A Transparent Object-Oriented Schema Change Approach Using View Evolution[1]

Young-Gook Ra and Elke A. Rundensteiner

Software Systems Research Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109–2122
e-mail: ygra@eecs.umich.edu, rundenst@eecs.umich.edu

April, 1994

Abstract

When a database is shared by many users, updates to the database schema are almost always prohibited because there is a risk of making existing application programs obsolete when they run against the modified schema. This paper addresses the problem by integrating schema evolution with view facilities. Each user is assigned his or her own database view, and develops application programs against the view. When new requirements necessitate schema updates for a particular user, then the user specifies schema changes to his personal view rather than to the shared base schema. Our view schema evolution approach then computes a new view schema that reflects the semantics of the desired schema change, and replaces the old view with the new one. This approach provides the means for schema change without affecting other views (and thus without affecting existing application programs). The persistent data is shared by different views of the schema, i.e. by both old as well as newly developed applications can continue to interoperate. To realize our approach, this paper identifies key features of how views must be extended to be *capacity-augmenting*. We then present algorithms that implement the set of typical schema evolution operations as view definitions. Throughout the paper we present examples that demonstrate our approach.

Index Terms — Schema Evolution, Object-Oriented View System, Interoperability, Capacity-augmenting Views, Object-Oriented Database.

[1]This work was supported in part by the NSF RIA grant #IRI-9309076 and the University of Michigan Faculty Award Program.

TABLE OF CONTENTS

1 Introduction

Views have been successful in relational databases. They allow a programmer to restructure the database to meet the specific needs of an application without affecting other application programs. Other advantages of relational views are data independence, access control and security. Perceiving the importance of views, many researchers have begun to investigate their counterpart in the object-oriented world [1, 6, 9, 24, 22, 19, 21]. The diverse view mechanisms proposed typically provide the functionality to restructure a base schema by hiding classes, by adding classes, by customizing the behavior or extent of classes, and by rearranging the generalization hierarchy. However, since views correspond to derived data computed based on data stored persistently in database, they by definition do not support the addition of new stored information to the database.

Together with views, schema evolution has been recognized as an essential feature of OODBs required to deal with the evolving nature of typical OODB applications such as CAD/CAM, VLSI design and office information systems. To measure the frequency of schema evolution, a health management system was observed over a 18 month period [26]. It shows that the number of relations in one study increased by 139%, the number of attributes by 274% and that every relation has been changed. This measurement confirms the need for tools and techniques for managing schema changes to a database. Similarly, another report [12] confirms that data models are typically not stable as we expect based on the observation of seven typical database applications such as project tracking, real estate inventory, accounting and sales management, etc.. These project observation showsthat about 59% of attributes are changed on the average. In summary, schema evolution is a pervasive problem in many large systems, and there certainly is a need for the capability of managing the evolution of a schema without any service interruption [25].

There is active research on schema evolution [4, 2, 10, 14, 11, 15, 29, 8, 31, 32]. Most work focuses on the modification of a database schema and the corresponding migration of database instances from the old to the modified schema. However, this direct change on the schema may modify existing views and the application programs written against the views can no longer run. Typically, this problem is not properly handled in most conventional systems.

We propose that many advantages could be gained by integrating these two capabilities of view support and schema evolution into one unified mechanism. This unique integration would allow for the continued functioning of programs written against the original version of the database while allowing for schema change required to handle the demands of new programs. This is based on the data independence property gained by view schemas, and the extensibility of the database achieved by schema evolution. Extending OODBs with these two capabilities would provide the following advantages: first, a view mechanism would enable them to provide programs with customized data representations, and second, schema evolution integrated with view support would provide extensibility of the system to allow for the incorporation of new data as required by new applications.

The basic principle of our view change approach is that, given a schema change request on a view schema, the system — rather than modifying the view schema in place – computes a new view which reflects the semantics of the schema change. The new view is assigned to the user, while the old one is maintained by the system as long as other application programs continue to operate on it.

Thus, our approach keeps the old versions of a schema instead of modifying them; achieving schema versioning using a view system. Note that in our approach, unlike in other systems [8, 7, 11], the scope of a schema version is not confined to the objects which have been created under this particular schema version. Instead any object in the database is able to be accessed and modified through any version of the schema, assuming the classes of the objects are included in the view schema. We achieve this for two reasons: (1) since all objects are associated with a single underlying global schema and (2) since each version of the schema is implemented via a view defined on the global schema.

To simulate schema evolution using view mechanisms, the view system must be capable of generating views that augment the capacity of the underlying schema in addition to restructuring and restricting it. In addition, views require multiple classification support in the underlying object model because instances of a new virtual class need to be classified both as members of the virtual class and as members of the original base class. In this paper, we present solutions to address these problems of the capacity-augmenting view capabilities and multiple classification object support. Finally, we show that we can simulate a comprehensive set of schema change operations by creating

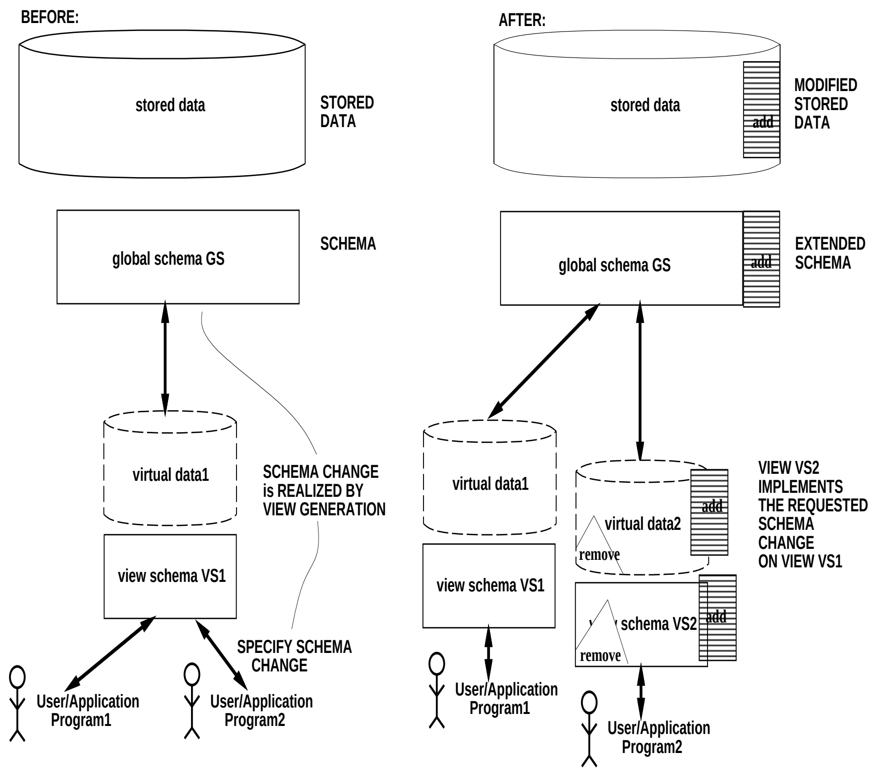

Figure 1: Overall Description of Our Transparent Schema Evolution (TSE) Approach.

a new view schema which reflects the semantics of schema change.

In Section 2, we give an overview of our basic view change approach and identify view system requirements. In Section 3, we discuss the extensions to a view system for our approach. Section 4 presents the implementation of the object model having all required features. In Section 5, we give the overall architecture of our system. To demonstrate the feasibility of our approach, we present algorithms in Section 6 for translating schema change requests into view specifications. In Section 7, we discuss the version capabilities of our approach. Related research is discussed in Section 7, and conclusions are presented in Section 8.

2 The Basic View Schema Change Approach and its Requirements

2.1 TSE: The Basic Transparent Schema Evolution Approach

In a typical database development environment, a developer must consult with others to figure out the impact of a requested schema change on the existing application programs. Thus, the decision process of even a small schema change is likely to be long. In current OODB systems, if the schema change does impact existing programs, we typically have two choices: (1) we can rewrite all affected application programs to work with the new schema, or (2) we can simply reject the upgrade of the schema. The former would be an extremely labor intensive and costly endeavor, especially if the programs are written in an older programming language and/or are not well-documented. The later is also undesirable, since it may result in the system not being able to take advantage of important upgrades in the form of new technology or new algorithmic developments.

To overcome these problems, we propose a transparent schema evolution (TSE) approach. After constructing an initial global schema, each developer defines her own view of the shared database to serve as a customized interface to the database. In our approach, whenever a schema change need is identified by a developer, she specifies the

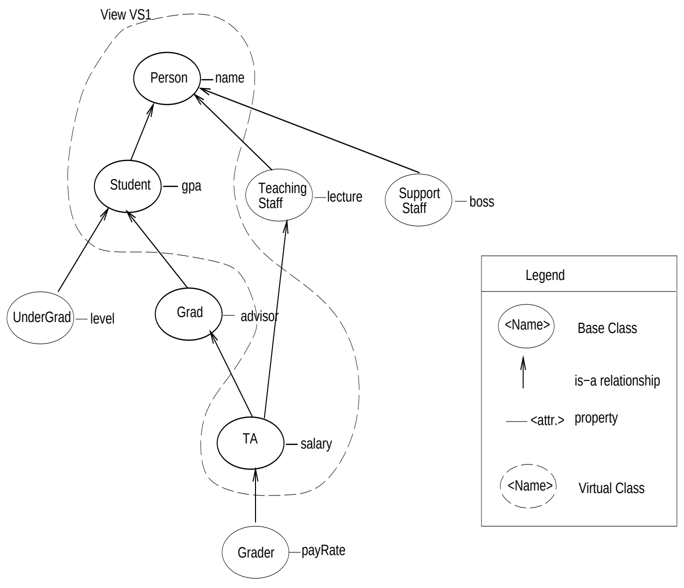

Figure 2: Example Schema: University Database.

necessary schema change on her own view (in Figure 1 left). The database system computes a new view schema, which reflects the semantics of the requested schema change, and replaces the old view with the new one (Figure 1 right). In short, rather than directly modifying the old schema, we will 'simulate' the requested schema change using a view schema (i.e., the schema change is performed virtually using the view mechanism). This process should be transparent to the user. In particular, she should not be able to distinguish beween this virtual schema change and the direct schema modification. While the user perceives that an actual change occured in place, Figure 1 shows the underlying changes in the database system.

While the view schema and its associated virtual database are adjusted exactly as requested, the global schema and its global database may not be. Note that even though the schema change is virtual, the global schema must sometimes be restructured to generate the view schema with the changed semantics. Database reorganization at the instance level is required – especially when the change is capacity augmenting [2]. In Figure 1, we see that the global schema is augmented for instance with new stored data, when the addition of a new instance variable was requested through a view change. However, when a delete is requested, it results in a removal from the view (and its associated virtual data) but not necessarily a delete from the global database.

2.2 A Comprehensive Example

The overall approach is best explained by giving an example using the university database in Figure 2. Assume one of the developers builds the view schema in Figure 3 (a) to serve as a customized database interface for further development. This view construction is depicted in Figure 2 by the dotted line. After several application programs have been written against the view, he finds that each student object should also carry *register* information for recording the registration status of each student. Figure 3 (b) shows the modified view schema that he now wants. The dotted ellipses represent virtual classes, whereas the solid ellipses represent base classes. In traditional systems, the developer would have to figure out the potential impact of the schema change on other developers' programs. In our system, the developer simply specifies the desired schema change directly to the view, in this case *"add attribute register to **Student** class"*, without going through all the irritating procedures explained above.

This requested change would be realized as follows by our system. After the user specifies the add attribute command, first, the system generates the two virtual classes **Student'** and **TA'** that refine the **Student** and **TA** classes, respectively, with the new attribute *register* (Figure 3 (c)). Second, the two virtual classes are integrated

[2]The capacity augmenting change is defined to be the schema restructuring to enhance the information contents of the schema

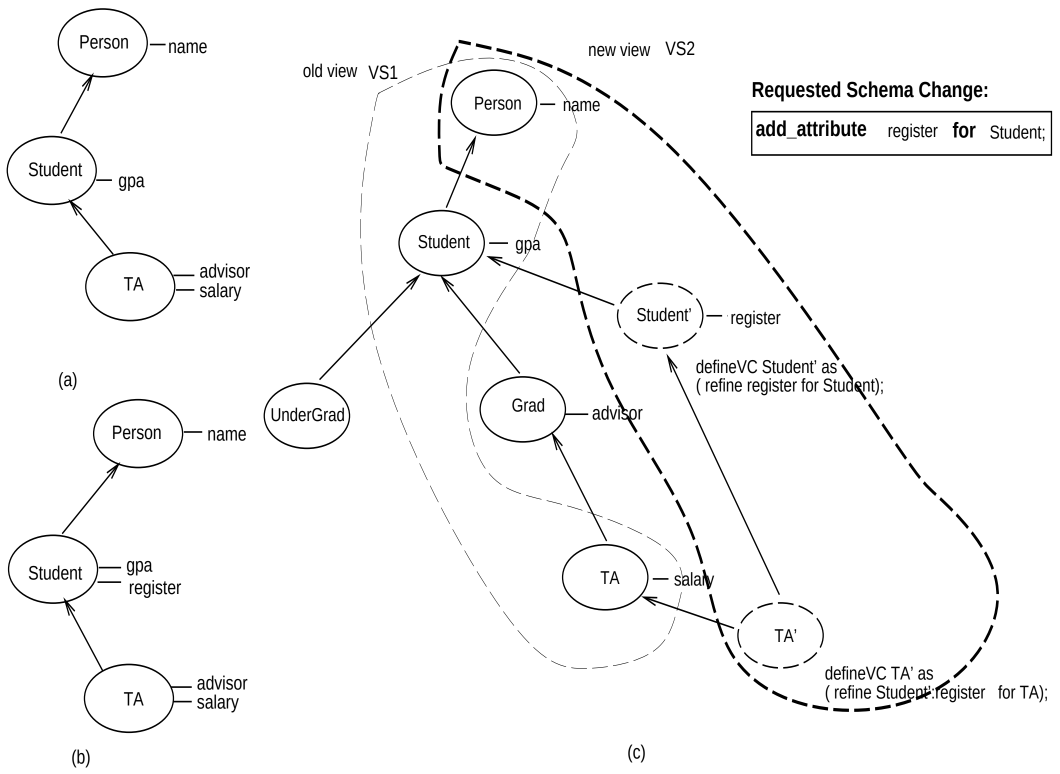

Figure 3: Schema Change for Adding an Attribute.

into the global schema by the classification algorithm as shown in Figure 3 (c) [17]. Third, the system selects the classes **Person**, **Student'** and **TA'** for the new view. It then renames the **Student'** and **TA'** classes to **Student** and **TA** within the context of the view, respectively. Fourth, a new view schema VS2 is generated from the selected classes by running the schema generation algorithm [21]. At last, the system replaces the old view with the newly generated view. Because all the above procedures are transparent to the schema change specifier, she will have the perception that she has actually modified her original schema.

Note that the above algorithm runs in the context of a view, so it only creates virtual classes for all subclasses of a class *C within a view*. Hence, other classes of the global schema, such as the **Grad** class, would not be affected in our approach. This avoids the creation of unnecessary virtual classes for a possibly deep global schema class hierarchy.

2.3 Requirements for Realizing the Transparent Schema Change Approach

In order to achieve the transparent schema change appraoch outlined above, the following requirements must be imposed onto our view evolution approach:

- The schema change on a view should *not affect any of the existing views* which still might have application programs running on them. This assures that old applications can continue to function properly, and can in fact inter-operate with new programs. This feature would allow the user to restructure her own view *independently* from a database administrator or other users working on the same global schema.
- The newly computed view schema has to be *updatable*. Many application tools, such as for instance design and manufacturing tools, frequently need to update the database, hence the updatability of the view is an essential requirement for many advanced applications.

