * 0.4 = 0.16$
10. $PI_{Sel20} = 0 * 0.2 + (1 - 0.2) * (0.2) * 1 = 0.16$
11. $PD_{Sel20} = 0 * 0.2 + (1 - 0.2) * (0.2) * 1 = 0.16$
12. $PC_{Sel20}(a) = 1 * 0.2 * 0.2 = 0.04$

Hence, we get [16]:

$$\begin{aligned}
COST_{GS80} &= \textstyle\sum_{\forall a:\ attribute\ of\ AtomicPart} propCost(change, a, AtomicPart) * numChange_{a,AtomicPart} \\
&= \textstyle\sum_{\forall a:\ attribute\ of\ AtomicPart} \sum_{c \in [AtomicPart, Sel80]} \\
&\qquad \{cost(insert, c) + cost(delete, c) + cost(change, a, c)\} \\
&= 1 + 0.16 + 0.16 + 0.64 = 1.96
\end{aligned}$$

Other global schemas *GS60, GS40*, and *GS20* have the same actual updates costs on the *AtomicPart* class, and the only difference lies on the Select classes with different selectivity factors. The propagation costs of all the global schemas are reported in Figure 15. To confirm the validity of the finding achieved by the cost model, we now evaluate the costs on our real system. We first run a random set of change-attribute updates on the *AtomicPart* base class (on 1% of the *AtomicPart* extent) having a certain percent of updates, approximately equal to the selectivity factors, propagate to the Select class, then we measure the view maintenance costs. As illustrated in Figure 15, the run time results coincide with the results we get from using our cost model.

Global Schema	GS80	GS60	GS40	GS20
Selectivity Factor	80%	60%	40%	20%
View Maintenance Cost	1.96	1.84	1.64	1.36
Running time (milliseconds)	6471	6377	4833	3177

Figure 15: The Results of Experiment 2.

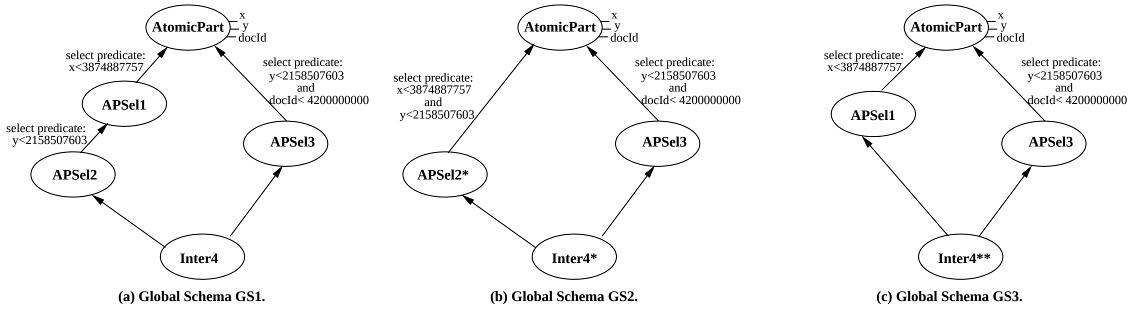

AtomicPart
x
y
docId
select predicate: x<3874887757
select predicate: y<2158507603 and docId< 4200000000
APSel1
select predicate: y<2158507603
APSel3
APSel2
Inter4
(a) Global Schema GS1.
select predicate: x<3874887757 and y<2158507603
APSel2*
Inter4*
(b) Global Schema GS2.
Inter4**
(c) Global Schema GS3.

Figure 16: Global Schema Used in Test Case 3.

9.4 Experiment Three

This experiment is designed to explore the usage of our cost model. For Figure 9(a), let us we 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 9(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 operations on 1% of the $AtomicPart$ extent, i.e., the $numChange_{a,AtomicPart}$ in Equation (1) is 100 change-attribute updates (1% of the 10000 objects), and $numInsert_i$ and $numDelete_i$ are both zero. We get:

$$\begin{aligned} COST_{GS2} &= \textstyle\sum_{\forall a:\ attribute\ of\ AtomicPart} propCost(change, a, AtomicPart) * numChange_{a,AtomicPart} \\ &= \textstyle\sum_{\forall a:\ attribute\ of\ AtomicPart} \sum_{c \in [AtomicPart, APSel2*, APSel3, Inter4*]} \\ &\qquad \{cost(insert, c) + cost(delete, c) + cost(change, a, c)\} \end{aligned}$$

$$\begin{aligned} COST_{GS3} &= \textstyle\sum_{\forall a:\ attribute\ of\ AtomicPart} propCost(change, a, AtomicPart) * numChange_{a,AtomicPart} \\ &= \textstyle\sum_{\forall a:\ attribute\ of\ AtomicPart} \sum_{c \in [AtomicPart, APSel1, APSel3, Inter4**]} \\ &\qquad \{cost(insert, c) + cost(delete, c) + cost(change, a, c)\} \end{aligned}$$

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 16(b) and (c)). Now, we compute the change-attribute propagation costs for these four classes. We begin with the 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, PC, and PM for $APSel1$, $APSel2*$, $Inter4*$, and $Inter4**$. 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. $PI_{Inter4*} = 0.2475 * 0.98 = 0.24255$

[16] We assume that the values of $cost_{insert}$, $cost_{delete}$, and $cost_{change-attr}(a)$ are approximately the same.

