A Transparent Schema-Evolution System Based on Object-Oriented View Technology

Young-Gook Ra, *Member*, *IEEE Computer Society*, and Elke A. Rundensteiner, *Member*, *IEEE*

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. 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. We show that our system 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., both old as well as newly developed applications can continue to interoperate. This paper describes a solution approach of realizing the evolution mechanism as a working system, which as its key feature requires the underlying object-oriented view system to support capacity-augmenting views. In this paper, we present algorithms that implement the complete set of typical schema-evolution operations as view definitions. Lastly, we describe the transparent schema-evolution system (TSE) that we have built on top of GemStone, including our solution for supporting capacity-augmenting view mechanisms.

Index Terms—Schema evolution, object-oriented view system, interoperability, capacity-augmenting views, software legacy problem, object-oriented database.

1 Introduction

Schema evolution is an important issue in object-oriented database (OODB) research [3], [18], [19], [33], [10], [34], not only because data models are less stable than expected [16], but also because typical OODB application areas such as CAD/CAM and multimedia information systems are not well understood and require frequent schema changes. However, even with the schema-evolution support provided by most OODBs, schema updates on shared OODBs are problematic. This is because in a typical environment, a developer must consult with others to figure out the impact of a desired schema change on other programs. This decision process of even a small schema change is likely to be long and tedious. In current OODBs, schema update capabilities are hence more limited by their impact on existing programs rather than by the power of the supported schema change language.

To ease this problem, we advocate the development of a schema version system that preserves old schemas for continued support of existing programs running on the shared database. Our schema version mechanism, called the *transparent schema-evolution* (TSE) system, uses object-oriented view techniques to achieve this desired capability. The basic principle of our TSE 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 for other application programs.

In TSE, unlike in other systems [10], [9], the scope of a schema version (view) is not confined to the objects which have been created under this particular schema version. Instead any object in the database can be accessed and modified through any version of the schema, assuming the classes of the objects are included in the view schema. Thus, each object instance can be directly shared by different versions of the schema without having to version the object at the instance level. We achieve this for two reasons:

1) all objects are associated with a single underlying global schema, and
2) each version of the schema is implemented via a view specification defined on the global schema.

In other database systems, tools written against the old schema would have to be rewritten or at least recompiled against the new schema to work with the new tools. This legacy problem requires significant human effort and skill, and in practice sometimes prohibits necessary schema upgrades. In our approach, old tools don't have to be rewritten or even recompiled for schema evolution because the old schema is "alive" despite the schema change. In this paper, we present algorithms for the complete set of schema-evolution operations typically supported by current OODB systems [3], [18], [19], [33], [10], [34]. These algorithms and their successful realization into a system demonstrate the feasibility of view evolution for schema transformation. Furthermore, we show that a comprehen-

• *Y.-G. Ra is with the Software Systems Research Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: ygra@eecs.umich.edu.*
• *E.A. Rundensteiner is with the Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609. E-mail: rundenst@cs.wpi.edu*

Manuscript received 29 Dec. 1994; revised 30 Oct. 1995.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104398.

sive set of schema transformations can be realized with object-preserving view mechanisms rather than object-generating ones. This is significant, since the former have been shown to be generally updatable [31]—which is a key property for our TSE system. Our TSE solution requires capacity-augmenting[1] views, i.e., views that augment the information content of the underlying schema. Another contribution of the work presented in this paper is to outline a strategy for extending the MultiView system [35] to also support capacity-augmentation using views.

The use of view mechanisms to achieve schema evolution has also recently been advocated by other researchers [33], [30], [4]. For instance, [33] presents several examples of schema changes that can be simulated by views. However they do not give general algorithms nor any implementation solutions. The O_2 view paper [30] refers to the issue of schema evolution as a potential feature—but no details on how this could be achieved nor any examples are given. Bertino [4] is the closest work in the sense that it discusses how an individual class can be modified using view specifications in place of schema modification operators. However, the effect of such class modifications on its subclasses is not considered. In contrast to these prior works, we demonstrate the feasibility of a schema version mechanism using a view approach by presenting general algorithms for a comprehensive set of schema-evolution operators. We show that TSE realizes all schema changes traditionally supported by commercial OODBs, such as Ithasca [3], GemStone [6], and O_2 [34]. In fact, we show that an object-preserving query language is sufficient. Lastly, the TSE algorithms create a view schema that achieves the semantics of the schema change operations in the context of the complete inheritance hierarchy rather than only on individual classes.