- Both old and new versions of a schema must be able to share the same (persistent) data, independently from through which schema they were originally created. Otherwise old and new programs would not be able to operate on and interchange the same data, i.e., *interoperability* of these applications would not be guaranteed.
- The schema change has to be *transparent* to the view user in terms of two perspectives. One, the user should not need to know whether she is dealing with base classes or virtual classes. Two, the fact that the schema change is virtual (i.e., a new view is assigned whenever possible) rather than a real change of the global database shoud not be apparent to the user.
- The views mechanism underlying our schema evolution system must be capable of generating views that augment the capacity of the schema in addition to restructuring and restricting it. Because current OODB systems do not provide such *capacity-augmenting* view mechanisms, we need to develop such a view system as foundation for our schema change prototype. This requires the investigation of the following two tasks: first, we need to extend the view definition language with schema-augmenting operators, and second, we need to provide a flexible architectural framework for supporting the dynamic augmentation of the schema and also of the database.
- When a virtual class is derived from a base class, instances of the virtual class need to be classified both as members of the new virtual class and the old base class. In the context of capacity-augmenting views, we thus need to provide *multiple classification* support at the object model level as well as *dynamic restructuring* support at the data representation level.

In this paper, we detail a view change approach that successfully addresses all of these issues.

3 Extending MultiView System for Transparent Schema Changes

In this paper, we assume a typical OO data model, such as COCOON's [28], MultiView's [22] and Orion's [4]. A list of basic terms of OO data models and views are found in the appendix.

3.1 MultiView: The Underlying Object-Oriented View Support System

Since our view schema evolution approach is based on object-oriented view concepts, we describe below the view system we have developed towards the specification and maintenance of views, called MultiView [19]. Unlike other object-oriented view mechanisms, MultiView creates a complete view schema rather than just deriving individual virtual classes. Furthermore, views in MultiView allow for the insertion of new classes or the modification of existing classes in the middle of class hierarchy. These unique characteristics effectively support our view change system, which requires that a view itself is a complete schema graph in order to make the distinction between the view schema and the base schema transparent to the user. In addition, MultiView offers the following features that made it suitable as foundation for our schema change approach. First, it generates updatable views [19]. Second, several of the view specification subtasks are already automated, and can be reused in our system. Lastly, a prototype of MultiView has been implemented at the University of Michigan, and thus can be utilized as platform for constructing the TSE system.

The task of creating a view schema is divided in the following tasks in MultiView: (1) the generation of customized classes using an object-oriented query, (2) the integration of these derived classes into one consistent global schema graph, and (3) the specification of arbitrarily complex view schemata composed of base and virtual classes. For the first subtask, MultiView provides the user with an object algebra for class derivation [19]. The second subtask is automated in MultiView by the classification algorithm [17]. The integration of virtual classes into one global schema provides many benefits, including detecting identical and thus redundant classes, sharing methods without code duplication by different views, and enabling efficient view schema generation. The third subtask requires the use of a view specification language for class selection, and a view schema generation algorithm for the construction of the view generalization hierarchy [21]. Automatic view generation [21] relieves the user of constructing the is-a hierarchy for each view schema and removes the potential inconsistencies of the view generalization hierarchy due to the mistakes of the user.

3.2 Extending Object Algebra for Capacity-Augmenting Views

Since our view schema evolution approach is built using the MultiView system, we utilize MultiView's view definition language, an object algebra [19, 22, 23, 21, 20, 17, 18], as foundation of our TSE system. While a complete description of the algebra can be found elsewhere [19], below we briefly introduce the operators. We also discuss the necessary extension of some of these operators required for schema change support. The object algebra is *set-oriented* in that the inputs and outputs are sets of objects. In the context of this paper, we apply them to classes in order to derive new virtual classes.

- The **select** operator defined by (**select from** <*class*> **where** < *predicate* >) returns a subset of the input set of objects, namely those satisfying the predicate < *predicate* >. The type of the resulting set is unchanged i.e., it is equal to *type(<class>)*.
- The **hide** operator defined by (**hide** <*properties*> **from** <*class*>) removes properties listed in <*properties*> from the set of objects <*class*> while preserving all other properties defined for the type of (<*class*>). The type of the output set is a supertype of the input type, as less functions are defined on the output. All objects of the input set are also the members of the output set.
- Set operations. As the extent of classes are sets of objects, we can perform set operations as usual [22]. The criterion of duplicate elimination is object identity equality, not value equality as assumed in the relational model. We need no restriction on the operand types of set operations (ultimately, they are all objects). The result type, however, depends on the input types: For the **union** it is the lowest common supertype of the input types. The **difference** operator yields a subset of its first argument with the same type. Finally, the **intersect** operator results in the greatest common subtype. The syntax of these set operators is defined by (**set-operator** <*class1*> **and** <*class2*>).
- The **refine** operator defined by (**refine** <*property-defs*> **for** <*class*>) returns a set with the same objects as the input, but with a new type, a subtype of the input type, as all the old properties plus the new one are defined on it. We require that each property name in <*property-defs*> must be different from all existing functions defined for the type of the <*class*>.

While view definition only derives new data as a function of existing stored data, we now must also support the extension of the DB with new independent data that is not a function of existing data. In particular, the refine operator utilized for view definition in MultiView only adds derived attributes (methods) to a virtual class, we now must modify the **refine** operator to support *stored attribute* extensions. In other words, **refine** is extended such that now it may contain both kinds of properties, stored attributes and methods, in the parameter <*property-defs*>. This important extension allows for *capacity-augmenting* views, because the enhanced refining operator can create virtual classes that effectively augment the capacity of the database by adding new stored attributes to existing classes. When we refine a class with stored attributes, the representation of each object in the class has to be restructured such that the object can carry the information associated with the new stored attributes. This restructuring capability must be supported by the underlying OODB architecture. Our approach to addressing this problem is explained in Section 4.

Furthermore, we need to modify the **refine** operator defined for MultiView to facilitate inheritance of a property between virtual classes. The new syntax now is:

refine *C1:<property-name>* **for** $C2$.

This statement specifies the inheritance of a property of class $C1$ into another class $C2$. The object instances of class $C2$ then share the code block of the new property (when it is a method) defined in class $C1$ or assign a new storage for the property (when it is *a stored attribute*).

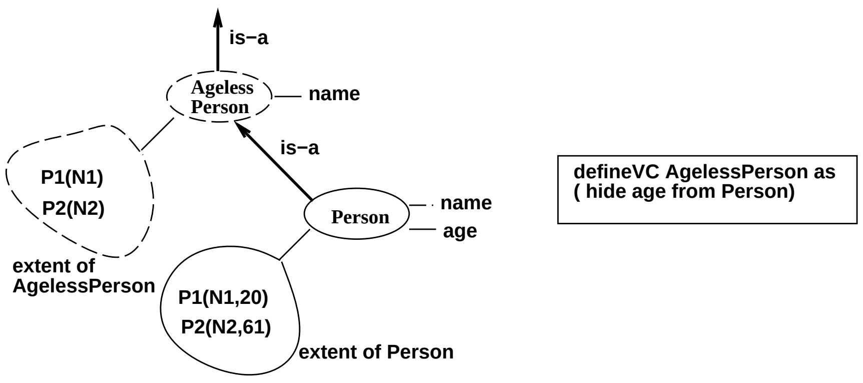

Figure 4: Example of Virtual Class Creation.

MultiView allows arbitrary queries composed by nesting these object algebra operators to serve as view definitions, exactly as in relational DBMSs: **defineVC** <*name*> **as** <*query*>. After the execution of this statement, <*name*> will appear as a persistent class of the database, just like base classes. The only difference is that the extent of the virtual class <*name*> is defined by the <*query*> expression, based on the extents of other virtual and/or base classes. In Figure 4, for example, the virtual class *AgelessPerson*, created by the ***hide*** operation, is classified as superclass of its source class *Person* because the type of *AgelessPerson* is more general and the extent is the same as that of the *Person* class[3]. In addition, the *age* attribute is hidden from the instances of *AgelessPerson*.

3.3 Generic Update Operators

In OODBs, updates are generally performed using *type-specific* update methods. We, however, want to provide a set of *generic update operations* to extend type-specific updates, similar as proposed in other view systems [24]. Such generic update operations can either be used directly or, if desired, overridden by type implementors to define type-specific methods. The generic update operations include **create** and **delete** to create and destroy objects and **set** to set attributes to new values. Additionally, we can define **add** and **remove** update operators which are applied to existing objects in order to add them to or remove them from a class. In effect, the existing object gets a new type or loses a type.

- **Create**, defined by (< *class* > **create** [< *assignment* >]), takes a class < *class* >, with member type T, and assignments of some values to attributes defined on T. An instance of T is created and added to the specified class. The properties of the new object listed are set to the given values. The class predicate is checked (if any) and the instance is also added to other classes, if appropriate. The result of the **create** operation is the newly created object.
- **Delete**, defined by (<*set-expr*> **delete**), destroys all objects returned by <*set-expr*>, i.e., they are removed from all the classes which they belong to.
- **Set**, defined by (<*set-expr*> **set** [< *assignment* >], assigns new values to the attributes of all objects returned by the <*set-expr*>. For example, (e:<*set-expr*>) **set**[salary(e) = 3,000] assigns a new salary of 3,000 to all the objects returned by <*set-expr*>.
- **Add**, defined by (<*set-expr*> **add** < *class* >), adds the objects returned by the <*set-expr*> to the class < *class* >. As a result, the objects acquire the type of < *class* >.
- **Remove**, defined by (<*set-expr*> **remove** < *class* >), removes the objects returned by the <*set-expr*> from the class < *class* >. It means that the objects lose the type of < *class* >.

[3]The extent of *AgelessPerson* is drawn with a dotted line representing that it is virtual and dependent on the extent of the source class.

Note that those generic operators can be used by type implementors to specify type-specific update methods. Then, arbitrary computations can be performed in such a method e.g., to check some constraints, to update additional information, or even to refuse the update. For example, updating a derived attribute may be implemented by either changing the underlying values of the associated base objects or by refusing the update.

3.4 Updatability of Object-Oriented Views

Similar to base classes, instances of virtual classes can be used as arguments of update operators. In our TSE system, we have to be able to update virtual objects in order to hide as much as possible the differences between virtual classes and base classes from the database user (or application programmer). It has been shown that virtual classes created by object-preserving algebra are updatable due to the one-to-one correspondence between base and virtual object instances [24]. In fact, all update operators have the same effect as if they were applied to the base class, because the virtual classes' extents are depending on the extents of base classes.

1. A **select** virtual class creates a subclass containing all objects satisfying the selection predicate. **Create**, **delete**, **add**, **remove** and **set** update operators applied to an instance of a virtual select class work on the source class. Creation, addition and setting values of objects that do not fulfill the selection predicate of the select class lead to the so-called value closure problem [6]. There are two solutions to this problem: (1) reject such creation/addition/set, or (2) allow them by inserting them into the source class or setting the value. Other update operators such as delete and remove don't cause the value-closure problem because they are applied to existing objects of the virtual class.

2. A **difference** virtual class is a subclass of the first argument class containing all objects of the first argument class that are not members of the second argument class. In general, difference is just a special case of selection where the selection predicate is just defined to be *"not a member of second class"*. **Create**, **delete**, **add**, **remove** and **set** update operators applied to an instance of a virtual difference class work on the first argument class. For further discussion, see the select operator.

3. A **union** virtual class is a superclass of its two argument classes. This case leads to ambiguity for the following updates: If we want to **create** or **add** objects to the virtual union class, we have to propagate this to at least one source class. In general, there are three choices. We can propagate it to either one of the two source classes, or to both classes. Generally, the choice depends on the context of the operation (an example will be detailed in a later section), or on the predicates associated with the source classes. The other generic update operations such as **delete**, **remove** and **set** are applicable in the obvious way: They always propagate to both source classes (if the target instances are members).

4. An **intersection** virtual class is a subclass of both argument source classes. Similar to the union virtual classes, the **remove** operations lead to ambiguity. We can remove an instance from either one of two source classes or from both. Creating, deleting, adding and setting objects should be always propagated to both source classes.

5. A **hide** virtual class is a superclass of the source class with a changed type. The **create** and **add** work as if they were applied to the source class. The only difference is that we can not assign values to the hidden attributes. We can address this problem by specifying default values in the view definition or by overriding update methods which assign the default values [4]. The **delete**, **remove** and **set** operations are propagated to the source class in a straightforward manner.

6. A **refine** virtual class is a subclass of the source class with an augmented type description. The update operations **create**, **add**, **delete** and **remove** work as if they were applied to the source class. If the **set** operation involves a refining attribute defined in the **Refine** virtual class, then the update is applied to the virtual class instead of being propagated to the source class. [5]

[4]This scheme doesn't always work especially when the hidden attributes are declared as REQUIRED.

[5]Details on the actual implementation of this approach are given in Section 4.

Notice that virtual classes may, of course, be defined based on other virtual classes.

Theorem 1. Any virtual class defined by our object algebra defined in Section 3.3 is updatable in terms of the generic update operators such as **create**, **delete**, **remove**, **add** and **set** defined in Section 3.3.

Proof: We can view the history of a virtual class derivation as a DAG with multiple roots where the roots are base classes and all other nodes are virtual classes. The edges in the DAG indicate that the class at the end point is a virtual class which is defined based on the classes at the start points of the edge. We can label the edges by the name of object algebra operator which defines the virtual class. An edge labeled as *hide*, *select* or *refine* have only one starting node, while the edge labeled as *union*, *intersect* or *difference* have two starting point classes.

Suppose we mark all the roots of the DAG as updatable, because they are all base classes. If we select all nodes (classes) of which incipient edges are all marked nodes, then all the selected classes are defined based on marked classes. In the above discussion, we have shown that the virtual classes are updatable in terms of the generic update operators if the source classes on which they are based are all updatable. As a result, all the selected classes are also updatable. So, we mark them as updatable. Repeating the above procedure, all the classes in the DAG will be marked as updatable. **q.e.d.**

The edges of the above DAG represent the *derivation* relationships between classes. If we keep the inverse of the edges, they represent the *source* relationships which indicate all the source classes on which a virtual class is based. The source relationships are useful for finding the classes to which the updates on a virtual class should be propagated. For each virtual class, following the source relationships leads to a set of base classes. They are called the *origin classes* of the virtual class. In other words, the origin classes are the base classes to which an update on the virtual class eventually propagated.

4 An Object Model Supporting Multiple Classification

4.1 Two solutions for supporting multiple classification

In Section 2, we have identified the following two requirements on the object model representation: (1) efficient dynamic restructuring of object representations and (2) multiple classification. Multiple classification is necessary in a capacity-augmenting view system, since an object may have to be an instance of different virtual classes (as well as their base class) [6]. To the best of our knowledge, current OODB systems do not support multiple classification — the only exception is IRIS [5] a functional database system that actually uses a relational database as storage structure, storing data from one object across many relations. Other OODBs typically represent an object as a chunk of contiguous storage determined at object creation time. They assume the invariant that *an object belongs to exactly one class only* — and indirectly also to all the class's superclasses.

We identify two general approaches for implementing multiple classification in OODBs: (1) the intersection-class approach and (2) the object-slicing approach [13]. These approaches are best explained via an example. Assume that given the schema in Figure 5 (a), we want to create a new car object $o1$ that is both of type *Jeep* and of type *Imported*. We cannot find a class in which to store $o1$, without violating the invariant that an object belongs to exactly one class. To resolve this dilemma, the intersection-class approach would create a new intersection class *Jeep&Imported* which is subclass of both *Jeep* and *Imported* classes. It then would create $o1$ as member of the new class (Figure 5 (b)). Next, suppose that $o1$ were already member of the *Jeep* class, and we simply wanted to dynamically reclassify it to become member of the *Imported* class. Then this would require us to create a new object $o2$ as member of the *Imported* class, to copy all attribute values from $o1$ to $o2$, and lastly to swap the object identities of these two objects. If this dynamic reclassification had the goal of $o1$ not loosing its membership in the *Jeep* class, then this dynamic reclassification of an object would again cause the creation of the *Jeep&Imported* intersection class.

The object-slicing approach (Figure 5 (c)) would implement multiple classification by creating three objects to represent the $o1$ object, each of which carries data and behavior specific to its corresponding class. As we can see,

[6] While regular view systems (i.e., that do not allow for capacity-augmenting views) also must support an object to be an instance of different virtual classes (as well as their base class), note that virtual classes do not carry any additional stored data – and it is thus trivial to make these objects transient members of virtual classes on access. This is no longer sufficient for capacity-augmenting views.