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. $PC_{Inter4**}(a) = 0.81$

Because $cost(insert, APSel1) + cost(delete, APSel1) + cost(change, a, APSel1) + cost(insert, Inter4**) + cost(delete, Inter4**) + cost(change, a, Inter4**) = 1.8882$ and $cost(insert, APSel2*) + cost(delete, APSel2*) + cost(change, a, APSel2*) + cost(insert, Inter4*) + cost(delete, Inter4*) + cost(change, a, Inter4*) = 1.3851$, if we consider that the values of $cost_{insert}, cost_{delete}$ and $cost_{change-attr}(a)$ [17] 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 $APSel1$.

Global Schema	GS	GS2	GS3
with Class		APSel2*	APSel1
Selectivity Factor		45.32 %	90.16 %
Num of total classes	6	5	5
Running time (milliseconds)	27023.5	10908.9	16793.4

Figure 17: The Results of Experiment 3.

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 change-attribute 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 17, the run time results are proportional to the results we get from using our cost model.

10 Conclusions

10.1 Contributions

In this paper, we provide a characterization of the virtual class removal problem in the context of MultiView/TSE. Results of this work should improve the efficiency of transparent schema evolution systems such as TSE [15], and thus increase their utility as mechanisms for enabling interoperability, 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 [2, 17, 20]. 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 nondeletions 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

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

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 DG and thus guiding the decision process of class removal.

In this paper, we also present results from the experiments we have conducted on top of the MultiView/TSE system. In the small example provided in Section 9 (see Figure 13), we were able to improve 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. Finally, we have implemented a preliminary version of the SVR system that removes classes and guarantees that the final schema is still consistent. This prototype version does however not yet incorporate the complete dependency graph model and in particular the transformation rules as detailed in this paper.

10.2 Future Work

In the future, a new version of the SVR implementation needs to be developed to incorporate the complete dependency graph as described in this paper, 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 MultiView. 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 MultiView/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 [17, 18] so that the new multiple operator classes can be properly placed in the global schema as well as changes in the view maintenance algorithm [12, 13]. The work described in this paper still applies for a system where multiple operator classes are allowed. The changes we envision need to be done to our solution presented in this paper are with regards to the way new sources are found (the algorithm itself), and the way redefinition is done.

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.

References

[1] Report on the Object-Oriented Database Workshop: Panel on Schema Evolution and Version Management. In *SIGMOD Records, Vol 18, No.3*, September 1989.

[2] S. Abiteboul and A. Bonner. Objects and views. *SIGMOD*, pages 238–247, 1991.

[3] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD*, pages 311–322, 1987.

[4] E. Bertino. Integration of heterogeneous data repositories by using object-oriented views. *International Workshop on Interoperability in Multidatabase Systems*, pages 22–29, April 1991.

[5] Svein Erik Bratsberg. Unified class evolution by object-oriented views. In *Proc. 12th Intl. Conf. on the Entity-Relationship Approach*, pages 423–439, 1992.

[6] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. *SIGMOD*, 1993.

[7] V. Crestana, A. Lee, and E. A. Rundensteiner. Sustaining software interoperability via shared, evolving object repositories: System optimization and evaluation. *To appear in CASCON'96*, 1996.

[8] V. Crestana and E. A. Rundensteiner. Consistent view removal in transparent schema evolution systems. *Sixth Int. Workshop on Research Issues on Data Eng., Interop. of Nontraditional DBMSs*, (RIDE'96, IEEE), 1996.

[9] C. Delcourt and R. Zicari. The design of an integrity consistency checker (ICC) for an object oriented-database system. In P. America, editor, *ECOOP*, pages 97–117, 1991.

[10] J. Garcia-Molina E. Bertino, B. Catania and G. Guerrini. A formal model of views for object-oriented database systems. (unpublished), 1996.

[11] H. A. Kuno and E. A. Rundensteiner. New benchmark issues for object-oriented view systems. In *OOPSLA Workshop on Object-Oriented Database Benchmarking*, October 1995.

[12] H. A. Kuno and E. A. Rundensteiner. The *MultiView* OODB view system: Design and implementation. In Harold Ossher and William Harrison, editors, *Accepted by Theory and Practice of Object Systems (TAPOS), Special Issue on Subjectivity in Object-Oriented Systems.* John Wiley New York, 1996.

[13] H. A. Kuno and E. A. Rundensteiner. Using object-oriented principles to optimize update propagation to materialized views. In *IEEE International Conference on Data Engineering*, pages 310–317, 1996.

[14] M. A. Morsi, S. B. Navathe, and H. J. Kim. A schema management and prototyping interface for an object-oriented database environment. In F. Van Assche, B. Moulin, and C. Rolland, editors, *Object-Oriented Approach in Information Systems*, pages 157–180. Elsevier Science Publishers B. V. (North Holland), 1991.

[15] Y. G. Ra and E. A. Rundensteiner. A transparent object-oriented schema change approach using view schema evolution. In *IEEE International Conference on Data Engineering*, pages 165–172, March 1995.

[16] Y. G. Ra and E. A. Rundensteiner. A transparent schema evolution system based on object-oriented view technology. *to be published in IEEE Transactions on Knowledge and Data Engineering*, 1996.