An earlier conference paper [24] contains a preliminary discussion of the TSE approach, while we now present TSE algorithms for all typical schema-evolution operations, and also describe our implementation solution on a commercial DBMS (see Section 7). The remainder of the paper is organized as follows: In Section 2, we introduce the TSE approach with a comprehensive example. In Section 3, we present key advantages of our approach. In Section 4, we identify the requirements on OO view systems in terms of their support for simulating schema changes imposed by TSE. Section 5 introduces a view system, and discusses necessary extensions of the view system so to meet the requirements to support TSE. Section 6 presents the comprehensive set of algorithms to realize our TSE approach, and Section 7 discusses TSE implementation issues. Section 8 compares our work with related research, and Section 9 concludes this paper.

2 Overview of the Transparent Schema-Evolution Approach

2.1 The Basic TSE Approach

Typically, 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 or are not well-documented. The latter 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 algorithms.

To overcome these problems, we have developed the transparent schema-evolution system. 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 the schema change need is identified by a developer, she specifies the necessary schema change on her own view (see Fig. 1a). The database system computes a new view schema, which reflects the semantics of the schema change, and replaced the old view with a the new one (Fig. 1b). In short, rather than directly modifying the old schema, we will ‘simulate’ the requested schema change using a view schema. This process is transparent to the user, i.e., this virtual schema change will be indistinguishable from the direct schema modification. While the user perceives a real schema change, Fig. 1b shows the actual changes in the underlying database system.

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 also required—especially when the change is capacity-augmenting. In Fig. 1b, we see that the global schema is augmented with new stored data, when the addition of a new instance variable is requested through a view change. However, when a delete is requested, it results in a removal from the view (and its virtual data) but not a delete from the global database.

2.2 A Comprehensive Example of TSE

The overall approach is now explained via the university database example depicted in Fig. 2a. Assume a developer builds the view schema in Fig. 2b to serve as a customized database interface. After several application programs have been written against the view, she finds that each student object should also carry *register* information for recording the registration. Fig. 2c shows the modified view schema that she now wants. 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 personalized view, in this case *“add attribute register to Student class.”*

This requested change would be realized as follows by our system. First, the system generates the two virtual classes *Student’* and *TA’* that refine the *Student* and *TA* classes, respectively, with the new attribute *register* (Fig. 2d). The