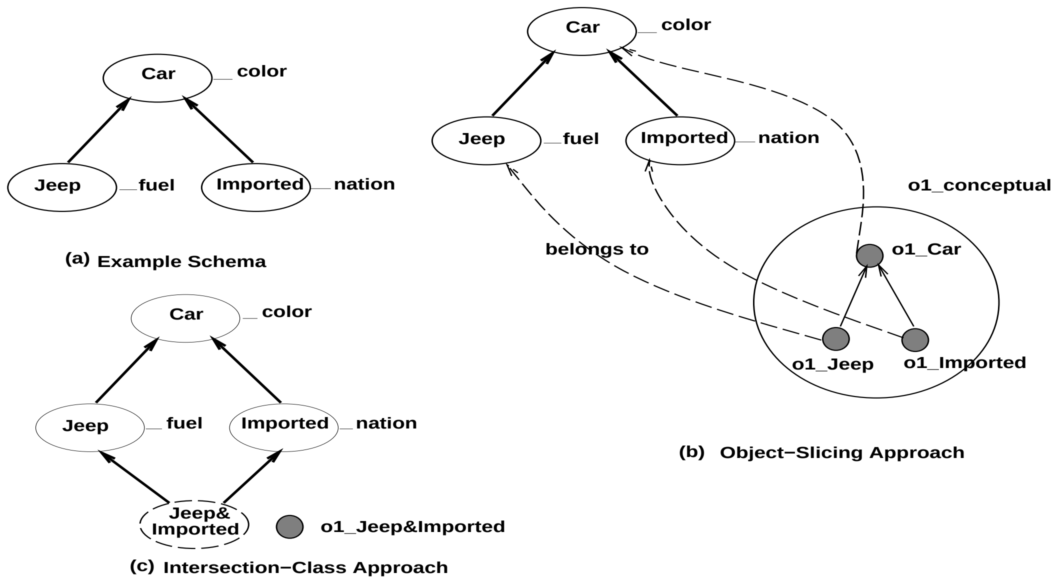

Figure 5: Two Approaches For Implementing Multiple Classification.

the $o1$ object corresponds to a hierarchy consisting of the $o1_{Car}$, $o1_{Jeep}$ and $o1_{Imported}$ objects. When the current class of the $o1$ object is Jeep, the $o1_{Jeep}$ object represents the $o1$ object. We call the $o1$ object itself the *conceptual object* and the three type-specific objects the *implementation objects*.

The object-slicing approach also enables efficient dynamic restructuring of object representations to account for the addition of new instance variables. Suppose that we extended our schema, which only contained the *Car* and the *Jeep* classes, by a new **refine** class called *Imported*, which refines the **Car** class by adding the stored attribute *nation*. Then each *Car* object (as well as each *Jeep* object) can acquire the type of the *Imported* class. This means that the *Car* object representation should be restructured such that it now carries the data for the new attribute. This can be accomplished by simply creating implementation objects of the *Imported* class and adding them to each *Car* object. So, restructuring of the object representation is relatively efficient and simple compared with the conventional architecture where each object carries all of its state information in a contiguous block of memory and belongs to only one class.

4.2 Comparing Two Approaches for Multiple Classification

Both approaches have advantages and their disadvantages. A detailed comparison is presented in Table 1, using the following criteria:

- Casting an object to a type is readily provided by switching between the representative implementation objects in the object-slicing approach. However, in the intersection-class approach, we need an additional mechanism (possibly implemented by the compiler) to implement the casting facility.
- In the object-slicing approach, one object consists of one conceptual object and a number of implementation objects equal to the number of types an object participates in. So, the number of object identifiers necessary for one object is equal to $1 + N_{impl}$, where N_{impl} denotes the average number of implementation objects for each object. The intersection-class approach requires only one object identifier for each object.
- Object-oriented databases use storage for purposes other than storing data values, such as indexes and object identifiers. The object-slicing approach requires the storage for a number of object identifiers $((1 + N_{impl}) \cdot sizeOf(oid))$ [7] and pointers to link the implementation objects with the conceptual objects $(2 \cdot N_{impl} \cdot$

[7] Object identifiers are necessary for one conceptual object and N_{impl} implementation objects.

	object-slicing	intersection-class
casting	change representative implementation object	need additional mechanism
#oids for one object	$1 + N_{impl}$	one
storage for managerial purpose	$(1 + N_{impl}) \cdot sizeOf(oid)$ $+N_{impl} \cdot 2 \cdot sizeOf(pointer)$	sizeOf(oid)
storage for data values	no redundancy	no redundancy
#classes	#user-defined classes	#user-defined classes + #intersection classes
performance for queries	good for select based on attributes	fast access to inherited attributes
dynamic classification	by creating and destroying implementation object	by creating another object and copying values and removing old one
multiple inheritance resolution	has flexibility to dynamically resolve	has to be determined statically

N_{impl} : the number of implementation objects for each object.

Table 1: Comparison of Two Multiple Classification Approaches.

$sizeOf(pointer))$ [8]. On the other hand, the intersection-class approach requires the storage for only one oid for each object.

- Both approaches do not require duplication of the data values. So, they are non-redundant in terms of the storage for data values unless data is purposely duplicated for performance reasons.

- In the object-slicing approach, all classes in the global schema are user-defined classes. There is no need to create hidden classes. However, the intersection-class approach needs intersection classes to accomplish multiple classification. For each object that takes two types, we must create a class to hold the combination of the two types, if it does not yet exist. The number of intersection classes may increase to $2^{N_{class}}$, where N_{class} is the number of user defined classes of the global schema.

- It is likely that the object-slicing approach will show good performance for select queries of predicates on simple attributes. There are two reasons: first, slices of the objects of the same attributes tend to cluster and second, these object slices are much smaller than complete objects and thus one page access should be sufficient to get all objects from secondary storage. However, the access to the inherited attributes involves several steps of traversing the pointers which link implementation objects. The intersection-class approach will be faster in accessing an inherited attribute because the values of all attributes of an object reside in the same location. Detailed comparison studies are needed to fully analyze the trade-off in performance for different types of access.

- Changing an object from being an instance of one class ($C1$) to being an instance of another class ($C2$) is called *dynamic classification* [13]. In the object-slicing approach, when the object is dynamically classified as the instance of the class $C2$ rather than that of class $C1$, the object instance takes an implementation object of the class $C2$ and discards that of the class $C1$. By combining the operators for adding and removing class membership, we can easily achieve dynamic classification functionality. In the intersection-class approach, we first must identify the proper class for the new classification and, if it does not exist, create the class. Second, we create an object of this new class and copy the values of the object to be reclassified into this newly created object. To preserve object identity, we must copy the object identity of the old object to the new object by utilizing a swap mechanism. OODBs such as GemStone that support rudimentary schema evolution capabilities provide such operators.

- The multiple inheritance resolution scheme has to be determined when the system is installed for the intersection-class approach. This is because representation of an object is affected by the resolution scheme.

[8]Each implementation object keeps the pointer to its conceptual object, vice versa.

Suppose a class inherits the same named attributes from both superclasses. Then, depending on resolution scheme, storage is allocated to either attribute or to both attributes. For the object-slicing approach, the object representation is the same regardless of resolution scheme for the multiple inheritance. This means that we can adopt various resolution schemes dynamically.

We have chosen the object-slicing approach as the basic architecture of our system because an explosion of intersection classes is likely to be generated in the intersection-class approach. In the worst case, the number of intersection classes could grow exponentially with respect to the number of user-defined classes. Also, as demonstrated above, dynamic classification may require the creation and/or removal of intersection-classes on the fly. Note that our system is independent of this choice of implementation architecture as long as the underlying object model supports the required properties of multiple classification and dynamic restructuring. Thus new mechanims for supporting multiple classification could be swapped in in the near future, when they become available.

5 The Transparent Schema Evolution (TSE) System Architecture

Figure 6 describes the overall architecture of the proposed *transparent schema evolution* system, or **TSE** system [9]. As shown in the figure, **GemStone**, version 3.2, Opal Interface [10] is used as the underlying platform providing persistent storage, concurrency control, etc. On top of GemStone, we have built the **TSE object model** to support the necessary features of multiple inheritance, multiple classification and dynamic restructuring. The **Global Schema Manager** is directly constructed on top of the **TSE model**, providing a layer between the database and all other system modules.

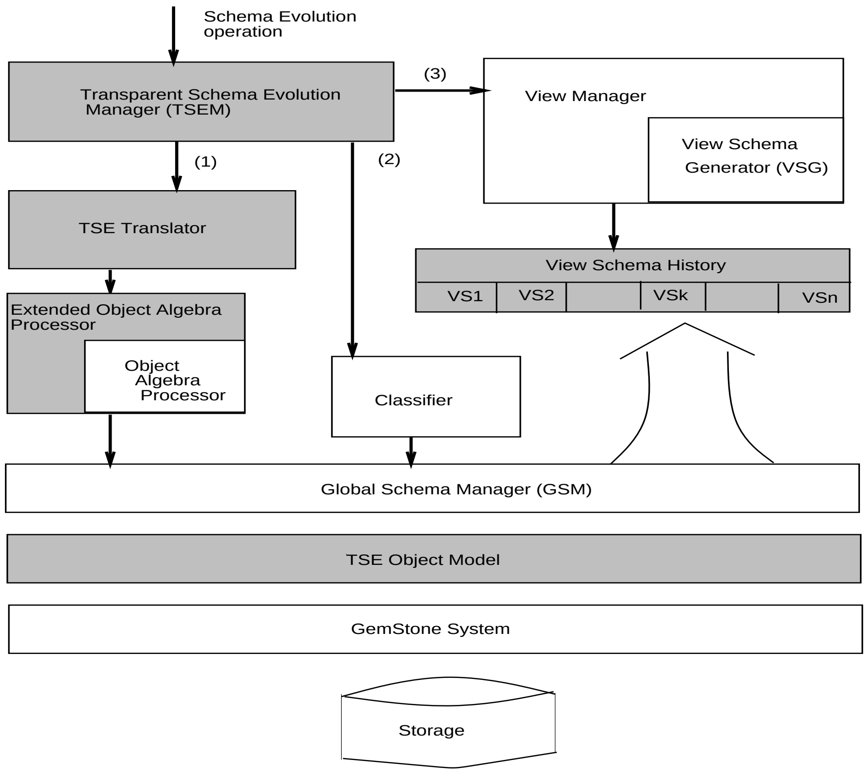
Schema Evolution operation
Transparent Schema Evolution Manager (TSEM)
(3)
View Manager
View Schema Generator (VSG)
(1)
(2)
TSE Translator
View Schema History
VS1
VS2
VSk
VSn
Extended Object Algebra Processor
Object Algebra Processor
Classifier
Global Schema Manager (GSM)
TSE Object Model
GemStone System
Storage

Figure 6: The System Architecture for Transparent Schema Evolution.

When a user specifies a schema change on a view (VS_k), the *Transparent Schema Evolution Manager* (**TSEM**) module calls the **TSE Translator** as indicated by the bold arrow labeled (1). The translator, in turn, maps the

[9] In Figure 6, the shaded modules are specific to the TSE system, and are not available as subsystems of MultiView.
[10] GemStone is registered trademark of Servio Corporation.

operation of schema change into statements of the **Extended Object Algebra** (Section 3.2). The execution of the translated algebra results in the creation of new virtual classes. Now, the **TSEM** calls the **Classifier** module which integrates the new virtual classes into one uniform global schema. This is indicated by the arrow labeled (2). The TSEM calls the **View Manager** to generate an appropriate view schema which reflects the modified schema and registers the generated view schema as a new version of the view (VS_k) in the dictionary, called **View Schema History**, indicated by the arrow labeled (3).

Each module of our TSE system is briefly explained below:

- **TSE object model**: The object model provides the required features of multiple classification, dynamic classification, multiple inheritance and casting to support our schema evolution system.
- **Transparent Schema Evolution Manager**: The control module takes a user input (a schema change) and calls appropriate modules to complete the requested schema change.
- **TSE Translator**: The module translates a requested schema change into a set of extended object algebra expansions that define all necessary virtual classes.
- **Extended Object Algebra**: The object algebra operators of MultiView take a set of objects as input and return a set of objects. The output characteristics of the operators enable the algebra to define virtual classes to constitute views. To achieve schema evolution through a view, we need to extend the operators to augment the content of the base schema. Thus, the extended algebra can not only customize the interface of existing data but it also can add new stored attributes to accommodate new data.
- **Classifier**: This module reclassifies the classes of the global schema to integrate the newly created virtual classes into the consistent global schema hierarchy, allowing for upwards inheritance for both base and virtual classes.
- **View Manager**: The module takes a set of classes as an input and generates a consistent view schema including the appropriate generalization hierarchy. Currently, we can check the type-closure of a view schema and incorporate necessary classes for the type-closure. In addition, it manages the history of views.
- **View Schema History**: The dictionary keeps track of the history of each view schema, allowing for the substitution of the old view by the newly cerated one.

We are currently implementing a prototype of the TSE system on top of GemStone. We are able to reuse several subsystems of MultiView, namely the **Classifier**, **View Manager** and **Object Algebra Processor** with minor modifications for integration purposes. We have completed an initial version of the **TSE Object Module**, the **View Schema History Manager** and the **Extended Algebra Processor**. We are now embarking on the development of the **TSE Translator**.

6 Algorithms for Realizing Schema Changes in the TSE System

One of the first object-oriented schema change approaches has been proposed by Banerjee *et al.* [8] for the ORION data model. Note that this taxonomy, adopted in most other schema evolution research for OODBs such as Encore [27], Goose [7, 11], CLOSQL [15], Rose [14], OTGen [10] and COCOON [30], still corresponds to the most frequently used set of schema changes. In fact, most commercial OODB systems such as O_2 [31] and GemStone [16] only support a subset of this taxonomy. To demonstrate the feasibility of the TSE approach, it is thus important to show that our approach can realize all the schema change operations supported by Orion.

Zicari [31] shows that the schema evolution operations of Orion can be reduced to a small set of primitive schema change operators, which can be combined to achieve the semantics of more complex operators. Zicari's schema update operators fall into two categories: those that change the content of one class, such as (1) add an attribute, (2) delete an attribute, (3) add a method and (4) delete a method, and those that act upon the class hierarchy, such as (1) add an is-a edge between two classes, (2) delete an is-a edge between two classes, (3) delete a class and (4) add a class. The implementation of this basic set of schema evolution operators is thus sufficient for our purpose. Below, we present algorithms for translating a schema change into a number of view definition statements for each primitive schema update operator.

6.1 Implementing the Add-Attribute Schema Change in TSE

6.1.1 Semantics of the Add-Attribute Operator

The schema change operator defined by "**add_attribute** x:*attribute-def* **to** C" augments the types of the class C and its subclasses C_{sub} with the new attribute 'x' [4]. The extents of the classes are not changed in terms of membership. However, the instance objects of the classes now have one additional attribute x. If there is a property in class C with the same name x, the operation is rejected. If a subclass C_k of C has a property with the name x locally defined, propagation of the added attribute to C_k and its subclasses is stopped because a local property overrides inherited ones. For multiple inheritance conflicts, we allow two same named properties to be inherited into the same class. However, due to the ambiguity, the properties can't be invoked until the user disambiguates the properties by renaming them. As an example, Figures 3 (a) and (b) show a schema before and after executing the "**add attribute** *register* **to** *Student* class" operation, respectively.

6.1.2 The Algorithm for Mapping the Add-Attribute Operator to Views

TSE translates the "**add_attribute** x:*attribute-def* **to** C" operator to the following view specification:

```
{ if x already exists in $C$, reject this operation;
defineVC $C'$ as (refine x:attribute-def for $C$);
push $C$ onto tmpStack;
while ($tmp$ := pop $tmpStack$) $\neq$ NULL do
    for all subclasses ($C_{sub}$) of the class $tmp$
        if attribute x not defined for $tmp$
            then { defineVC $C'_{sub}$ as (refine $C'$:x for $C_{sub}$);
                push $C_{sub}$ onto tmpStack;
    $tmp$ := pop;
endwhile; }
```

This algorithm creates virtual classes to reflect the modification of the class C and its subclasses. The virtual classes have types which refine the types of their source classes with the attribute 'x', while the extents are not modified. If there exists an attribute named 'x' in C, then the **refine** is rejected. In addition, if a subclass of C has already defined a property named 'x', the procedure of propagating 'x' terminates.