[17] E. A. Rundensteiner. *MultiView*: A methodology for supporting multiple views in object-oriented databases. In *18th VLDB Conference*, pages 187–198, 1992.

[18] E. A. Rundensteiner. A classification algorithm for supporting object-oriented views. In *International Conference on Information and Knowledge Management*, pages 18–25, November 1994.

[19] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *International Conference on Extending Database Technology (EDBT)*, 1994.

[20] M. H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proceedings of the Second DOOD Conference*, December 1991.

[21] R. Zicari. A framework for O_2 schema updates. In *7th IEEE International Conf. on Data Engineering*, pages 146–182, April 1991.

A DG Transformation Rules

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: remove the link.

   Proof: By substituting D_i by 0 in the original clause: $(OS_i \bigvee NP_i \bigvee \overline{D_i}) = (OS_i \bigvee NP_i \bigvee 1) = 1$. That means that the clause is satisfied no matter what the values of OS_i and NP_i are. Therefore the link can be removed.

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:

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

   Proof: The original clause is: $(OS_i \bigvee NP_i \bigvee \overline{D_i})$, where
   $OS_i = (ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee ((OS_{i_1} \bigvee \overline{D_{i_1}} \bigvee LP_{i_1}) \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n})$
   $\bigvee \cdots \bigvee (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge (OS_{i_j} \bigvee \overline{D_{i_j}} \bigvee LP_{i_j}) \bigwedge ST_{i_{j+1}} \bigwedge \cdots \bigwedge ST_{i_n})$

   $\bigvee \cdots \bigvee (ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge (OS_{i_n} \bigvee \overline{D_{i_n}} \bigvee LP_{i_n}))$. If $D_{i_j} = 0$, then $ST_{i_j} = 0$. By substituting ST_{i_j} by 0 we get:
   $OS_i = (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge 0 \bigwedge ST_{i_{j+1}} \bigwedge \cdots ST_{i_n}) \bigvee \cdots \bigvee ((OS_{i_1} \bigvee \overline{D_{i_1}} \bigvee LP_{i_1}) \bigwedge \cdots ST_{i_{j-1}} \bigwedge 0 \bigwedge ST_{i_{j+1}} \bigwedge \cdots ST_{i_n})$
   $\bigvee \cdots \bigvee (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge (OS_{i_j} \bigvee 1 \bigvee LP_{i_j}) \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n})$

   $\bigvee \cdots \bigvee (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge 0 \bigwedge ST_{i_{j+1}} \cdots \bigwedge (OS_{i_n} \bigvee \overline{D_{i_n}} \bigvee LP_{i_n})) = (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n})$.
   So, the new clause is:
   $((ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n}) \bigvee NP_i \bigvee \overline{D_i})$ which is the link given above.

   note: if in the original link there was only one subclass , i.e. $n = 1$, and so $j = 1$, then we could remove the link.

   Proof: The original clause is: $(OS_i \bigvee NP_i \bigvee \overline{D_i})$, where
   $OS_i = (ST_{i_1}) \bigvee (OS_{i_1} \bigvee \overline{D_{i_1}} \bigvee LP_{i_1}))$. By substituting ST_{i_1} by 0 and D_{i_1} by 0, we get:
   $OS_i = (0 \bigvee (OS_{i_1} \bigvee 1 \bigvee LP_{i_1})) = 1$. Therefore, the clause $(OS_i \bigvee NP_i \bigvee \overline{D_i}) = (1 \bigvee NP_i \bigvee \overline{D_i}) = 1$. Therefore the clause is true no matter what the values of NP_i and D_i are. So we can remove the link.

3. 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 $j \in \{n+1..n+m\}$:

   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, in case it was possible that properties were migrated to it. If a direct-superclass will not be deleted, we know that class C_i will not get any properties migrated from C_{i_j}. However, we still have to account for the other superclasses. Therefore, change link to:

   $OSorNP(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_{n+m}}\})$

   Proof: The original clause is: $(OS_i \bigvee NP_i \bigvee \overline{D_i})$, where
   $NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge (\overline{D_{i_j}} \bigvee NP_{i_j}) \bigwedge \cdots \bigwedge (\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$. By substituting D_{i_j} by 0, we get:
   $NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge (1 \bigvee NP_{i_j}) \bigwedge \cdots \bigwedge (\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$

$= \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge (\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$. So, the new clause is:

$(OS_i \bigvee NP_i \bigvee \overline{D_i})$, where NP_i is the clause computed above. This clause is equivalent to the link given above.

note: if in the original link there was only one superclass, i.e. $m = 1$, so $j = n + 1$ then we could remove the link.

Proof: The original clause is: $(OS_i \bigvee NP_i \bigvee \overline{D_i})$, where
$NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}})$. By substituting D_{i_j} by 0 and LP_i by 0 (since C_i does not have any local property up to the moment, otherwise the link *OSonly* would have been generated instead), we get: $NP_i = 1 \bigwedge (1 \bigvee NP_{i_{n+1}}) = 1$. Therefore, the clause $(OS_i \bigvee NP_i \bigvee \overline{D_i}) = (OS_i \bigvee 1 \bigvee \overline{D_i}) = 1$. Therefore the clause is true no matter what the values of OS_i and D_i are. So we can remove the link.

4. If *OSonly*$(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ 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: remove the link.