1. A capacity-augmenting change is defined to be schema restructuring to enhance the information content of the schema.

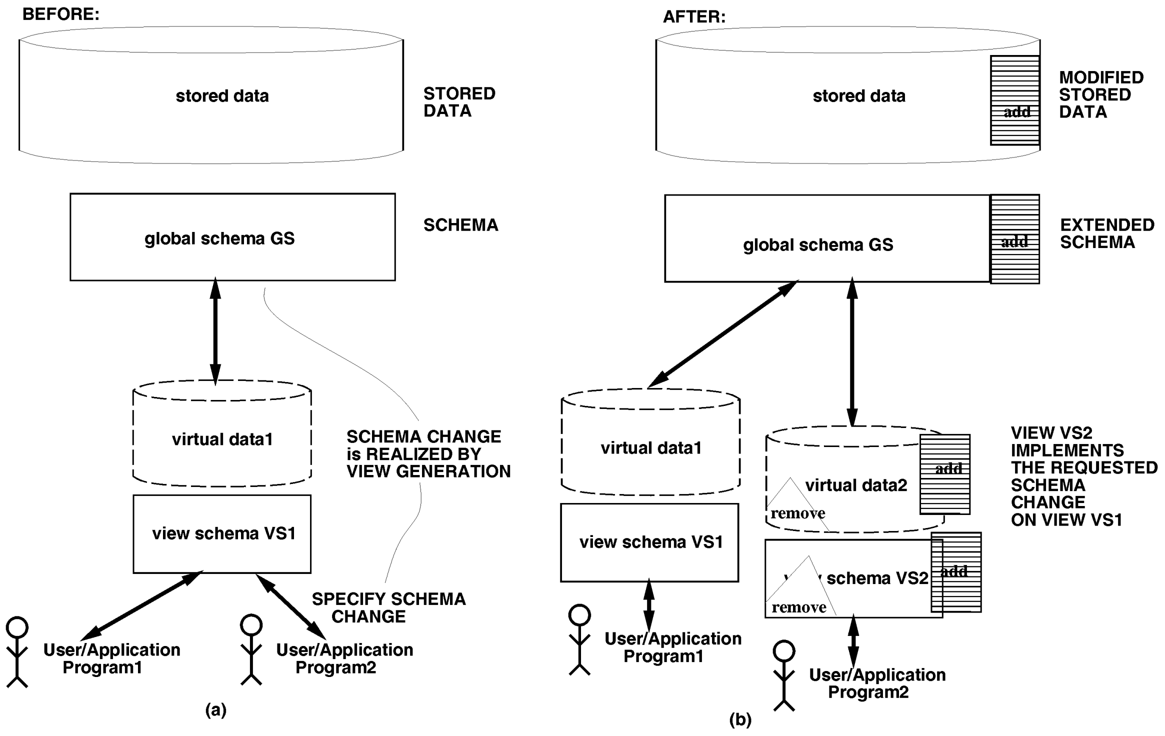


Fig. 1. The transparent schema-evolution approach.

dashed ellipses represent virtual classes, whereas the solid ellipses represent base classes. Second, the two virtual classes are integrated into the global schema by the classification algorithm of our view system (Fig. 2d) [27]. Third, the system selects the classes *Person*, *Student'*, and *TA'* for the new view. It 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 [26]. At last, the system replaces the old view with the newly generated view.

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, are not affected. This avoids the creation of unnecessary virtual classes for a possibly deep global class hierarchy.

3 The Advantages of Our TSE Approach

Our *TSE* approach offers the following three advantages over conventional approaches:

1) it allows all instances to be directly shared by each schema,
2) it integrates the restructuring power of views and schema evolution, and
3) it simplifies version merging.

Each of these advantages is further detailed below.

3.1 Instance Sharing Among Schema Versions

Most schema version systems [10], [9] utilize traditional versioning concepts where new versions of the schema and object instances are constructed, with every instance of the old schema being copied and converted to become an instance of the new schema version. Maintaining separate copies of the instances for each schema version makes it difficult and expensive to propagate the update on an instance through one schema version to all other copies of the instance under different versions. In our approach, because all object instances are associated with a single underlying global schema and each schema version is a view defined on the global schema, object instances are directly shared by all versions of the schema. There exists only one copy of each instance.

Fig. 3 shows the advantages of this direct instance sharing. Suppose we have a CAD tool *findCriticalPath* that is running against *Schema1*. Later, a new tool, called *Layout*, needs to be developed against *Schema1*, which requires physical layout dimensions of components. Identifying this new requirement, *Schema1* is modified into *Schema2* by adding new attributes *height* and *width* to the *Component* class. When the new *Layout* tool creates an instance *instance3*, it is also in the scope of *Schema1* and can be accessed from the old tool *findCriticalPath*. Conversely, an update on *instance4* by the old tool can simultaneously be seen by the new tool. So, the old and new tool can *interoperate* with each other, potentially achieving a higher level of functionality then either of them could achieve by themselves.

Sometimes it is preferable to upgrade the existing software to operate on the new schema directly, when

1) the overhead becomes unbearable,
2) the new schema becomes stable enough to know that there is no need to go back to the old schema, and
3) large enough labor force and time are available for rewriting of the application software.