6.1.3 The Complete Process of View Schema Evolution

Figure 7 shows the whole process of our schema change approach. Upon the schema evolution request, "**add_attribute** *register* **for** *Student*", on the view schema of a user (Figure 7 (a)), the **TSE Translator** generates a set of view specification statements (Figure 7 (b)) using the above algorithm. The **Extended Object Algebra Processor** module takes the set of statements and creates the virtual classes, **Student'** and **TA'** in this example, according to the statements. The **Classifier** module then integrates the new virtual classes into the global schema as shown in Figure 7 (c). During the classification process, if there already exists a class in the global schema which is the same as a newly derived class, then the classification algorithm will discover this duplicate and discard the new class. From the augmented global schema, the **View Schema Manager** (VSM) picks all base classes and virtual classes from the old view unless they have corresponding new *primed* classes[11]. In Figure 7 (c), the VSM module selects the classes **Person**, **Student'** and **TA'** for the new view VS2. It then renames the **Student'** and **TA'** classes to **Student** and **TA** within the context of VS2, respectively. Finally, generalization edges are generated by **View Schema Generator** for the classes selected for VS2 [21]. At last, the system replaces the old

[11] The algorithm names each virtual class by appending a prime to the name of their corresponding original class.

view VS1 with the newly generated view VS2. Because all the above procedures are transparent to the schema change specifier, she will have the perception that her original schema has actually been modified.

6.1.4 Verification of the Translation Process

Proposition A: *The algorithm in Section 6.1.2 correctly simulates the semantics of the desired schema change described in Section 6.1.1.*
Let an original schema $S = (V, E)$ be defined as a DAG, where $V = \{v_1, v_2, ..., v_n\}$ and $E = \{e_1, e_2, ..., e_m\}$. Assume that normal schema modification of add-attribute would change the schema S into $S' = (V', E')$; and our view change approach would generate the view $S'' = (V'', E'')$ instead. Then, we will verify that the newly created schema S'' correctly represents the desired change S' by showing $S' = S''$.

Let us assume without loss of generality that a subclass of class C, $(C_{sub} \neq C)$, has an attribute 'x' already defined. Then we may partition the set V into three subsets $V1$, $V2$ and $V3$, where $V1$ contains the set of classes that are subclasses of C_{sub}, $V2$ the set of classes that are subclasses of C (including C) but not subclasses of C_{sub} and $V3$ the remaining classes. Let $V1 = \{v_1, v_2, ..., v_k\}$, $V2 = \{v_{k+1}, v_{k+2}, ..., v_{k+i}\}$ and $V3 = \{v_{k+i+1}, v_{k+i+2}, ..., v_{k+i+j}\}$, with $k + i + j = n$.

Let we first discuss how S' is created. Then, normal schema modification would change only the classes of $V2$ by augmenting the types of the classes in $V2$ by the attribute 'x'; the extents are not modified. The remaining classes are not changed. So, $V2'$ would be $\{v'_{k+1}, ..., v'_{k+i}\}$, with v'_p is the result of modifying v_p, $k + 1 \leq p \leq k + i$, defiend by:

$$\begin{aligned} type(v'_p) &= type(v_p) \cup \{x\} \\ extent(v'_p) &= extent(v_p). \end{aligned} \tag{6.1}$$

Let us now examine how S'' is created. Our **TSE Translator** algorithm in Section 6.1.2 identifies the classes for which virtual classes are to be created. It can be easily seen that they also correspond to the classes of the set $V2$. After generating the virtual classes, the new view schema V'' would be equal to $\{v_1, ..., v_k, v''_{k+1}, ..., v''_{k+i}, v_{k+i+1}, ..., v_n\}$, where v''_p is a virtual class created for v_p, $k + 1 \leq p \leq k + i$. The v''_p classes, for $k + 1 \leq p \leq k + i$, are defined by:

$$\begin{aligned} type(v''_p) &= type(v_p) \cup \{x\} and \\ extent(v''_p) &= extent(v_p) \end{aligned} \tag{6.2}$$

because the above algorithm creates virtual classes that refine the classes of $V2$ with the attribute 'x'. From the equations 6.1 and 6.2, we see that $v'_p = v''_p$ for all p with $k+1 \leq p < k+i$, and $V' = V''$. Because this schema change does not modify the is-a relationships between classes, we know $E = E'$. The *view schema generation algorithm* of TSE, which is already found in [21], will generate the same is-a relationships as in E, and thus $E'' = E$. This implies that $E'' = E' = E$. Because $V' = V''$ and $E' = E''$, we can say that $S'' = S'$ and that the semantics of the schema change have been correctly simulated.

Proposition B: *Existing view schemas are not affected by this schema change.*
The global schema is restructured for this schema change operator in order to add some new refining virtual classes. However, as shown elsewhere as the view independence property [19], existing views are not affected by such global schema restructuring.

6.1.5 Updatability

For this view, virtual classes are created only by the *object-preserving* **refine** operator. Because it has been shown that virtual classes defined by object preserving object algebra are updatable [24], all classes of the view are updatable.

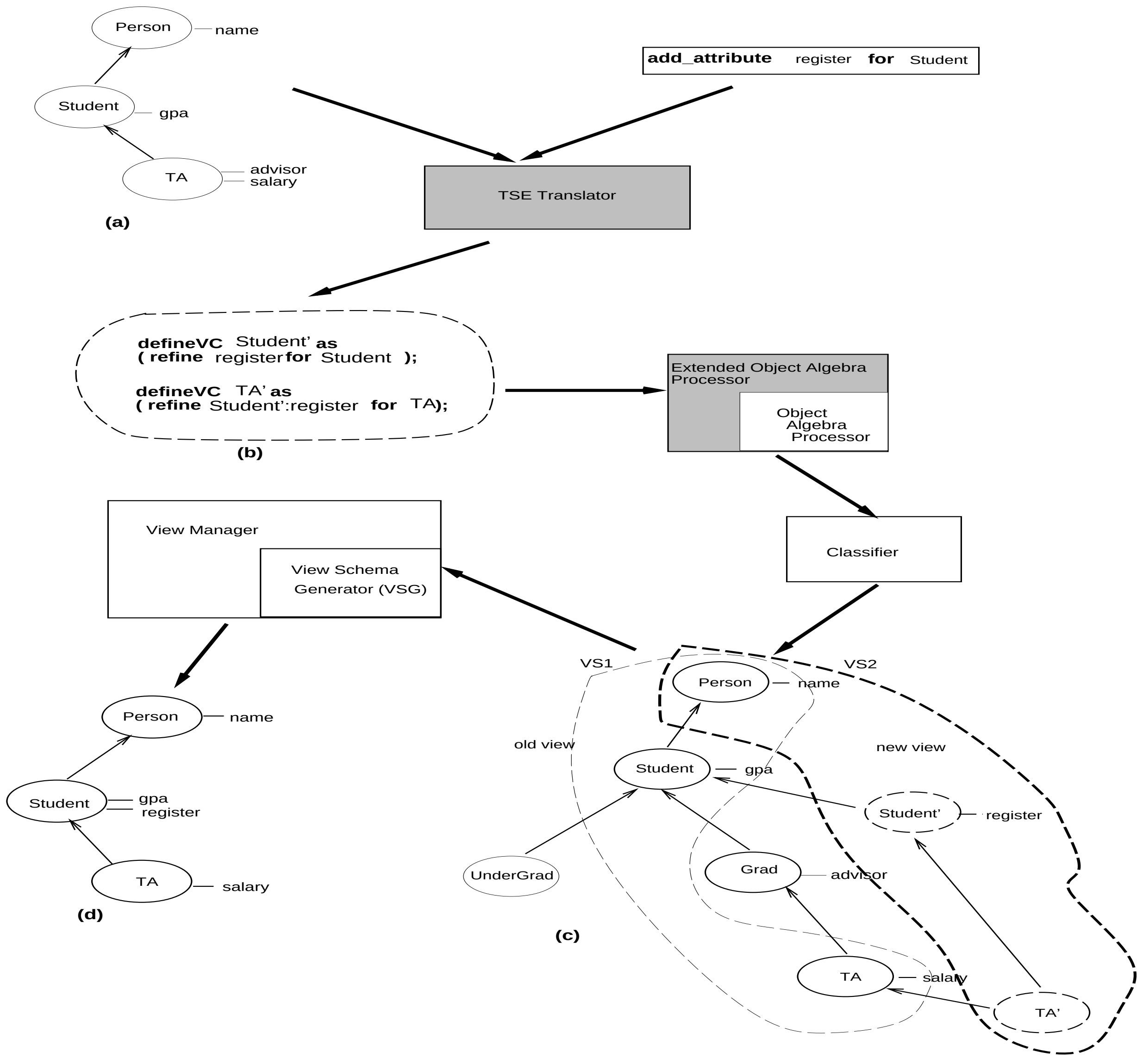

Figure 7: Complete Procedure for a Schema Change.

6.2 Implementing the Delete-Attribute Schema Change in TSE

6.2.1 Semantics of the Delete-Attribute Operator

The schema change operator defined by "**delete_attribute** *attribute* (x) **from** C" removes the attribute 'x' from the types of the class C and its subclasses. The extents of the classes remain the same. We can delete only attributes that are locally defined in class C to guarantee the full inheritance invariant [12]. If the attribute 'x' was overriding a same named attribute in class C, then the suppressed attribute is restored and propagated to the subclasses.

The term of 'local' above must be redefined in our context, because local property in our context means local in terms of the view schema. We consider an attribute 'x' to be locally defined in class C if class C is the uppermost class in the view schema having the attribute 'x', even if the property is defined outside the view.

6.2.2 The Algorithm for Mapping the Delete-Attribute Operator to Views

TSE translates the "**delete_attribute** *attribute* (x) **from** C" operator to the following view specification:

```
{    for all subclasses $subC$ of the class $C$, including $C$
         defineVC $subC'$ as ( hide x from $subC$ );
     if there exists an inherited attribute named 'x' in $C$ then
     { $superC$ := the class defining the inherited attribute;
         for all subclasses $C_{sub}$ of the class $C$, including $C$
             defineVC $C''_{sub}$ as ( refine $superC$:x from $C'_{sub}$ ); }
}
```

The above algorithm creates virtual classes that hide the attribute 'x' from the class C and its subclasses. When the attribute 'x' has been overrding a same named attribute in the class C, which has been inherited from a class (*superC*), the algorithm creates additional virtual classes that refine the class C and newly created subclasses C'_{sub} with the attribute, *superC*:x. This will effectively restore the suppressed attribute. For an example, Figures 8 (a) and (b) show an old and a new view schema, and Figure 8 (c) shows how the global schema is augmented to support the schema change. The key here is that the attributes to be deleted are not removed from the underlying global schema, but rather are made invisible to the view.

6.2.3 Verification of the Translation Process

Proposition A: *The algorithm in Section 6.2.2 correctly simulates the semantics of the desired schema change of deleting an attribute as described in Section 6.2.1.*
Let an attribute named 'x' is to be deleted from class C. Then we may partition the set V into two subsets $V1$ and $V2$, where $V1$ contains the set of classes that are subclasses of class C_{sub} (including C) and $V2$ the set of classes which are superclasses of class C. Again assume without loss of generality that $V1 = \{v_1, v_2, ..., v_k\}$ and $V2 = \{v_{k+1}, v_{k+2}, ..., v_{k+i}\}$, where $k + i = n$.

Then, normal schema modification would change the classes of $V1$ such that the types of the classes are reduced by the attribute 'x' and the extents are not changed. The classes of $V2$ are not affected by this schema change. So, V' would be $\{v'_1, ..., v'_k, v_{k+1}, ..., v_n\}$, where v'_p is modified from v_p, $1 \leq p \leq k$. The v'_p classes are defined by:

$$\begin{aligned} type(v'_p) &= type(v_p) - \{x\} \; and \\ extent(v'_p) &= extent(v_p) \; for \; 1 \leq p \leq k. \end{aligned} \tag{6.3}$$

Let us now how our view schema approach change the classes of V. The bf TSE Translator algorithm for the *delete-attribute* identifies the classes for which virtual classes need to be created. It can easily be seen that they are identical to the classes of the set $V2$, which in turn are equal to the set of classes modified by the normal schema change. Then, V'' would be $\{v''_1, ..., v''_k, v_{k+1}, ..., v''_n\}$, where v''_p are the virtual classes created for v_p for $1 \leq p \leq k$.

[12] The full inheritance forces all the properties of a superclass to be inherited by its subclasses.

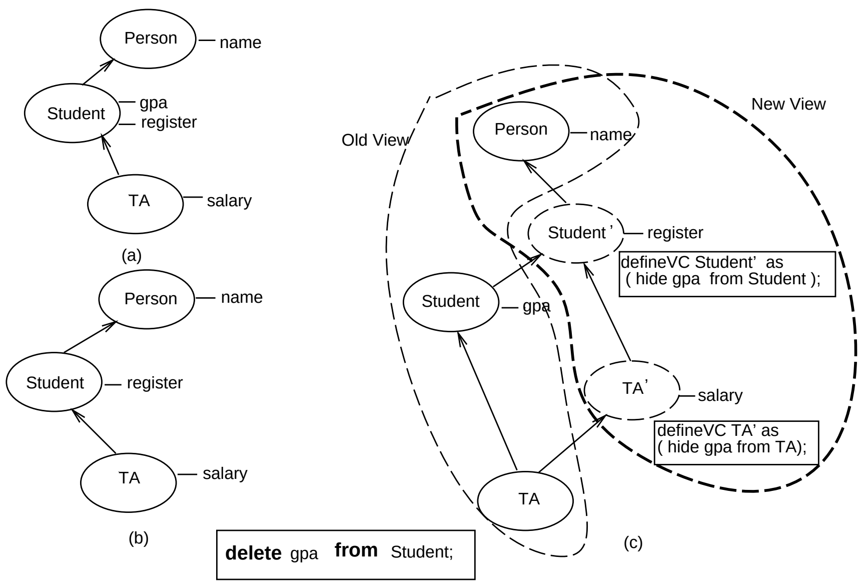

Figure 8: Schema Change for Deleting an Attribute.

The v''_p classes are defined by:

$$\begin{aligned} type(v''_p) &= type(v_p) - \{x\}\ and \\ extent(v''_p) &= extent(v_p)\ for\ 1 \leq p \leq k. \end{aligned} \tag{6.4}$$

This is so because the above algorithm creates virtual classes that hide the attribute 'x' from the classes of $V2$. From the equations 6.3 and 6.4, we see that $v'_p = v''_p$ for all $p\ s.t.\ 1 \leq p \leq k$, and thus $V' = V''$.

In addition, in case the class C could have inherited a same named attribute 'x' (denoted by $x_{suppressed}$) but suppressed by the attribute 'x' to be hidden (denoted by x_{hidden}), the attribute $x_{suppressed}$ will be restored when the suppressing attribute x_{hidden} is deleted. Then the type equations of 6.3 and 6.4 will be changed into

$$type(v'_p) = type(v_p) - \{x_{hidden}\} \cup \{x_{suppressed}\} \tag{6.5}$$
$$type(v''_p) = type(v_p) - \{x_{hidden}\} \cup \{x_{suppressed}\}\ for\ 1 \leq p \leq k, \tag{6.6}$$

respectively. Even for this case, we can see that $v''_p = v'_p$ for $1 \leq p \leq k$, and V'' is equal to V', too. Because this schema change does not change the is-a relationships between classes, we have $E = E'$. Besides, the *view schema generation algorithm* will generate the same is-a relationships as in E [21], and then $E'' = E$. In short, it means that $E'' = E' = E$. Because $V' = V''$ and $E' = E''$, we can say that $S'' = S'$ and that the semantics of the schema change have been correctly simulated.