   Proof: By substituting D_i by 0 in the original clause: $(OS_i \bigvee \overline{D_i}) = (OS_i \bigvee 1) = 1$. That means that the clause is satisfied no matter what the value of OS_i is. Therefore the link can be removed.

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

   The link states that C_i can be deleted provided it has only one subclass (since it has local properties). If we know that one subclass will remain in the schema, class C_i can be deleted only if all other subclasses' subtrees are also deleted. Therefore, we change the link to:

   STonly$(C_i, \{C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_n}\})$

   Proof: The original clause is: $(OS_i \bigvee \overline{D_i})$, where
   $OS_i = (ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee ((OS_{i_1} \bigvee \overline{D_{i_1}} \bigvee LP_{i_1}) \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n})$
   $\bigvee \cdots \bigvee (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge (OS_{i_j} \bigvee \overline{D_{i_j}} \bigvee LP_{i_j}) \bigwedge ST_{i_{j+1}} \bigwedge \cdots \bigwedge ST_{i_n})$

   $\bigvee \cdots \bigvee (ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge (OS_{i_n} \bigvee \overline{D_{i_n}} \bigvee LP_{i_n}))$. If $D_{i_j} = 0$, then $ST_{i_j} = 0$. By substituting ST_{i_j} by 0 we get:
   $OS_i = (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge 0 \bigwedge ST_{i_{j+1}} \bigwedge \cdots ST_{i_n}) \bigvee \cdots \bigvee ((OS_{i_1} \bigvee \overline{D_{i_1}} \bigvee LP_{i_1}) \bigwedge \cdots ST_{i_{j-1}} \bigwedge 0 \bigwedge ST_{i_{j+1}} \bigwedge \cdots ST_{i_n})$
   $\bigvee \cdots \bigvee (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge (OS_{i_j} \bigvee 1 \bigvee LP_{i_j}) \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n})$

   $\bigvee \cdots \bigvee (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge 0 \bigwedge ST_{i_{j+1}} \cdots \bigwedge (OS_{i_n} \bigvee \overline{D_{i_n}} \bigvee LP_{i_n}))$
   $= (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n})$. So, the new clause is:

   $((ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n}) \bigvee \overline{D_i})$ which is the link given above.

6. If *OSonly*$(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and for some $j \in \{1..n\}$, $ST_{i_j} = 0$, $LP_{i_j} = 0$ and $OS_{i_j} = 0$:

   The link states that C_i can be deleted provided it has only one subclass (since it has local properties). If we know that one subclass (still unmarked), will have more than one subclass (indicated by $ST_{i_j} = 0$ and $OS_{i_j} = 0$), and has no local properties (therefore could be deleted), when we delete C_i (if we do so) will prevent us from deleting C_{i_j}, since C_{i_j} will have local properties migrated to it, and it has more than one subclass (indicated by $OS_{i_j} = 0$). Therefore, we can substitute the above link the two following links:

   STonly$(C_i, \{C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_n}\})$

   $minimalRemaining(C_i, \{C_{i_j}\})$

   Proof: The original clause is: $(OS_i \bigvee \overline{D_i})$, where
   $OS_i = (ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee ((OS_{i_1} \bigvee \overline{D_{i_1}} \bigvee LP_{i_1}) \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n})$

$\bigvee \cdots \bigvee (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge (OS_{i_j} \bigvee \overline{D_{i_j}} \bigvee LP_{i_j}) \bigwedge ST_{i_{j+1}} \bigwedge \cdots \bigwedge ST_{i_n})$

$\bigvee \cdots \bigvee (ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge (OS_{i_n} \bigvee \overline{D_{i_n}} \bigvee LP_{i_n}))$. By substituting ST_{i_j} by 0, LP_{i_j} by 0 and OS_{i_j} by 0, we get:
$OS_i = (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge 0 \bigwedge ST_{i_{j+1}} \bigwedge \cdots ST_{i_n}) \bigvee \cdots \bigvee ((OS_{i_1} \bigvee \overline{D_{i_1}} \bigvee LP_{i_1}) \bigwedge \cdots ST_{i_{j-1}} \bigwedge 0 \bigwedge ST_{i_{j+1}} \bigwedge \cdots ST_{i_n})$

$\bigvee \cdots \bigvee (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge (0 \bigvee \overline{D_{i_j}} \bigvee 0) \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n})$

$\bigvee \cdots \bigvee (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge 0 \bigwedge ST_{i_{j+1}} \cdots \bigwedge (OS_{i_n} \bigvee \overline{D_{i_n}} \bigvee LP_{i_n}))$

$= (ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge \overline{D_{i_n}} \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n})$. So, the new clause is:

$((ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge \overline{D_{i_n}} \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n}) \bigvee \overline{D_i})$, which can be decomposed into:

$((ST_{i_1} \bigwedge \cdots ST_{i_{j-1}} \bigwedge ST_{i_{j+1}} \cdots \bigwedge ST_{i_n}) \bigvee \overline{D_i}) \bigwedge (\overline{D_{i_n}} \bigvee \overline{D_i})$, which corresponds to the two links given above.

7. If $NPonly(C_i, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$ 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: remove the link.

   Proof: By substituting D_i by 0 in the original clause: $(NP_i \bigvee \overline{D_i}) = (NP_i \bigvee 1) = 1$. That means that the clause is satisfied no matter what the value of NP_i is. Therefore the link can be removed.