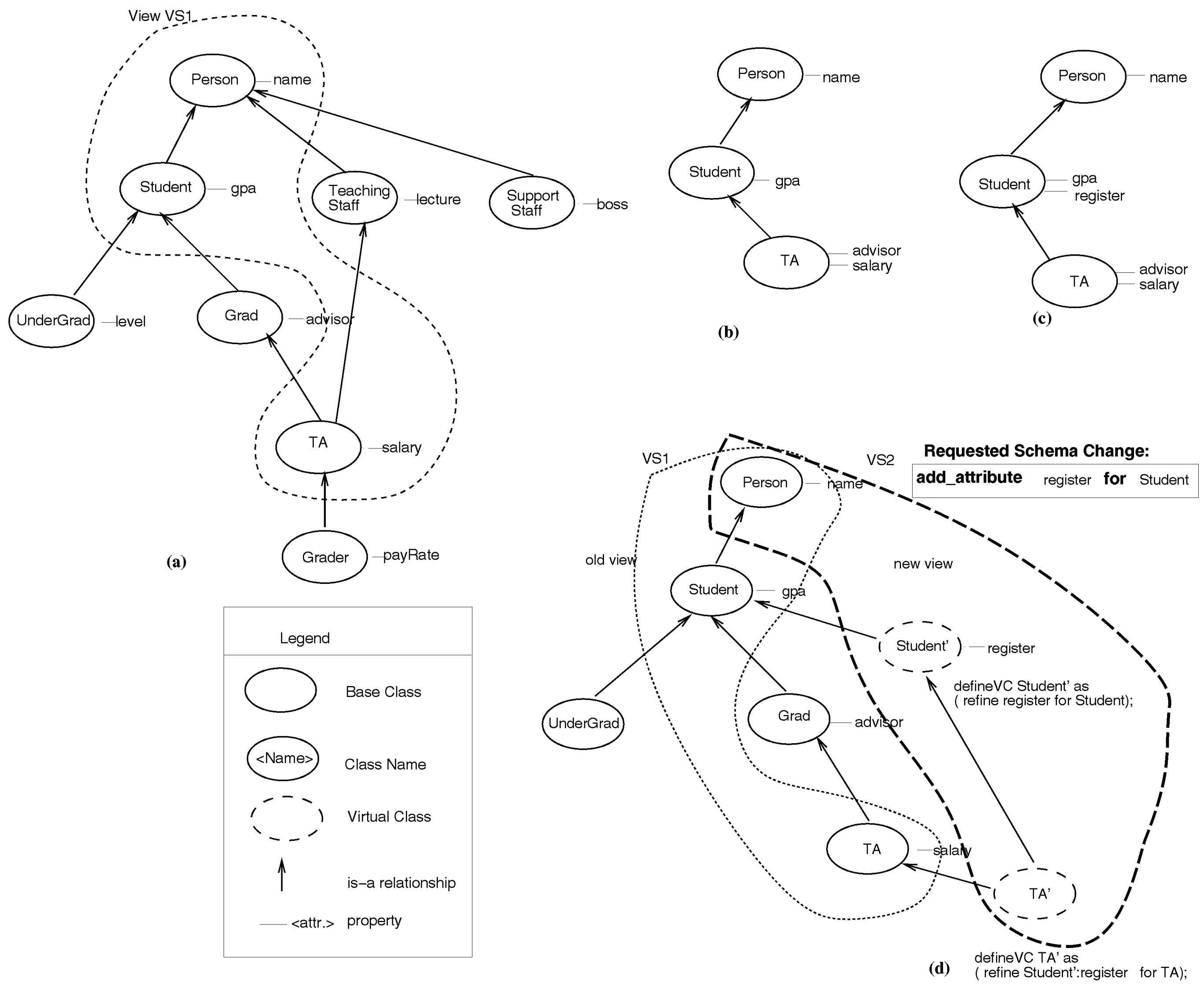


Fig. 2. Add-attribute schema change example.

In the latter case, the direct instance sharing provided by our view approach makes the schema transition *easy* and *smooth* by keeping the old schema "alive" as long as needed.

3.2 Increased Functionality by Integrating View and Evolution Mechanisms

Views allow a programmer to restructure the database to meet the specific needs of an application without affecting other application programs [12], [31], [25]. View mechanisms 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, they by definition do not support the addition of new stored information to the database.

We propose that many advantages could be gained by integrating the two capabilities of view support and schema evolution into one unified mechanism. First, a view mechanism would enable us 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.