Proposition B: *Existing views are not affected by this schema change.*
For this operator, existing classes in the global schema may go though some changes. Some of methods and instance variables that had been locally defined have now moved upward to other classes [13]. In this example, the definition of the instance variable *register* is moved upwards from **Student** to the class **Student'**. It may be possible that this code movement has affected the type of existing classes, namely, the classes **Student** and **TA**. For example, if the attribute *register* in the **Student** class in Figure 8 (c) was overriding an inherited and same named attribute, now the two attributes become conflicting. This problem can be solved by always giving higher priority for inheritance to the class that is the hide class created from the inheriting class. *In other words, the property that had been defined at a class and projected into a superclass always has higher priority over other same named properties inherited from other superclasses.* Given this name conflict resolution rule, we can assure that users of the other views are not affected by the fact that a property that had been locally defined has been moved up and now is only inherited into the class. This outlines our argument why other views are not affected by this operation.

[13] This code promotion is performed in MultiView to allow for true upwards method resolution for both base and virtual classes.

6.2.4 Updatability

For the same reason as that of adding an attribute, this view is also updatable.

6.3 Implementing the Add-Method Schema Change in TSE

6.3.1 Semantics of the Add-Method Operator

The schema change operator defined by "**add_method** m: *method-def* **to** C" augments the types of the class C and its subclasses with the new method 'm'. The extents of the classes are not changed. If there is a same named method locally defined in class C, the operation is rejected. If a subclass of C has a method 'm' locally defined, the propagation of adding the new method to the subclasses is stopped because a local method overrides an inherited one. We allow two same named methods to be inherited into the same class. However, the conflict has to be resolved by the user by renaming the methods.

6.3.2 The Algorithm for Mapping the Add-Method Operator to Views

The algorithm for this schema update is the same as that of the *add_attribute* operator given in Section 6.1.2 except that this operation doesn't have to deal with reorganizing the underlying object representation.

6.3.3 Verification of Translation Process and Updatability

The same arguments for teh correctness of the TSE algorithm given for the *add_attribute* schema change given in Section 6.1.4 also hold for this schema change. The updatability of the resulting schema can be shown similarly to the updatability for the *add_attribute* schema change case (Section 6.1.5.

6.4 Implementing the Delete-Method Schema Change in TSE

6.4.1 Semantics of the Delete-Method Operator

The schema change operator defined by "**delete_method** m **from** C" changes the types of the class C and its subclasses such that method 'm' is no longer defined. The extents of the classes remain the same. We can delete only locally defined methods in class C to guarantee the full inheritance invariant. Again the term local property refers to properties with respect to the view schema. In other words, even if the property 'm' is defined outside the view, but the class C is the uppermost class having the method 'm' in the view schema, then 'm' is considered a local property of class C. If the method 'm' was overriding a same named method in class C in the view, then the method is restored and propagated to the subclasses.

6.4.2 The Algorithm for Mapping the Delete-Method Operator to Views

The algorithm for this schema update is the same as that of the *delete_attribute* operator.

6.4.3 The Verification of Translation Process and Updatability

The same arguments for the *delete_attribute* schema change also hold for this schema change (Section 6.2.3). The updatability of the resulting schema is similar to that for the *delete_attribute* schema change (Section 6.2.4).

6.5 Implementing the Add-Edge Schema Change in TSE

6.5.1 Semantics of the Add-Edge Operator

The schema change operator defined by "**add_edge** C_{sup}-C_{sub}" adds an is-a relationship between two classes by making C_{sup} a superclass of C_{sub}. Semantically, the addition of the is-a relationship between two classes results in all properties of class C_{sup} being inherited by class C_{sub} and its subclasses. The added edge may cause multiple inheritance problems. Our solution is that same named properties can be inherited into one class, and to leave the resolution up to the user as previously discussed. If a subclass defines a property whose name is the same as one of the properties in the class C_{sup}, then propagation of the property of the class C_{sup} is blocked (overridden). The addition of the is-a relationship also results in the addition of the extent of class C_{sub} to the extent of class C_{sup} and its superclasses. In Figures 9 (a) and (b), the class *SupportStaff* is made a superclass of the class *TA*. As a result, the class *TA* and its subclass *Grader* now inherit the property *boss*. In addition, the extent of the class *TA* is added to the extent of the *SupportStaff* class and of the *Person* class. Note that in the figure the extent of a class is denoted by the set bracket below the class [14]. The extent of the class *SupportStaff* { o2 o3} is expanded to { o2 o3 o4 o5 o6 }.

6.5.2 The Algorithm for Mapping the Add-Edge Operator to Views

TSE translates "**add_edge** C_{sup}-C_{sub}" into the following view specification:

```
{   for all subclasses ($w$) of $C_{sub}$, including $C_{sub}$
            **defineVC** $w'$ **as** ( **refine** *properties of* $C_{sub}$ **for** ($w$)); [15]
     for all superclasses ($v$) of $C_{sup}$ that are not already superclasses of $C_{sub}$, including $C_{sup}$
            **defineVC** $v'$ **as** (**union**($v$ , $C'_{sub}$));}
```

The first and second statement add properties of the class C_{sup} to the classes which now become subclasses of C_{sup} due to the added is-a relationship. In Figure 9 (c), the classes *TA'* and *Grader'* are created with their type augmented by the property *boss* of *SupportStaff* (C_{sup}). The third statement adds the extent of class C_{sub} to all classes which now have become superclasses of C_{sub} due to the added is-a relationship. In Figure 9 (c), the extent of class *TA* (C_{sub}) is added to the extent of class *SupportStaff* (C_{sup}). The union virtual class *SupportStaff'* contains the union of the extents of the source classes of *TA'* and *SupportStaff*. The *Person* class is not modified, because the *TA* class was already a subclass of the *Person* class before the schema change.

6.5.3 Verification of the Translation Process

Proposition A: *The algorithm in Section 6.5.2 correctly simulates the semantics of the desired schema change of adding an is-a relationship as described in Section 6.5.1.*
Let us assume without loss of generality that one of properties of class C_{sup} is named 'x' and that one of the subclasses of class C_{sub} ($C_{override}$) has already locally defined an overriding property named 'x'. We now denote the property 'x' of the class C_{sup} as $x_{inherited}$, and the (overriding) property 'x' as $x_{override}$. Then we partition the set of V into three subsets $V1$, $V2$ and $V3$, where $V1$ contains the set of classes that are subclasses of class $C_{override}$, $V2$ the set of classes that are subclasses of the class C_{sub} (including C_{sub}) but not subclasses of $C_{override}$, and $V3$ contains the superclasses of C. Again assume without loss of generality that $V1 = \{v_1, v_2, ..., v_k\}$, $V2 = \{v_{k+1}, v_{k+2}, ..., v_{k+i}\}$ and $V3 = \{v_{k+i+1}, v_{k+i+2}, ..., v_{k+i+j}\}$, where $k + i + j = n$.

Then, normal schema modification would make the class C_{sub} a subclass of C_{sup}. The addition of the new generalization relationship will create the following new schema $V' = \{v'_1, ..., v'_k, v'_{k+1}, ..., v'_{k+i}, v'_{k+i+1}, ..., v'_n\}$, where v'_p is derived from v_p, for $1 \leq p \leq k + i + j$. The v'_p classes are defined by:

$$\textit{for } 1 \leq p \leq k$$
$$\textit{type}(v'_p) \quad = \quad \textit{type}(v_p) \cup (\textit{type}(C_{sup}) - \{x_{inherited}\}) \textit{ and}$$

[14] The term 'extent' used in this paper is implicitly assumed to be global extent, and not local extent.

[15] Note that when class C_{sub} already has same named properties as those to be added by **refine** operation, those properties are not added to a new *refine* virtula class. This effectively achives the semantics of overriding.

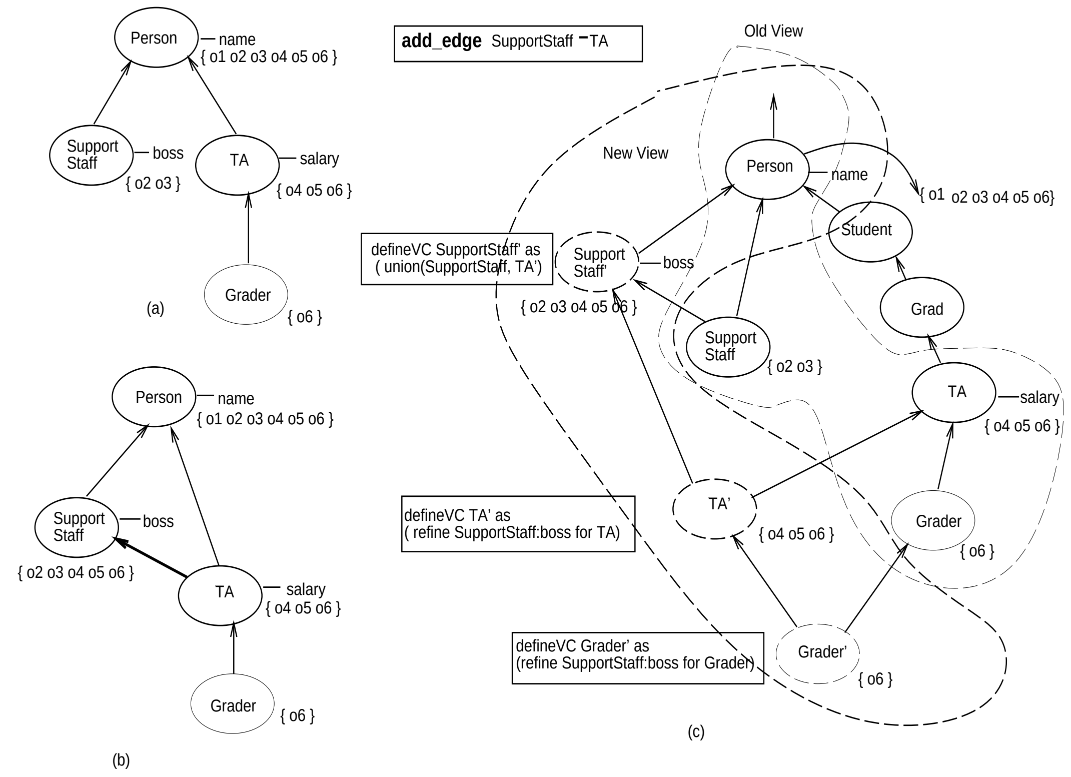

Figure 9: Schema Change for Adding a Generalization Edge.

$$
\begin{aligned}
extent(v'_p) &= extent(v_p); && (6.7)\\
for\ k+1 \le p \le k+i & \\
type(v'_p) &= type(v_p) \cup type(C_{sup})\ and & \\
extent(v'_p) &= extent(v_p); && (6.8)\\
for\ k+1 \le p \le k+i & \\
type(v'_p) &= type(v_p)\ and & \\
extent(v'_p) &= extent(v_p) \cup extent(C_{sub}). && (6.9)
\end{aligned}
$$

For our view schema approach, the algorithm in Section 6.5.2 creates virtual classes for the superclasses of C_{sup} and C_{sup} itself and adds the extent of class C_{sub} to the superclasses of C_{sup} and to C_{sup}. In addition, for the subclasses of the class C_{sub}, it creates virtual classes whose types are defined to be the union of the original type and that of class C_{sub} except for the overridden properties. More formally, after generating the necessary virtual classes, the resulting view schema classes V'' would be $\{v''_1, ..., v''_k, v''_{k+1}, ..., v''_{k+i}, v''_{k+i+1}, ..., v''_n\}$, where v''_p is a virtual class created for v_p, $1 \le p \le n$. The v''_p classes are defined by:

$$
\begin{aligned}
for\ 1 \le p \le k & \\
type(v''_p) &= type(v_p) \cup type(C_{sup}) - \{x_{inherited}\} \cup \{x_{override}\}\ and & \\
extent(v''_p) &= extent(v_p) && (6.10)\\
for\ k+1 \le p \le k+i & \\
type(v''_p) &= type(v_p) \cup type(C_{sup})\ and & \\
extent(v''_p) &= extent(v_p) && (6.11)\\
for\ k+i+1 \le p \le n & \\
type(v''_p) &= type(v_p)\ and & \\
extent(v''_p) &= extent(v_p) \cup extent(C_{sub}). && (6.12)
\end{aligned}
$$

From the equations, we see that $v'_p = v''_p$ for all p $s.t.$ $1 \leq p \leq n$, and $V' = V''$.

Because normal schema modification just adds an is-a relationship, $E' = E \cup \{< C_{sup}, C_{sub} >\}$. Besides, the *view schema generation algorithm* will generate the is-a relationships of E augmented by the edge $C_{sup} - C_{sub}$ for the new schema E", and thus $E'' = E \cup \{< C_{sup}, C_{sub} >\}$ [21]. Then, we can see that $E'' = E'$. Because $V' = V''$ and $E' = E''$, we can say that $S'' = S'$ and that the semantics of the schema change have been correctly simulated.

Proposition B: *Existing view schemas are not affected by this schema change.*
The process of realizing this view schema change also involves the promotion of methods. Especially when the union classes are created, the common properties of the two source classes are promoted up to the union class. Again, if we apply the conflict resolution rule that gives higher priority to the property which had been locally defined, the source classes are changed by neither type nor extent. In general, no other views are affected by this schema change by the view indenpendency property shown elsewhere [19].

6.5.4 Updatability

The **union** operator could possibly be problematic because there are two source classes associated with each unioned virtual class. In Figure 9 (c), the class *SupportStaff'* has two source classes *SupportStaff* and TA'. As we have discussed in Section 4, we have three options for propagating the **create** update on a union class. We can choose either *SupportStaff* or TA', or both classes for the source class to which the create is propagated. Assuming TA' is chosen and an object $o1$ is inserted into the class TA', then the object must also be inserted into the base class TA. Because TA' has the same extent as TA, this inserted object is visible in the class TA'. This does not correspond to the expected behavior because now every object inserted into a superclass (*SupportStaff'*) is visible to the subclass (TA'). If the operators such as **create** are propagated to both source classes, we will see similar unexpected behavior. We can avoid this undesirable situation by selecting the class *SupportStaff* as the propagation class. Since the *SupportStaff* class itself will no longer be visible to the view, the propagation of the **create** from the *SupportStaff'* superclass to the *SupportStaff* subclass goes unnoticed. In general, when a unioned virtual class (C') substitutes a source class (C) in an old view, **create** and **add** are propagated to the substituted class (C). To strengthen this argument, also note that class C' always has the same type as the class C. Other update operations such **delete**, **remove** and **set** are simple. They propagate to both source classes if they contain corresponding objects.

6.6 Implementing the Delete-Edge Schema Change in TSE

6.6.1 Semantics of the Delete-Edge Operator

The schema change operator defined by "**delete_edge** C_{sup}-C_{sub} [**connected_to** C_{upper}]" deletes the is-a relationship between the two classes, assuming C_{sup} is a superclass of C_{sub}. The option **connected_to** C_{upper} indicates that in case the class C_{sub} is disconnected from the DAG structure after deletion of the is-a relationship (i.e., C_{sup} is the only superclass of C_{sub}), then the class C_{sub} is made a direct subclass of the class C_{upper}. We require that the class C_{upper} needs to be a superclass of the class C_{sup}. If the optional clause is not specified, then the class C_{sub} is made a direct subclass of the system class **ROOT** [16] to avoid disconnecting the C_{sub}. Semantically, the deletion of the is-a relationship between a class C_{sub} and its superclass C_{sup} results in hiding from class C_{sub} and its subclasses all the properties that had been inherited from the class C_{sup} unless the properties are still inherited through another is-a relationship. It also results in hiding from the class C_{sup} and its superclass the extent of the subclass C_{sub} unless it is still in the scope of the class C_{sup} and its superclasses through another is-a relationship.

Figures 10 (a) and (b) show example view schemas before and after the update. The deletion of the is-a relationship between *TeachingStaff* and TA results in the property *lecture* being no longer inherited into the class TA. In addition, it also results in hiding the extent of TA { o4 o5 } from the extent of *TeachingStaff*, i.e., the extent is decreased from { o2 o3 o4 o5 } to { o2 o3 }.

[16]The **ROOT** class the root class of the DAG structure.