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

   The link states that C_i can be deleted (since it has no local properties) but it can affect deletion of superclasses, in case it was possible that properties were migrated to it. If a direct-superclass will not be deleted, we know that class C_i will not get any properties migrated from C_{i_j}. However, we still have to account for the other superclasses. Therefore, change link to:

   $NPonly(C_i, \{C_{i_{n+1}}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_{n+m}}\})$

   Proof: The original clause is: $(NP_i \bigvee \overline{D_i})$, where
   $NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge (\overline{D_{i_j}} \bigvee NP_{i_j}) \bigwedge \cdots \bigwedge (\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$. By substituting D_{i_j} by 0, we get:
   $NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge (1 \bigvee NP_{i_j}) \bigwedge \cdots \bigwedge (\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$

   $= \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge (\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$. So, the new clause is:

   $(NP_i \bigvee \overline{D_i})$, where NP_i is the clause computed above. This clause is equivalent to the link given above.

9. If $NPonly(C_i, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$ and for some $j \in \{n+1..n+m\}$ $NP_{i_j} = 0$:

   The link states that C_i can be deleted (since it has no local properties) but it can affect deletion of superclasses, in case it was possible that properties were migrated to it. If the NP of a direct-superclass is 0, it means that the superclass, if deleted, will migrate the properties to C_i, and therefore prevent C_i from being deleted. So, we can substitute the link above by:

   $NPonly(C_i, \{C_{i_{n+1}}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_{n+m}}\})$
   $minimalRemaining(C_i, \{C_{i_j}\})$

   Proof: The original clause is: $(NP_i \bigvee \overline{D_i})$, where
   $NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge (\overline{D_{i_j}} \bigvee NP_{i_j}) \bigwedge \cdots \bigwedge (\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$. By substituting NP_{i_j} by 0, we get:
   $NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge (\overline{D_{i_j}} \bigvee 0) \bigwedge \cdots \bigwedge (\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$

$= \overline{LP_i} \bigwedge(\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge(\overline{D_{i_j}}) \bigwedge \cdots \bigwedge(\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$. So, the new clause can be rewritten as:
$(\overline{LP_i} \bigwedge(\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge(\overline{D_{i_{j-1}}} \bigvee NP_{i_{j-1}}) \bigwedge \cdots \bigwedge(\overline{D_{i_{j+1}}} \bigvee NP_{i_{j+1}}) \bigwedge \cdots \bigwedge(\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}}) \bigwedge \overline{D_{i_j}}) \bigvee \overline{D_i})$
which is the same as:
$(\overline{LP_i} \bigwedge(\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge(\overline{D_{i_{j-1}}} \bigvee NP_{i_{j-1}}) \bigwedge \cdots \bigwedge(\overline{D_{i_{j+1}}} \bigvee NP_{i_{j+1}}) \bigwedge \cdots \bigwedge(\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})) \bigvee \overline{D_i})$

$\bigwedge(\overline{D_{i_j}} \bigvee \overline{D_i})$ which corresponds to the two links given above.

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*:

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

    Proof: By substituting D_i by 0 in the original clause:

    $((ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee NP_i \bigvee \overline{D_i}) = ((ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee NP_i \bigvee 1) = 1$. That means that the clause is satisfied no matter what the values of $ST_{i_1}, ST_{i_2}, \ldots, ST_{i_n}$ and NP_i are. Therefore the link can be removed.

11. 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_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. The link also states that one of the subclasses of C_i will not be deleted, and that is why we have ST instead of OS. Therefore, if one more subclass will not be deleted, C_i will have at least 2 subclasses in the remeaining schema. And so, it can only be deleted if it does not get any property migrated to it. Consequently, we change the link to:

    $NPonly(C_i, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$

    Proof: The original clause is: $(ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee NP_i \bigvee \overline{D_i})$. If $D_j = 0$, then $ST_j = 0$. By substituting ST_j by 0 in the above clause, we get: $(0 \bigvee NP_i \bigvee \overline{D_i}) = (NP_i \bigvee \overline{D_i})$ which is the above link.

12. 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_j} is *non-deletable* for some $j \in \{n+1..n+m\}$:

    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, in case it was possible that properties were migrated to it. If a direct-superclass will not be deleted, we know that class C_i will not get any properties migrated from C_{i_j}. However, we still have to account for the other superclasses. Therefore, we change link to:

    $STorNP(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_{n+m}}\})$

    Proof: The original clause is: $(ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee NP_i \bigvee \overline{D_i})$ where
    $NP_i = \overline{LP_i} \bigwedge(\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge(\overline{D_{i_j}} \bigvee NP_{i_j}) \bigwedge \cdots \bigwedge(\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$. By substituting D_{i_j} by 0, we get:
    $NP_i = \overline{LP_i} \bigwedge(\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge(1 \bigvee NP_{i_j}) \bigwedge \cdots \bigwedge(\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$

    $= \overline{LP_i} \bigwedge(\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots \bigwedge(\overline{D_{i_{n+m}}} \bigvee NP_{i_{n+m}})$. So, the new clause is:

    $(ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee NP_i \bigvee \overline{D_i})$, where NP_i is the clause computed above. This clause is equivalent to the link given above.

    note: if in the original link there was only one superclass, i.e. $m = 1$, so $j = n + 1$ then we could remove the link.

Proof: The original clause is: $(ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee NP_i \bigvee \overline{D_i})$ where $NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}})$. By substituting D_{i_j} by 0 and LP_i by 0 (since C_i does not have any local property up to the moment, otherwise the link *STonly* would have been generated instead), we get: $NP_i = 1 \bigwedge (1 \bigvee NP_{i_{n+1}}) = 1$. Therefore, the clause $(OS_i \bigvee NP_i \bigvee \overline{D_i}) = (ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee 1 \bigvee \overline{D_i}) = 1$. Therefore the clause is true no matter what the values of ST's and D_i are. So we can remove the link.

13. If $STonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ 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: remove the link.

    Proof: By substituting D_i by 0 in the original clause:

    $((ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee \overline{D_i}) = ((ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee 1) = 1$. That means that the clause is satisfied no matter what the values of $ST_{i_1}, ST_{i_2}, \ldots, ST_{i_n}$ are. Therefore the link can be removed.

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

    The link states that C_i can be deleted provided all the subtrees of classes $C_{i_1}, C_{i_2}, \ldots, C_{i_n}$ are also deleted. If one of this classes is non-deletable (C_{i_j}), then C_i can not be deleted. Therefore: mark C_i *non-deletable*.

    Proof: The original clause is: $(ST_{i_1} \bigwedge ST_{i_2} \bigwedge \cdots \bigwedge ST_{i_n}) \bigvee \overline{D_i})$. If $D_j = 0$, then $ST_j = 0$. By substituting ST_j by 0 in the above clause, we get: $(0 \bigvee \overline{D_i})$. Therefore, the only way to satisfy the above clause is by making $D_i = 0$.

15. If $remainPropagate(C_i, \{C_{i_1}, C_{i_2}, \ldots C_{i_n}\})$ 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 $C_{i_1}, \ldots, C_{i_n}$ should remain. Therefore, this link is translated into a link that represent this dependency between classes $C_{i_1}, \ldots, C_{i_n}$ and does not include class C_i: $minimalRemaining(C_{i_1}, \{C_{i_2}, \ldots, C_{i_n}\})$

    Proof: By substituting D_i by 0 in the original clause:
    $(D_i \bigvee \overline{D_{i_1}} \bigvee \overline{D_{i_2}} \bigvee \cdots \bigvee \overline{D_{i_n}}) = (0 \bigvee \overline{D_{i_1}} \bigvee \overline{D_{i_2}} \bigvee \cdots \bigvee \overline{D_{i_n}}) = (\overline{D_{i_1}} \bigvee \overline{D_{i_2}} \bigvee \cdots \bigvee \overline{D_{i_n}})$ which is the *minimalRemaining* link given above. Note that, the *minimalRemaining* link corresponds to an or-ed clause of negated D_i's, and therefore, it would be also correct to transform the link to: $minimalRemaining(C_{i_2}, \{C_{i_1}, C_{i_3}$
    for example.

    note: a special case happens when there is only one class in the destination set (say C_{i_1}). Since at least one of the classes in the destination set should remain and C_{i_1} is the only one, C_{i_1} should remain. Therefore: C_{i_1} is *non-deletable*.

    Proof: By substituting D_i by 0 in the original clause: $(D_i \bigvee \overline{D_{i_1}}) = (0 \bigvee \overline{D_{i_1}}) = (\overline{D_{i_1}}) \Rightarrow D_{i_1} = 0$.

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

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

    Proof: by substituting D_{i_j} by 0 in the original clause:

    $(D_i \bigvee \overline{D_{i_1}} \bigvee \cdots \overline{D_{i_j}} \bigvee \cdots \bigvee \overline{D_{i_n}}) = (D_i \bigvee \overline{D_1} \bigvee \cdots \overline{D_{i_{j-1}}} \bigvee 1 \bigvee \overline{D_{i_{j+1}}} \bigvee \cdots \bigvee \overline{D_{i_n}}) = 1$. That means that the clause is satisfied no matter what the values of $D_i, D_{i_1}, \ldots, D_{i_{j-1}}, D_{i_{j+1}}, \ldots, D_{i_n}$ are. Therefore, we can remove the clause/link.

17. If $minimalRemaining(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and C_i is *non-deletable*:

    The link states that if C_i is deleted, at least one of the classes $C_{i_1}, C_{i_2}, ..., C_{i_n}$ should remain. Since C_i will not be deleted, the information in the link is not needed. Therefore: remove the link.

Proof: by substituting D_i by 0 in the original clause: $(\overline{D_i} \bigvee \overline{D_{i_1}} \bigvee \cdots \overline{D_{i_n}}) = (1 \bigvee \overline{D_{i_1}} \bigvee \cdots \overline{D_{i_n}}) = 1$. That means that the clause is satisfied no matter what the values of $D_{i_1}, \ldots, D_{i_n}$ are. Therefore, we can remove the clause/link.

18. If $minimalRemaining(C_i, \{C_{i_1}, C_{i_1}, \ldots, C_{i_1}\})$ and there exists one $i, i \in \{1..n\}$ such that C_{i_j} is *non-deletable*:

    The link states that if C_i is deleted if at least one of the classes $C_{i_1}, C_{i_2}, ..., C_{i_n}$ should remain. Since C_{i_j} will remain, there is no restriction on the deletion of C_i with respect to the dependency represented in this link. Therefore: remove the link.