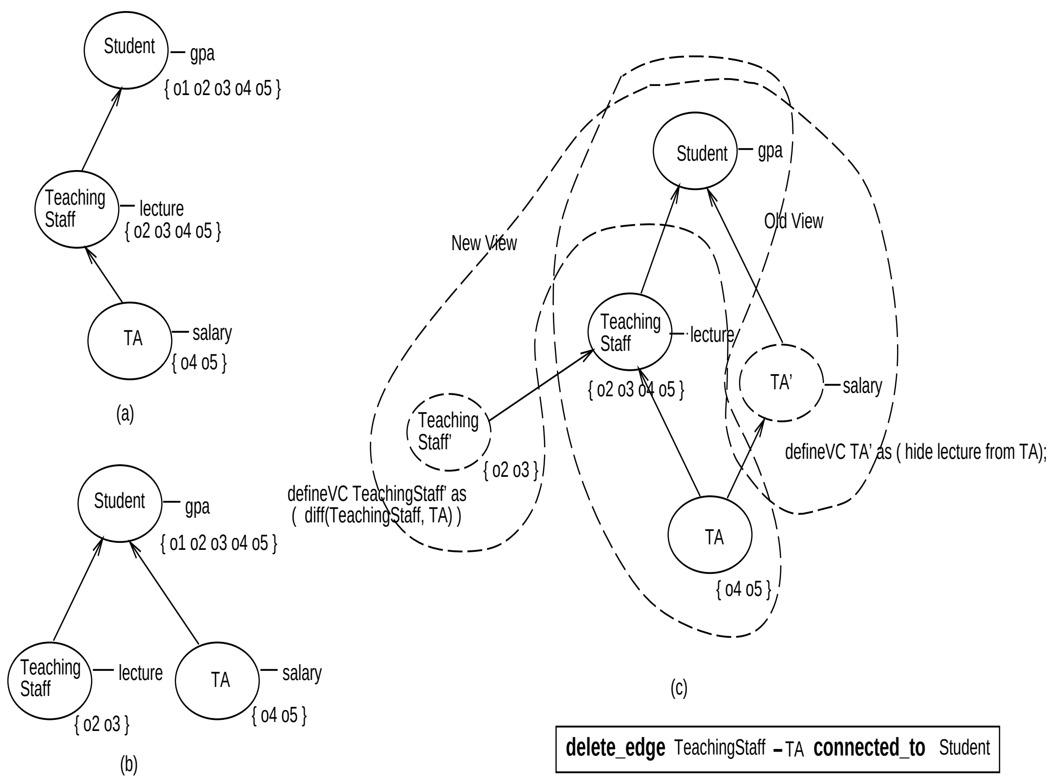

Figure 10: Schema Change for Deleting an Edge.

6.6.2 The Algorithm for Mapping the Delete-Edge Operator to Views

TSE translates the "**delete_edge** C_{sup}-C_{sub} [**connected_to** C_{upper}]" operator to the following view specification:

```
{     for all superclasses ($v$) of $C_{sup}$ (including $C_{sup}$) that are not superclasses of $C_{sub}$ through some other relationships
      {     defineVC $X$ as union( commonSub($v$,$C_{sub}$,$C_{sup}$-$C_{sub}$));
            defineVC $v'$ as (union(diff($v$,$C_{sub}$),$X$));          }
      for all subclasses ($w$) of $C_{sub}$, including $C_{sub}$
      {     $y$ = findProperties($w$,$C_{sup}$-$C_{sub}$);
            defineVC $w'$ as( hide $y$ from $w$);          }     }
```

The first loop in the algorithm hides from the extent of the superclasses of C_{sup} all instances that should become invisible to the superclasses when deleting the edge. To modify the extent of the superclasses, we may want to subtract the extent of C_{sub} from the superclasses. However, this will not always be correct, as shown by the example in Figure 11. In this example, the extents of classes $C1$, $C2$ and $C3$ would be still visible to the class v even after the is-a relationship between C_{sup} and C_{sub} is deleted. Hence, it can't be simply be removed from the class v. The macro **commonSub**(v,C_{sub},C_{sup}-C_{sub}) returns the list of classes that are the greatest common subclasses of v and C_{sub} assuming the edge C_{sup}-C_{sub} has been deleted. It will return the classes $C1$, $C2$ and $C3$ for the above example. The **union** of these returned classes is called X. The temporary virtual class X contains the instance objects that are still visible to class v even without the edge C_{sup}-C_{sub}. So, the extent of X should be added to **diff**(v,C_{sub}) to correctly simulate the effect of this schema change on the extent of the superclasses. This is achieved by the statement **union**(**diff**(v,C_{sub}),X).

The types of the superclasses are preserved because the type of the superclasses of C_{sup} are not affected by the deletion of the edge C_{sup}-C_{sub}. So, the type of the class v' is that same as that of the class v. The types of superclasses are also preserved even though **union** operation usually construct the type of virtual class, which is possibly different from any of source classes. However, the **union** operations in the above algorithm create virtula

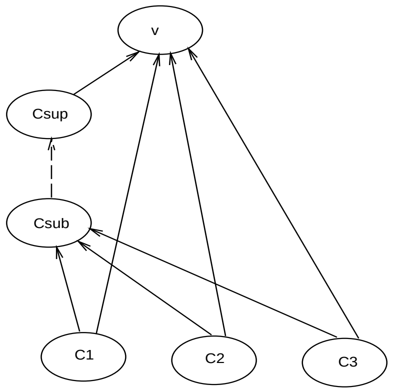

Figure 11: An Example of Deleting an Edge E.

classes whose types are same as those of first argument source classes, because the set of classes returned from **commonSub** are subclasses of the class v.

The macro, **findProperties**(w,C_{sup}-C_{sub}), returns the properties that have been inherited to the class w only through the is-a relationship C_{sup}-C_{sub} [17]. These properties have to be hidden from the class C_{sub} and its subclasses. In this case, when the properties are removed, no overridden property is restored because every property to be removed has not been defined locally but inherited.

Figure 10 (c) shows how this schema update restructures the global schema. The extent of the class $TeachingStaff'$ ({ o2 o3 }) is equal to the extent of $TeachingStaff$ ({ o2 o3 o4 o5 }) decreased by the extent of TA ({ o4 o5 }). The macro **commonSub**(TA,TA,$TeachingStaff$-TA) returns the TA class. The macro **findProperties**(TA,$TeachingStaff$-TA) returns the *lecture* property.

6.6.3 Verification of the Translation Process

Proposition A: *The algorithm in Section 6.5.2 correctly simulates the semantics of the desired schema change of deleting is-a edge as described in Section 6.6.1.*
We partition the set V into two subsets $V1$ and $V2$, where $V1$ contains the set of classes that are superclasses of the class C_{sup} and C_{sup} itself and $V2$ the subclasses of class C_{sub}. $V1 = \{v_1, v_2, ..., v_k\}$ and $V2 = \{v_{k+1}, v_{k+2}, ..., v_{k+i}\}$, where $k + i = n$. Assume that normal schema modification of deleting the is-a edge between C_{sup} and C_{sub} changes the classes (v_p) of the schema into the modified classes (v'_p).

For the classes v_p, which are superclasses of C_{sup} including C_{sup}, some instances would become invisible due to the deletion of the edge. The resulting classes v'_p, for $1 \leq p \leq k$, can be defined as follows:

$$\begin{aligned} extent(v'_p) &= extent(v_p) - \{instances\ to\ become\ invisible\ without\ the\ edge\} \\ &= extent(v_p) - extent(C_{sub}) \cup \{instances\ of\ C_{sub}\ still\ visible\}\ and \\ type(v'_p) &= type(v_p). \end{aligned} \tag{6.13}$$

For the classes v_p, which are subclasses of C_{sub} including C_{sub}, some properties of the class C_{sup} are no longer inherited. These classes v'_p, for $k + 1 \leq p \leq n$, as defiend by:

$$\begin{aligned} type(v'_p) &= type(v_p) - \{properties\ of\ class\ C_{sup}\ no\ longer\ inherited\}\ and \\ extent(v'_p) &= extent(v_p). \end{aligned} \tag{6.14}$$

Let v''_p be the virtual classes created by the algorithm describe in Section 6.6.2. For the classes v''_p for $1 \leq p \leq k$:

$$extent(v''_p) \quad = \quad (extent(v_p) - extent(C_{sub})) \cup extent(\bigcup\ classes\ returned\ from\ the\ commonSub)$$

[17]The algorithm to implement the macro identifies all paths from the origin class of an inherited property to the class w, and decides whether these paths contain the edge. If it does, the macro returns the property.

$$type(v''_p) \quad = \quad type(v_p). \tag{6.15}$$

For the classes v''_p for $k+1 \leq p \leq n$:

$$\begin{aligned} extent(v''_p) &= extent(v_p) \ and \\ type(v''_p) &= type(v_p) - \{properties\ returned\ from\ the\ findProperties\ macro\}. \end{aligned} \tag{6.16}$$

Comparing the equations of 6.13 and 6.15, we can see that v'_p and v''_p are the same if the union of extents of classes returned from the macro **commonSub** is equal to the instances still visible without the edge. By definition, the macro **commonSub** returns all classes that are the greatest common subclasses of v_p and C_{sub} assuming the edge C_{sup}-C_{sub} has been deleted, and in turn these classes cover all the still visible instances. So, $v''_p = v'_p$ for $1 \leq p \leq k$. Referring to both equations of 6.14 and 6.16, again v'_p and v''_p are the same if the macro **findProperties** returns the no longer inherited properties of class C_{sup}. This is the exactly the definition of the macro. So, $v''_p = v'_p$ for $k+1 \leq p \leq n$.

Normal schema modification just deletes an is-a relationship, $E' = E - \{< C_{sup}, C_{sub} >\}$. The *view schema generation algorithm* will generate the is-a relationships of E decreased by the edge $C_{sup} - C_{sub}$, and then $E'' = E - \{< C_{sup}, C_{sub} >\}$ [21]. Then, we can see that $E'' = E'$. Because $V' = V''$ and $E' = E''$, we can know that $S'' = S'$ and that the semantics of the schema change have been correctly simulated.

Proposition B: *Existing view schemas are not affected by this schema change.*
For this operation, the global schema is also restructured such that the new classes are added and some properties are promoted. The promotion of properties does not affect the existing classes if we follow the *multiple inheritance priority* rule.

6.6.4 Updatability

Some of virtual classes created for this schema chagne of deleting an edge are *union* classes. For the same reason as in the case of adding an edge, **create** is propagated to the replaced source class. In Figure 10 (c), the **create** method applied to the class $TeachingStaff'$ is propagated to the class $Teachingstaff$. This decision is also supported by the fact that the class v' (in the above algorithm) has the same type as the class v. For **delete**, **remove** and **set** updates, they are propagated to both source classes if the modified instances are members of them.

6.7 Implementing the Add-Class Schema Change Operator in TSE

6.7.1 Semantics of the Add-Class Operator

The schema change operator defined by "**add_class** C_{add} [**connected_to** C_{sup}]" creates a class C_{add}, and makes it a direct subclass of the class C_{sup}. We require that the extent of C_{add} is implicitly empty and the new class to be a leaf class. The type of the class C_{add} is the same as the type of the class C_{sup}. If the **connected_to** clause is not specified, then the class C_{add} is connected to the system class **ROOT**.

6.7.2 The Algorithm for Mapping the Add-Class Operator to Views

TSE translates the "**add_class** C_{add} [**connected_to** C_{sup}]" operator to the following view specification:

```
{   for all the origin classes ($C_{origin}$) of $C_{sup}$ [18]
            create a base class $C_x$ as a direct subclass of $C_{origin}$;
      create a virtual class ($C_{add}$) based on the $C_x$ classes by
      following the same derivation procedures of deriving $C_{sup}$ from the $C_{origin}$ classes;
   }
```

[18] All the origin classes of a virtual class are found by recursively tracing back the *derivation relationships* until base classes are met. The definition is show in Section 3.4.

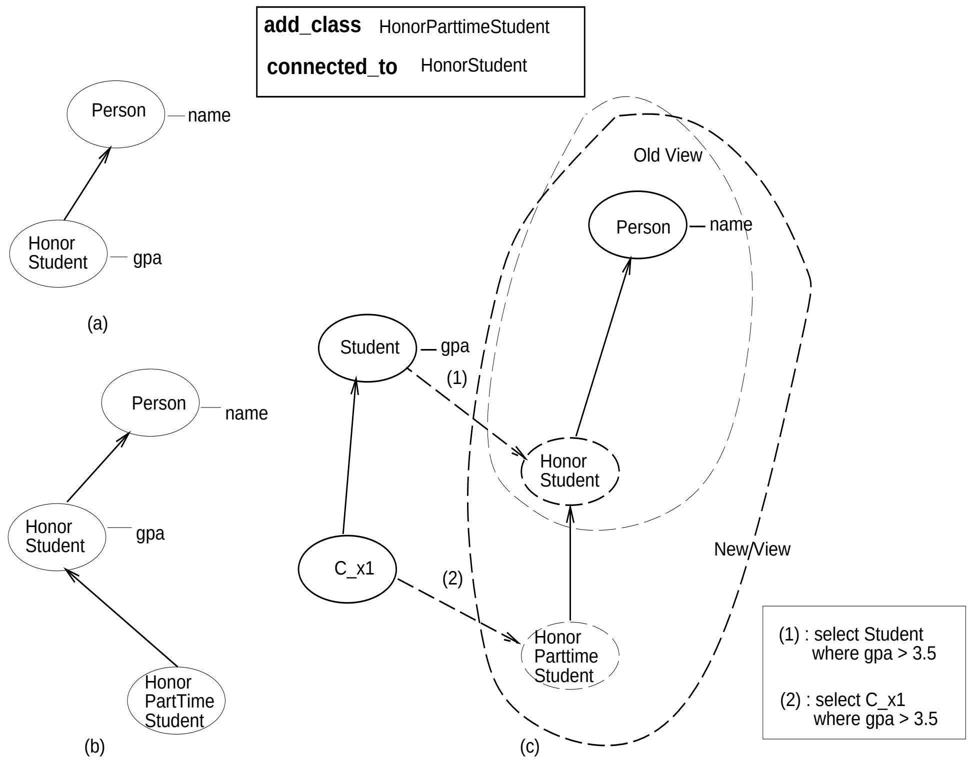

Figure 12: Schema Change for Adding a Class.

The above algorithm creates base classes C_x, which are direct subclasses of each C_{origin} class. The origin classes (C_{origin}) are the base classes found when we trace back through the chains of deriving the class C_{sup}. The virtual class C_{add} is derived by applying to C_x the same procedures as deriving the class C_{sup} from the base classes C_{origin}. The class C_{add} will be classified as a direct subclass of the class C_{sup}. The class C_{add} is selected to also belong to the new view schema.

Figures 12 (a) and (b) show an example of this change of the view schema. In this figure, the class $HonorParttimeStudent$ is created as subclass of the class $HonorStudent$. Figure 12 (c) shows the resulting global schema change. There may be more than one *origin* class, but for this example the class *Student* is the only *origin* class of $HonorStudent$. The dotted arrows represent the view class derivation relationships from the origin base classes to the virtual classes. The direction of the arrows are not necessarily super/subclass relationships, but rather indicate the direction of the derivation. Hence *Student* is not necessarily a superclass of *HonorStudent*. It just happened in this example. In addition, we don't know whether the origin classes have all the properties of the derived virtual classes because the properties might have been added during the derivation process.

We now create a base class (in our case, C_{x1}) as a subclass of *Student*. We then apply the same derivation process that has been applied to derive $HonorStudent$ from *Student* to create the virtual class C_{add} from C_{x1} (the derivation is labeled (1) in the figure). So, the derivations labeled (1) and (2) are the same except that the origin base classes are different. We can guarantee that the $HonorParttimeStudent$ class is a subclass of the class $HonorStudent$ because the origin classes of $HonorStudent$ are all superclasses of the origin classes of $HonorParttimeStudent$.