    The proof is the same as the previous one, except that we substitute D_{i_j} by 0 instead of substituting D_i by 0.

Transformation Rules in case a class is *deletable*:

We will give rules for deletion of classes, only considering the case where the remaining set of links, after marking the given class *deletable*, are guaranteed to have an assignment, namely, all classes being mark *non-deletable*, as discussed in Section 7.5. If we do not guarantee a feasible assignment, there is no reason for giving transformation rules. Instead, we would try out all possible combinations and check validity by substituting the values of the D variables in the corresponding clauses.

19. 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 *deletable*:

    The link states that C_i can be deleted if will always have only one subclass ($OS_i = 1$) or if will never have any properties migrated to it ($NP_i = 1$). Guaranteeing that there will be only one subclass is impossible in the general case, because there might be some consistency requirement that is not met when deleting the other subclasses. So, we want to guarantee $NP_i = 1$. Therefore, we mark every class in $\{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\}$ as *non-deletable*, unless the corresponding NP_{i_j} is 1.

    Proof: By substituting D_i by 1 in the original clause: $(OS_i \bigvee NP_i \bigvee \overline{D_i}) = (OS_i \bigvee NP_i \bigvee 0) = (OS_i \bigvee NP_i)$. Since we cannot guarantee OS_i, we guarantee NP_i. NP_i is given by: $NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots$ Therefore, we need $D_{i_j} = 0$ or $NP_{i_j} = 1, \forall j \in \{n+1..n+m\}$, which is the condition stated above.

20. 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 *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 one subclass is deleted, as an effect of this deletion, its direct-subclasses will be connected to class C_i. Therefore, we substitute the link by:

    $OSorNP(C_i, \{C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j_1}}, \ldots, C_{i_{j_n}}, C_{i_{j+1}}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$

    where $C_{i_{j_1}}, \ldots, C_{i_{j_n}}$ are the direct-subclasses of C_{i_j}.

21. 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 *deletable* for some $j \in \{n+1..n+m\}$:

    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 one superclass is deleted, as a side effect, some properties might be migrated to C_i. In this is the case, we change the link to:

    $OSonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$

    If this is not the case (i.e., no properties migrated), we change the link to:

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

    where $C_{i_{j_{n+1}}}, \ldots, C_{i_{j_{n+m}}}$ are the direct-superclasses of C_{i_j}.

22. If $OSonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and C_i is *deletable*:

    The link states a condition for C_i to be deleted. Since we cannot guarantee that $OS_i = 1$. We do not give this transformation rule.

23. If $OSonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and C_{i_j} is *deletable* for some $j \in \{1..n\}$:

    The link states a condition for C_i to be deleted. If one subclass is deleted, as an effect of this deletion, its direct-subclasses will be connected to class C_i. Therefore, we substitute the link by:

    $OSonly(C_i, \{C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j_1}}, \ldots, C_{i_{j_n}}, C_{i_{j+1}}, \ldots, C_{i_n}\})$

    where $C_{i_{j_1}}, \ldots, C_{i_{j_n}}$ are the direct-subclasses of C_{i_j}.

24. If $NPonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and C_i is *deletable*:

    The link states that C_i can be deleted if it will never have any properties migrated to it ($NP_i = 1$). Therefore, we mark every class in $\{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\}$ as *non-deletable*, unless the corresponding NP_{i_j} is 1.

25. If $NPonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and C_{i_j} is *deletable* for some $j \in \{n+1..n+m\}$: The link states that C_i can be deleted if it will never have any properties migrated to it ($NP_i = 1$).

    The link states that C_i can be deleted if it will never have any properties migrated to it ($NP_i = 1$). If one superclass is deleted, as a side effect, some properties might be migrated to C_i. In this is the case, we can no longer delete C_i. Therefore: mark C_i as *non-deletable*.

    If this is not the case (i.e., no properties migrated), we change the link to:

    $NPonly(C_i, \{C_{i_{n+1}}, \ldots, C_{i_{j-1}}, C_{i_{j_{n+1}}}, \ldots, C_{i_{j_{n+m}}}, C_{i_{j+1}}, \ldots, C_{i_{n+m}}\})$

    where $C_{i_{j_{n+1}}}, \ldots, C_{i_{j_{n+m}}}$ are the direct-superclasses of C_{i_j}.

26. 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 *deletable*:

    The link states that C_i can be deleted if all subtrees of classes $C_{i_1}, C_{i_2}, \ldots, C_{i_n}$ are deleted, or if C_i will never have any properties migrated to it ($NP_i = 1$). Guaranteeing that there all the subtrees will be deleted is impossible in the general case, because there might be some consistency requirement that is not met when deleting the other subclasses. So, we want to guarantee $NP_i = 1$. herefore, we mark every class in $\{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\}$ as *non-deletable*, unless the corresponding NP_{i_j} is 1.

    Proof: By substituting D_i by 1 in the original clause: $(OS_i \bigvee NP_i \bigvee \overline{D_i}) = (OS_i \bigvee NP_i \bigvee 0) = (OS_i \bigvee NP_i)$. Since we cannot guarantee OS_i, we guarantee NP_i. NP_i is given by: $NP_i = \overline{LP_i} \bigwedge (\overline{D_{i_{n+1}}} \bigvee NP_{i_{n+1}}) \bigwedge \cdots$ Therefore, we need $D_{i_j} = 0$ or $NP_{i_j} = 1, \forall j \in \{n+1..n+m\}$, which is the condition stated above.

27. 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_j} is *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 *STonly* instead), but it can affect deletion of super and subclasses. If one subclass is deleted, as an effect of this deletion, its direct-subclasses will be connected to class C_i. Therefore, we substitute the link by:

    $STorNP(C_i, \{C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j_1}}, \ldots, C_{i_{j_n}}, C_{i_{j+1}}, \ldots, C_{i_n}\}, \{C_{i_{n+1}}, C_{i_{n+2}}, \ldots, C_{i_{n+m}}\})$

    where $C_{i_{j_1}}, \ldots, C_{i_{j_n}}$ are the direct-subclasses of C_{i_j}.

28. 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_j} is *deletable* for some $j \in \{n+1..n+m\}$:

    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 one superclass is deleted, as a side effect, some properties might be migrated to C_i. In this is the case, we change the link to:

    $STonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$

    If this is not the case (i.e., no properties migrated), we change the link to:

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

    where $C_{i_{j_{n+1}}}, \ldots, C_{i_{j_{n+m}}}$ are the direct-superclasses of C_{i_j}.

29. If $STonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and C_i is *deletable*:

    The link states a condition for C_i to be deleted. Since we cannot guarantee that alll subtrees will be deleted, this clause might be violated later on. So, we do not give this transformation rule.

30. If $STonly(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and C_{i_j} is *deletable* for some $j \in \{1..n\}$:

    The link states a condition for C_i to be deleted. If one subclass is deleted, as an effect of this deletion, its direct-subclasses will be connected to class C_i. Therefore, we substitute the link by:

    $STonly(C_i, \{C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j_1}}, \ldots, C_{i_{j_n}}, C_{i_{j+1}}, \ldots, C_{i_n}\})$

    where $C_{i_{j_1}}, \ldots, C_{i_{j_n}}$ are the direct-subclasses of C_{i_j}.

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

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

    Proof: by substituting D_i by 1 in the original clause: $(D_i \bigvee \overline{D_{i_1}} \bigvee \overline{D_{i_2}} \bigvee \cdots \bigvee \overline{D_{i_n}}) = (1 \bigvee \overline{D_{i_1}} \bigvee \overline{D_{i_2}} \bigvee \cdots \bigvee \overline{D_{i_n}}) = 1$. That means that the clause is satisfied no matter what the values of $D_{i_1}, \ldots, D_{i_n}$ are. Therefore, we can remove the link.

32. If $remainPropagate(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and there exists one $j, j \in \{1..n\}$ such that C_{i_j} is *deletable*:

    The link states that if C_i is not *deletable*, at least one of the classes $C_{i_1}, C_{i_2}, \ldots, C_{i_n}$ should remain. In order to maintain consistency, we are still dependent on the deletion of the classes $C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_n}$. Therefore: we change the link to:

    $remainPropagate(C_i, \{C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_n}$

    Proof: by substituting D_{i_j} by 0 in the original clause:

    $(D_i \bigvee \overline{D_{i_1}} \bigvee \cdots \overline{D_{i_j}} \bigvee \cdots \bigvee \overline{D_{i_n}}) = (D_i \bigvee \overline{D_{i_1}} \bigvee \cdots \overline{D_{i_{j-1}}} \bigvee 0 \bigvee \overline{D_{i_{j+1}}} \bigvee \cdots \bigvee \overline{D_{i_n}}) =$

    $(D_i \bigvee \overline{D_{i_1}} \bigvee \cdots \overline{D_{i_{j-1}}} \bigvee \overline{D_{i_{j+1}}} \bigvee \cdots \bigvee \overline{D_{i_n}})$ which is the *remainPropagate* link given above.

33. If $minimalRemaining(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and C_i is *deletable*:

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

    $minimalRemaining(C_{i_1}, \{C_{i_2}, \ldots, C_{i_n}\}$.

    Proof: by substituting D_i by 0 in the original clause:

    $(\overline{D_i} \bigvee \overline{D_{i_1}} \bigvee \cdots \bigvee \overline{D_{i_n}}) = (0 \bigvee \overline{D_{i_1}} \bigvee \cdots \bigvee \overline{D_{i_n}}) = (\overline{D_{i_1}} \bigvee \cdots \bigvee \overline{D_{i_n}})$ which is the *minimalRemaining* link given above.

34. If $minimalRemaining(C_i, \{C_{i_1}, C_{i_2}, \ldots, C_{i_n}\})$ and there exists one $j, j \in \{1..n\}$ such that C_{i_j} is *deletable*:

    The link states that if C_i is deleted, at least one of the classes $C_{i_1}, C_{i_2}, \ldots, C_{i_n}$ should remain. In order to maintain consistency, we are still dependent on the deletion of the classes $C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_n}$. Therefore: we change the link to:

    $minimalRemaining(C_i, \{C_{i_1}, \ldots, C_{i_{j-1}}, C_{i_{j+1}}, \ldots, C_{i_n}$

    The proof is the same as the previous one, except that we substitute D_{i_j} by 1, instead of substituting D_i.