6.7.3 Verification of the Translation Process

Proposition: *The algorithm in Section 6.7.2 correctly simulates the semantics of the desired schema change of adding a new class as described in Section 6.7.1.*

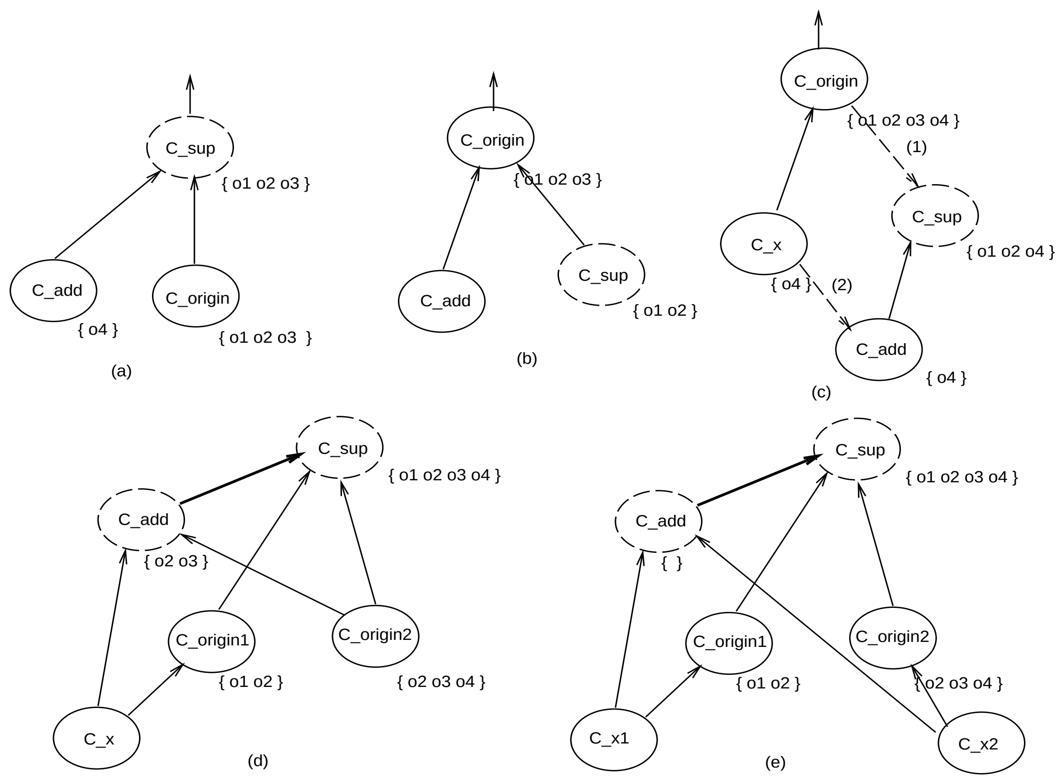

Figure 13: Alternative (undesirable) Approach of Examples of Schema Changes for Adding a Class.

We explain our add-class procedure by detailing the problems associated with alternatives, initially plausibly appearing approaches To achieve the desired add-class semantics of this operator, we could have attempted to create a base class as a direct subclass of C_{sup}. C_{add} then would inherit its type and membership constraints from C_{sup} as in Figure 13 (a). Assume as in this example figure that the class C_{sup} is a **hide** virtual class of the class C_{origin}. When we insert an object instance (o4) into the class C_{add}, then the object (o4) would not be visible in the class C_{sup}. C_{sup}, being a **hide** class derived from C_{origin}, would continue to have the same extent { o1,o2,o3 }. This violates the constraint of the genearlization concept that a superclass's extent is a superset of subclass's extent.

Next assume that we just could have made C_{add} a subclass of C_{origin} as in Figure 13 (b). Suppose C_{sup} were a **select** virtual class derived from the class C_{origin}. Then, we can't make the new class C_{add} a subclass of C_{sup} because the new class can't be a subset of C_{sup} because the *select predicate* is not associated with the new class.

Again, we try to achieve the semantics of the add-class operator by creating a new base class (C_x) as a subclass of C_{origin}, and by deriving class C_{add} using the same derivation procedures as from C_{origin} to C_{sup} shown in Figure 13 (c). Its purpose is to impose the constraint of class C_{sup} on the class C_{add}. It also guarantees that the new class C_{add} is a subclass of C_{sup}. To further investigate this scheme, especially for the case where there are more than one origin class, we consider the example in Figure 13 (d) where C_{sup} is a **union** virtual class based on two source classes $C_{origin1}$ and $C_{origin2}$. Following the above scheme, the class C_x is created as subclass of $C_{origin1}$ and the class C_{add} is created as **union** of classes C_x and $C_{origin2}$. Clearly, the class C_{add} is a subclass of class C_{sup}. Also when we insert a new object instance ($o5$) into the class C_{add}, the insert will be propagated to either (or both) source classes (C_x and $C_{origin2}$) and the new instance will be visible in class C_{sup}. So, the subset property of is-a relationship is also shown to be assured. However, as we can see in the figure, when we add a new class C_{add}, the class is not empty but is filled with some instances from its two source classes, i.e., instances $o2$ and $o3$. This clearly is not desirable.

To remedy this problem, we create a base class C_x for all the origin base classes C_{origin} as a subclass of each origin class, and derive the class C_{add} from the classes C_x by the same procedure as deriving the class C_{sup} from its origin base class C_{origin}. In the example of Figure 13 (e), the classes C_{x1} and C_{x2} are created as subclasses of the origin base classes $C_{origin1}$ and $C_{origin2}$, respectively. Then, the class C_{add} is created as **union** of classes $C_{origin1}$ and $C_{origin2}$. Because all the classes on which the extent of the new class is based on are empty, the new class is also empty.

In the above discussion, the class C_{add} is guaranteed to be a direct subclass of the class C_{sup} because the derivation procedure of the class C_{add} is the same as that of the class C_{sup} except that C_{add}'s origin classes are subclasses of C_{sup}'s origin classes. The classification algorithm will thus position it as a direct subclass of C_{sup}. So, it exactly simulates the semantics of subclassing of a base class (even though we use a virtual class for C_{add}).

6.7.4 Updatability

By assumption, the virtual class *HonorStudent* is updatable, and the derivation process labeled (1) maintains updatability. Because we apply the same procedure to derive the class *HonorParttimeStudent* from C_x, the class *HonorParttimeStudent* is also guaranteed to be updatable.

6.8 Implementing the Delete-Class Schema Change Operator in TSE

6.8.1 Semantics of the Delete-Class Operator

The schema change operator defined by "**delete_class** C" corresponds to removing the class C from a view schema. Its semantics are that the local extent of the class C (if any) [19] is still visible to its superclasses, and the local properties (if any) are still inherited to its subclasses. In fact, it doesn't affect any other class in its view except that it is dropped from the view schema. An operator with these semantics is already provided by a command of the view specification language, **removeFromView** *aClass*, of MultiView [19]. More complex version of this delete-class operator with the same semantics as the delete-class in Orion [3] can be achieved by composing several simpler change operators in our system. This will be shown in Section 6.9.2.

6.9 Constructing More Complex Schema Change Operators

The schema evolution capability of our system is not limited to the schema change operators discussed so far. This section shows how some of the popular schema operations can be achieved by a combination of our primitive schema change operators. This idea of combining primitive operators to achieve complex schema evolution has also been discussed by Zicari [31]. For example, we can achieve the complex schema change of inserting a class between two existing classes or of deleting a class with the same semantics as that of delete-class in Orion [3] as follows.

6.9.1 Constructing the Insert-Class Schema Change in TSE

Semantics of the Insert-Class Operator: The schema change defined by "**insert-class** C_{insert} **between** C_{sup}-C_{sub}" creates a class C_{insert} as subclass of C_{sup} and superclass of C_{sub}. The type of the class C_{insert} is the type of the class C_{sup}. Initially, the class C_{insert} is empty, i.e., the local extent is empty and the global extent is equal to the global extent of the class C_{sup}.

The algorithm for Mapping the Insert-Class Opeartor to Primitive Operators: TSE translates the "**insert-class** C_{insert} **between** C_{sup}-C_{sub}" operator to the following script composed of primitive schema change operators supported by our TSE system:

```
{   add_class $C_{insert}$ connected_to $C_{sup}$;
    add_edge $C_{insert}$-$C_{sub}$; }
```

Figure 14 shows an **insert-class** example. First, Figure 14 (b) shows the view schema after the class C_{insert} is

[19]The local extent has meaning only for base classes. For virtual classes, the local extent is equal to the global extent.

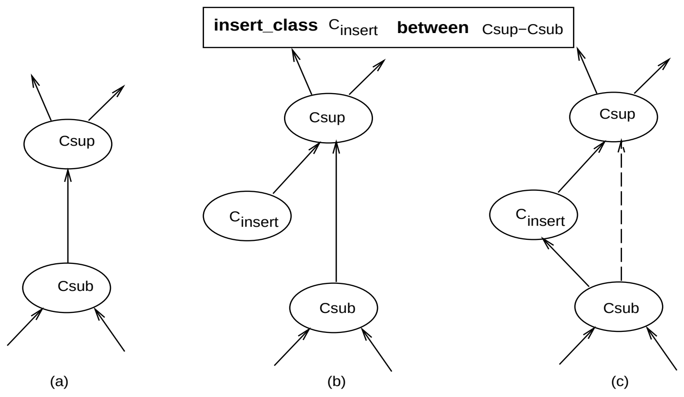

Figure 14: Applying Primitive SE operators for inserting a class.

added to the schema in Figure 14 (a). The Figure 14 (c) shows the view schema after adding an edge C_{insert}-C_{sub}. Note that the is-a relationship C_{sup}-C_{sub} becomes redundant in Figure 14 (c), and could be removed after the running classification algorithm, if so desired.

Verification of the Translation Process and Updatability: The resulting schema is updatable and other views are not affected by the schema change operations because the whole procedure consists of only primitive schema change operators. Since the primitive operators guarantee the updatability and view preservation, the macro of *inserting* a class also satisfies the properties.

6.9.2 Constructing the Delete-Class-2 Schema Change in TSE

Semantics of the Delete-Class-2 Macro: The schema change defined by "**delete_class_2** C_{delete}" deletes a class C_{delete} so that subclasses of the class C_{delete} no longer inherit local properties of C_{delete}, and the local extent of class C_{delete} is no longer visible to the superclasses of the class C_{delete}. If a local property of the class C_{delete} was overriding some property, the overridden property will be restored in the subclasses of the class C_{delete}.

The Algorithm for Mapping the Delete-Class-2 Macro to Primitive Operators: TSE translates "**delete_class_2** C_{delete}" operator to the following script composed of primitive schema change operators:

```
{    for all direct subclasses ($v$) of $C_{delete}$
     {    delete_edge $C_{delete}$-$v$;
          for all direct superclasses ($u$) of $C_{delete}$
               add_edge $u$-$v$; }
     for all superclasses ($w$) of $C_{delete}$
        delete_edge $w$-$C_{delete}$; }
```

The example run of the algorithm is shown in Figure 15. First, all the edges incipient to class C_{delete} are deleted, and each subclass of C_{delete} is connected all the superclasses ($S1$ and $S2$) of C_{delete}. The Figure (b) shows the intermediate schema. Now, all the edges from the class C are deleted, and the resulting schema is shown in the Figure (c). As you can see, the class C_{delete} is now connected solely to **OBJECT**, and it would be removed from the view [20].

Verification of Translation Process and Updatability: In Figure 15, the subclasses of C ($C1$ and $C2$) are connected to all the superclasses of C. It means that all local properties of the class C_{delete} are hidden from $C1$ and $C2$. In addition, only the direct extent of class C is now hidden from the superclasses of C. So, the semantics of

[20]The operation of removing a class from a view is supported by *the view specification language* in MultiView [22].

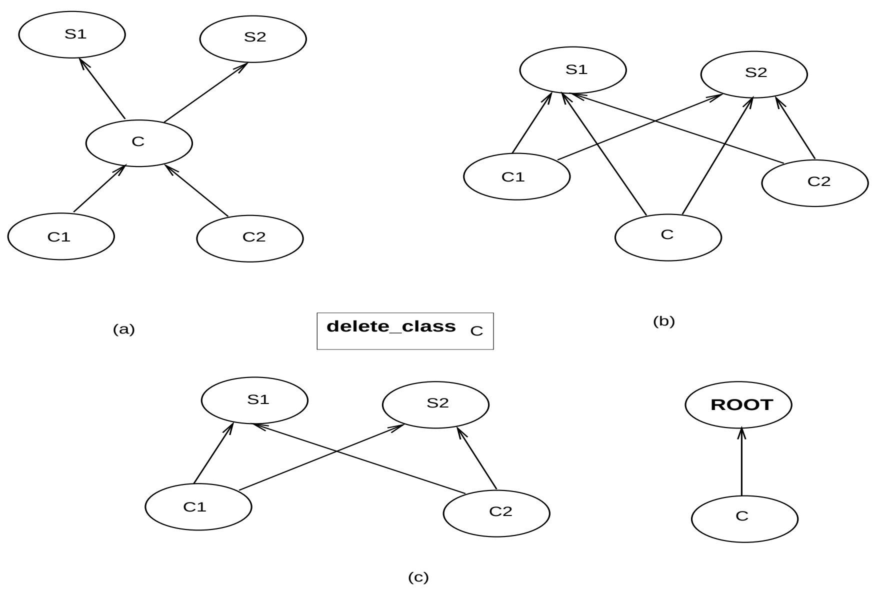

Figure 15: Applying primitive operators for deleting a class

this operation is achieved. The updatability and view preservation is automatically guaranteed because the macro only consists of our primitive operators.

7 Version Merging Using Views

Differing from other schema version systems [8, 7, 11], which keep track of constituting classes for each schema version, the version system of TSE is simple to implement – having only to keep track of view names that correspond to schema versions. In addition, it doesn't have to copy instances for each version, because instances are kept under one global schema and shared by all views defined on the global schema. Another advantage is that TSE system does not permit duplicate classes. When a duplicate class is created, it is detected by the classification algorithm [21], even if it has a different name. The existing class will replace the newly created duplicate one.

Sometimes, the user may want to merge two versions into one version schema in order to take advantage of improvements made in both schema versions. This merging process is likely to be very complicated in other schema version systems such as Orion [8], Goose [11] and Encore [27]. First, if instances have been copied for each version, all instance versions (duplicates) with the same object identity should be merged into a single instance. Second, the two separate schemas must be combined into one consistent schema, integrating also their generalization and aggregation hierarchies. This requires for instance that the taxonomical position of each class in the combined schema is determined, and that identical classes with different names as well as distinct classes with the same names must be found by this integration process.

These difficulties of version merging can be easily solved in the TSE system. In the TSE system, instances associated with different view schemas are never duplicated for other view schemas but rather are kept as unified objects under one global schema. Thus, the merging of object instances is by design a non-issue in our system. The integration of two schemas into one is also automatically achieved in our system, since the classification algorithm integrates all virtual classes into one consistent global schema graph [17]. In our system, it is easy to determine based on the global schema whether the same named classes are really identical. Similarly, differently named classes of separate schemas are said to be identical if they are identical in the global schema.

Figure 16 shows an example of merging two view schema versions. Two users are assigned the view schema VS.0 in Figure 16 (a). The upper branch of the figure corresponds to the schema VS.1 created by adding attribute *register* to the *Student* class in VS.0 by one user, and the lower branch VS.2 by adding attribute *student_id* to the

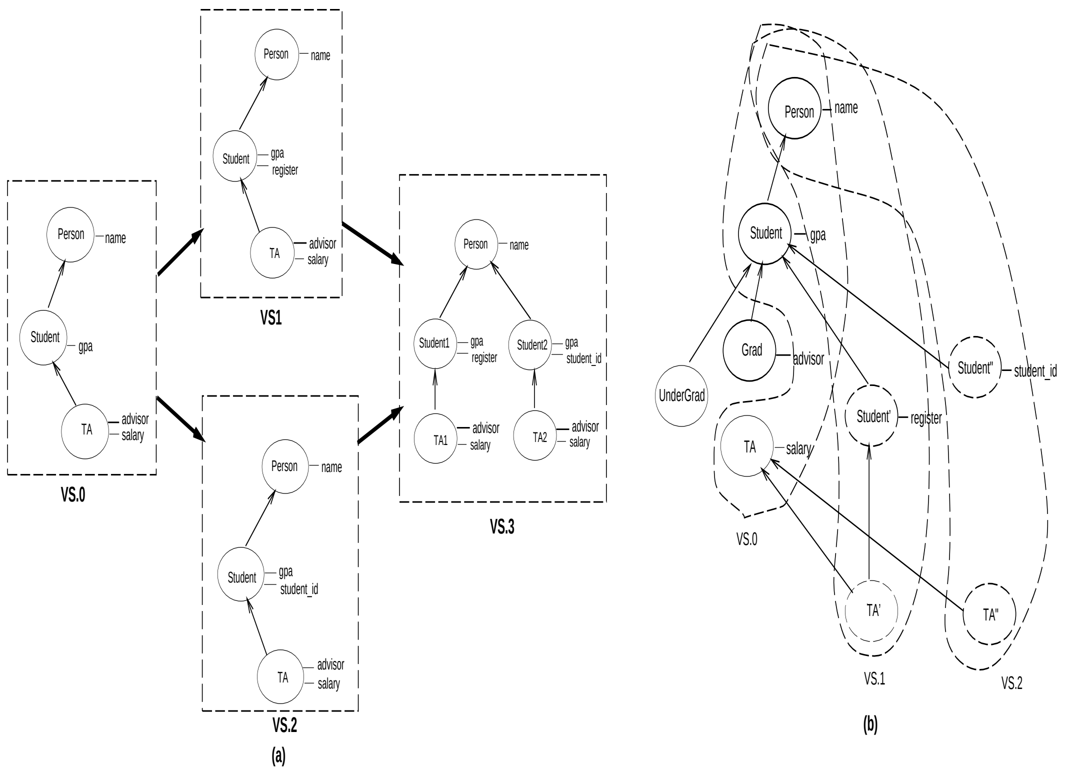

Figure 16: Merging Two Schema Versions in TSE

Student class in VS.0 by the other user. If another user wants to utilize both newly added attributes *register* and *student_id* for developing an application, then simply adding another *student_id* attribute to the VS.1 version will not be desirable. It would create duplicate fields for the same attribute, causing a waste of storage space and a potential source of inconsistencies and errors.

In our system, there are two better ways to fulfill the user's requirement. First, the user could explicitly select the two student classes, *Student'* and *Student''*, from the global schema shown in Figure 16 (b) using the *view specification language* and construct the desired integrated schema from the selected classes using the *view schema generation* algorithm provided by MultiView [19]. Second, the user can explicitly request a merge of the two versions VS.1 and VS.2 as depicted in Figure 16 (a). Then, the system collects all classes of VS.1 and VS.2 and integrates them into a single view schema VS.3. When merging VS.1 and VS.2, the *Person* classes of VS.1 and VS.2 are found to be identical, since they correspond to the same class in the global schema. However, the *Student* class of VS.1 is found to be distinct from the *Student* class of VS.2 even though they have the same name *Student*. Because they are actually different classes, they should have distinct names when they coexist in VS.3. In the example of Figure 16 (a), the problem is solved by adding version numbers to the class names. The user can of course rename them within the context of VS.3, if desired.

8 Related Research

The continued support of old programs when performing schema evolution has been recognized as a key issue in the literature [27, 8, 7, 14]. This section compares proposals addressing this issue using the following set of criteria for comparison:

- **sharing** of objects by different schemas (application programs): It refers to the capability to access the object instances from a schema version independent from under which schema version the object instances were originally created.

	sharing	effort required by user	flexibility of composing schema	subschema evolution	combination of views with schema change
Encore	yes	must create exception handler	yes	no	no
Orion	no	nothing particular	no	no	no
Goose	yes	keep track of class versions for each schema	yes	no	no
CLOSQL	yes	must create update/backdate functions	yes	no	no
Rose	yes	nothing particular	yes	no	no
TSE system	yes	nothing particular	no	yes	yes

Table 2: Comparison of Related Work.

- **manual effort** required: It refers to the necessary user's responsibility for changing the schema. Some systems require the user to provide the exception handlers to resolve the type mismatches between the underlying object instance representation and the schema to access it [27]. Others require the user to keep track of class versions for each valid schema [7].
- **flexibility** to build a new schema from class versions: It refers to the capability of composing various schemas by combining class versions. Some systems allow the user to build a schema from class versions only if the constituting classes are consistent with each other.
- **subschema evolution:** It refers to the capability to confine the effect of schema evolution to a subgraph rather than propagating the effect to many, possibly unnecessary, classes of the schema. A schema change is generally expensive and might require extensive database reorganization at the physical level. Evolving only the necessary subschema will thus improve the efficiency in terms of computation time and storage overhead.
- **combination of views with schema change:** The combination of these two capabilities offers several advantages. First, if a user can't specify the necessary schema change through a customized user view, then she has to *guess* the necessary changes on the underlying base schema and then specify a view to obtain the desired interface. Second, the content-based derivation power of views could be exploited to support more complex schema customizations as possible with the graph manipulation operators typically supported by schema evolution.
- **version merging:** It refers to support for merging multiple versions into one integrated consistent schema. As discussed in Section 7, by the virtue of being based on a view system using an integrated global schema as backbone, the TSE system provides a simple, yet elegant solution to version merging. To the best of our knowledge, other schema evolution database systems have not addressed this issue.

Table 2 shows the comparisons of our TSE system with other systems such as Encore [27], Orion [8], Goose [7, 11], CLOSQL [15] and Rose [14]. Typically, these other systems so not use the view approach, rather they utilize more traditional versioning concepts. They typically construct new versions of the schema as well as of the object instances, with instances being assigned to the schema version under which they have been created. Another important issue not addressed by these systems is the subschema evolution. It deserves more attention considering that most application programs run on some portion of the schema rather than on the whole global schema, and schema evolution is a very expensive procedure. We solve this problem by specifying the schema change directly on a view rather than on the global schema.

In a recent SIGMOD record article by Tresch and Scholl [29], the authors also advocate views as a suitable mechanism for simulating schema evolution. They state that schema evolution can be simulated using views if they are *not* capacity augmenting. In this paper, we go one step further, however, by putting forth that capacity augmenting schema change is necessary for truly extensible systems and that thus view systems must be extended accordingly rather than restructuring change capabilities. We also present an architectural solution to this data reorganization problem using the object-slicing paradigm. They present several simple examples rather than describing general-purpose algorithms for generating views to simulate schema evolution, as done in our paper. Furthermore, they do not show how to achieve conventional schema evolution operations that are more graph-manipulation oriented, such as adding/deleting an is-a relationship and adding/deleting a class.

Zdonik et al.'s approach towards type changes in the Encore System [27] is to keep different versions of each type, and to bind objects to a specific version of the type. Objects of different versions can be accessed by providing *exception handlers* for the properties that the types of the object instances do not contain. When a new attribute is added to a type, old versions of the type have to be provided with an exception handler for the case when a new program accesses *undefined* fields of old instances. However, it is both labor-intensive as well as difficult to provide semantically meaningful exception handlers. Because the schema is not versioned, a virtual version of the schema is constructed as a lattice of versioned types – with one version instance per type. This approach forces the user to manage the virtual versions of schemas by keeping track of which versions of types belong to which virtual versions of the schema.

The schema version mechanism proposed for *Orion* by Kim and Chou [8] keeps versions of the whole schema hierarchy instead of the individual classes or types. Every instance object of an old version schema can be copied and converted to become an instance of the new version schema. Usually, the old objects are frozen to be non-updatable and only new objects can be updated under the new schema. Object instances are thus not truly shared among the different schema versions. This approach doesn't allow backwards propagation. Suppose, for example, that a user deletes an object under a new version. If the user then operates under an old version schema, the object will still be visible. By adopting the view mechanism as foundation for our approach, the object instances are shared by all views, independently from the order in which these view schemas were created. This removes the inconsistency caused by not allowing back propagation in the schema version approach.

Kim et al. [7, 11] propose the versioning of individual classes instead of the entire schema. A complete schema is constructed in *Goose* by selecting a version from each class. This gives flexibility to the user in constructing many possible schemas, but it also results in the overhead of figuring out whether a given schema is consistent.

Another class versioning approach CLOSQL, proposed by Monk [15], provides update/backdate functions for each attribute which convert the instances from the format in which the instance is stored to the format that an application program expects. In such a system, the user's responsibility would be great even if the system provides the default conversion functions. In addition, the computation time for conversion might be a significant overhead. Lastly, extensions for handling new stored attributes have not yet been dealt with.

9 Conclusion and Future Work

In this paper, we present a solution to the problem of schema evolution affecting existing programs. We propose that schema changes should be specified on a view schema rather than the underlying global schema. Then, a new view reflecting the semantics of the desired schema change is computed automatically by our TSE system to replace the old one. In addition, by associating all objects with a single underlying schema (the global schema), we solve the problem of sharing the persistent objects among all versions of the schema — independently from the schema version under which they were created.

To support the view technology required for our approach, MultiView [19] is chosen because it generates updatable views and complete view schemas rather than just individual view classes. In this paper, we have made several extensions to MultiView to successfully support view evolution, in particular, we added capacity-augmenting capabilities. We have identified multiple classification as a key feature required of capacity-augmenting view systems in order to support schema evolution. To the best of our knowledge, none of the emerging view systems does provide this feature. In this paper, we presented our solution that addressed this problem using an object-slicing approach. The TSE object model has been successfully implemented on top of Gemstone, providing the necessary features of multiple classification and dynamic data restructuring to the TSE system.

We have demonstrated our approach on a comprehensive set of schema evolution operators, i.e., those typically supported by OODBs[27, 11, 15, 14, 10, 31, 16]. As a result, we have also shown that object-preserving algebra operators, as provided by MultiView, are sufficient for supporting a comprehensive set of schema changes. Most importantly, the resulting view schemas are updatable, because views generated by an object-preserving algebra have been shown to be updatable.

Some more complex schema evolution operators, such as partitioning a class and coalescing classes, can most likely not be simulated by an object-preserving algebra. Since they may require more powerful algebra capabilities, the resulting views may no longer be guaranteed to be updatable with generic update operations. Thus this issue of updatability of such object-generating views represents a challenging open problem. We also want to develop optimization strategies for update propagation in our TSE system. This is important because the update on a virtual class may have to be propagated through long chains of dependent classes.

Acknowledgements: We thank Harumi A. Kuno, who has implemented the current *MultiView* prototype system, for her help and advice. Without her software and expertise, this work could not have been possible.

References

[1] S. Abiteboul and A. Bonner, "Objects and Views," in *Proc. ACM SIGMOD*, pp. 238–247, 1991.

[2] F. Bancilhon and W. Kim, "Object-Oriented Database Systems: In Transition," in *SIGMOD RECORD*, volume 19, December 90.

[3] J. Banerjee, H. Chou, and W. Kim, "Data Model issues for Object-Oriented Applications," *ACM Trans. on Office Information Systems*, vol. 5, no. 1, pp. 3–26, January 87.

[4] J. Banerjee, W. Kim, and H. Kim, "Semantics and Implementation of Schema Evolution in Object-Oriented Database ," in *ACM SIGMOD*, 1987.

[5] D. Fishman, "Iris: An Object Oriented Database Management System," in *ACM Trans. on Office Info. Sys.*, volume 5, pp. 48–69, January 1987.

[6] S. Heiler and S. B. Zdonik, "Object views: Extending the vision," in *Proc. IEEE Data Engineering Conf.,Los Angeles*, pp. 86 – 93, February 1990.

[7] H. Kim, " Issues in Object-Oriented Database Schemas," in *Diss., Dept. of Comp. Sci., Univ. of Texas at Austin, TR-88-20*, May 1988.

[8] W. Kim and H. Chou, "Versions of Schema For Object-Oriented Databases," in *Proc. 14th VLDB*, pp. 148–159, 1988.

[9] H. A. Kuno and E. A. Rundensteiner, "Developing an Object Oriented View Management System," in *CAS-CON*, November 1993.

[10] B. S. Lerner and A. N. Habermann, "Beyond Schema Evolution to Database Reorganization," in *OOPSLA*, pp. 67–76, 1990.

[11] M. Magdi, A. Morsi, Navathe, and H. Kim, *"Object Oriented Approach in Information System"*, Elsevier Science Publishers B.V.(North-Holland), 1991.

[12] S. Marche, "Measuring the Stability of Data Models," *European Journal of Information Systems*, vol. 2, no. 1, pp. 37–47, 1993.

[13] J. Martin and J. J. Odell, *Object-Oriented Analysis and Design*, volume 1, Prentice Hall, A Simon & Schuster Company, Englewood Cliffs, NJ 07632, first edition, 1992.

[14] A. Mehta, D. L. Spooner, and M. Hardwick, "Resolution of Type Mismatches in an Engineering Persistent Object System," in *Technical Report,Rensselaer Design Research Center and Computer Science Dept. in Rensselaer Polytechnic Institute*, 1993.

[15] S. Monk and I. Sommerville, " Schema Evolution in OODBs Using Class Versioning," in *SIGMOD RECORD, VOL. 22, NO.3*, September 1993.

[16] D. Penney and J. Stein, "Class Modification in the GemStone Object-Oriented DBMS," in *Proc. 2nd OOPSLA*, pp. 111–117, 1987.

[17] E. A. Rundensteiner, "A Classification Algorithm For Supporting Consistent Object Views," in *Information and Computer Science Department, Univ. of California, Irvine, Technical Report, 92-50*, June 1992.

[18] E. A. Rundensteiner, "Design Views for Synthesis: Provide Both Uniform Data Integration and Diverse Data Customization," in *EECS, Univ. of Michigan, Tech. Rep. CSE-TR-148-92*, November 1992.

[19] E. A. Rundensteiner, "*MultiView:* A Methodology for Supporting Multiple View Schemata in Object-Oriented Databases," in *18th VLDB*, pp. 187–198, August 1992.

[20] E. A. Rundensteiner, "Research Initiation Award: An Object-Oriented Extensive View System for Computer-Aided Design Application," in *NSF proposal*, 1993.

[21] E. A. Rundensteiner, "Tools for View Generation Object-Oriented Database," in *CIKM*, pp. 635–644, November 1993.

[22] E. A. Rundensteiner and L. Bic, "Set Operation in Object-Based Data Models," *IEEE Trans. on Data and Knowledge Engineering*, pp. 382–398, June 1992.

[23] E. Rundensteiner, "Object-Oriented Views: An Approach to Tool Integration in Design Environments," in *Diss., Info. & Comp. Sci. Dept. Univ. of Cal., Irvine, Fall*, 1992.

[24] M. Scholl and C. Laasch, "Updatable views in object-oriented databases," in *Proceedings of the Second DOOD Conference*, pp. 1–19, December 1991.

[25] M. E. Segal, "Personal Discussion about software interface evolution ," in *Member of Technical Staff, Applied Research, Bellcore*, 1994.

[26] D. Sjøberg, " Quantifying Schema Evolution," *Information and Software Technology*, vol. 35, no. 1, pp. 35–54, January 1993.

[27] A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," in *Proc. 1st OOPSLA*, pp. 483–494, 1986.

[28] M. Tresch and M. Scholl, "Implementing an Object Model on top of Commercial Database Systems," in *Proc.* 3rd *GI Workshop on Foundation of Database Systems*, May 1991.

[29] M. Tresch and M. H. Scholl, "Schema Transformation without Database Reorganization," in *SIGMOD RECORD*, pp. 21–27, 1993.

[30] M. Tresch and M. H. Scholl, " Meta Object Management and its Applications to Database Evolution," in *EDBT*, October 1992.

[31] R. Zicari, "A Framework for O_2 Schema Updates," in *7th IEEE International Conf. on Data Engineering*, pp. 146–182, April 1991.

[32] R. Zicari, "Primitives for schema updates in an Object-Oriented Database System: A proposal," in *Computer Standards & Interfaces*, pp. 271–283, 1991.

Appendix: Glossary of Terms

- **virtual classes**: Classes derived via an object-oriented query.
- **base classes**: Classes to be able to actually store instances.
- **source classes**: Classes on which the derivation for a virtual class is based (they can be both base or virtual classes).

- **global schema**: The schema integrating all base and all virtual classes
- **view schema**: The schema containing a subset of both base and virtual classes as required by a particular user.
- **view classes**: the classes in a view schema which can be both base and virtual classes.
- **attribute**: The state of an object
- **method**: The behavior of an object
- **property**: This refers to both attributes and methods
- **type**: A library of methods and instance variables defined for a class
- **extent**: The set of object instances belonging to a class.