Suppose the parking bureau plans to develop its own application programs on the university database of Fig. 2. Because only staff are allowed to park in university parking spaces, the parking bureau is only interested in the *TeachingStaff* and *SupportStaff* classes from the global database of Fig. 2. So, as shown in Fig. 4, they select these classes and create a virtual-class *Staff* using the **union** object algebra operator (see Section 5.1). Then, the *FulltimeStaff* virtual class is created by the **select** operator that selects instances from the *Staff* class that satisfy the predicate *status* = *'fulltime'*.[2] These two classes form a *Parking-View1* view that serves as a customized interface for the parking bureau.

However, customization by view specification alone does not support extensibility of the database. For instance, the parking office is likely to keep track of each staff's permit number that is not available nor can be derived from the global schema. Because, in TSE, schema change can be

2. For this example, we have assumed that *Staff* has the attribute *status*.

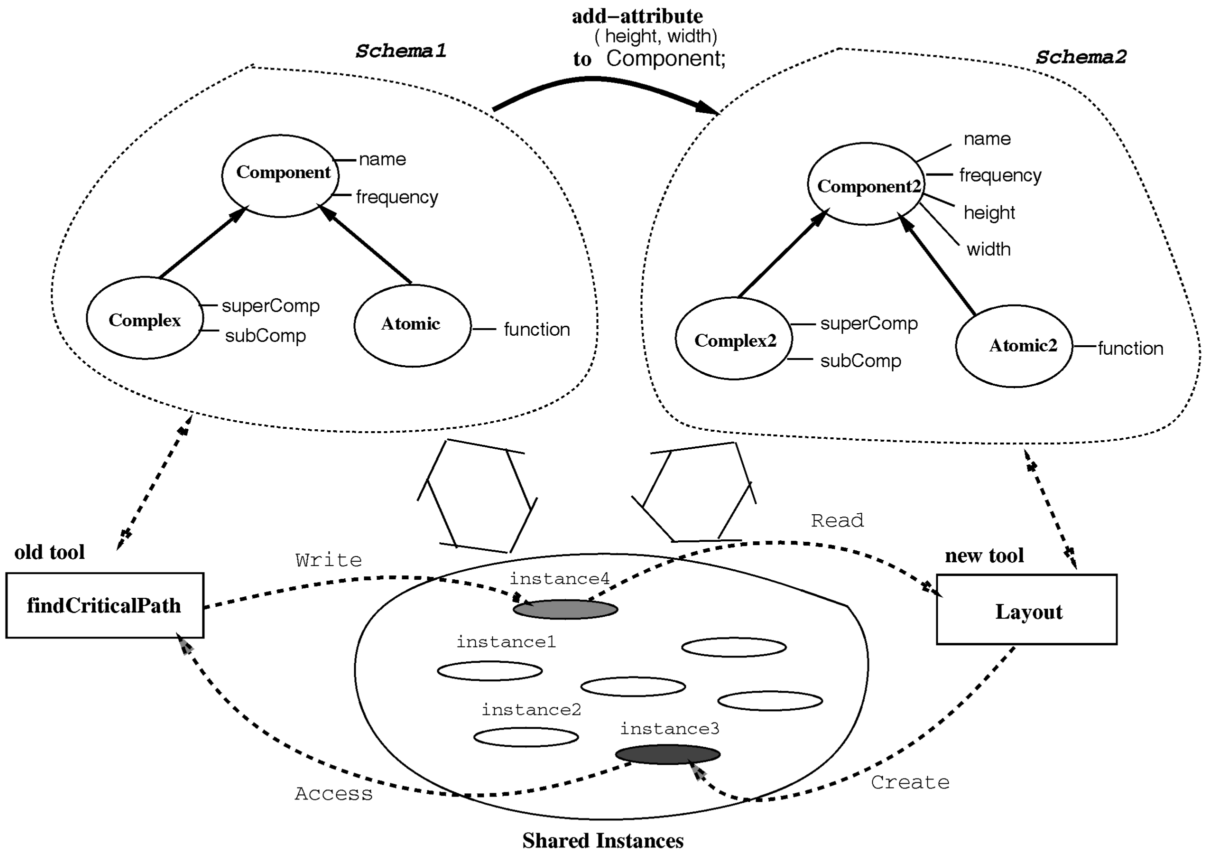


Fig. 3. Sharing instances between schemas in TSE.

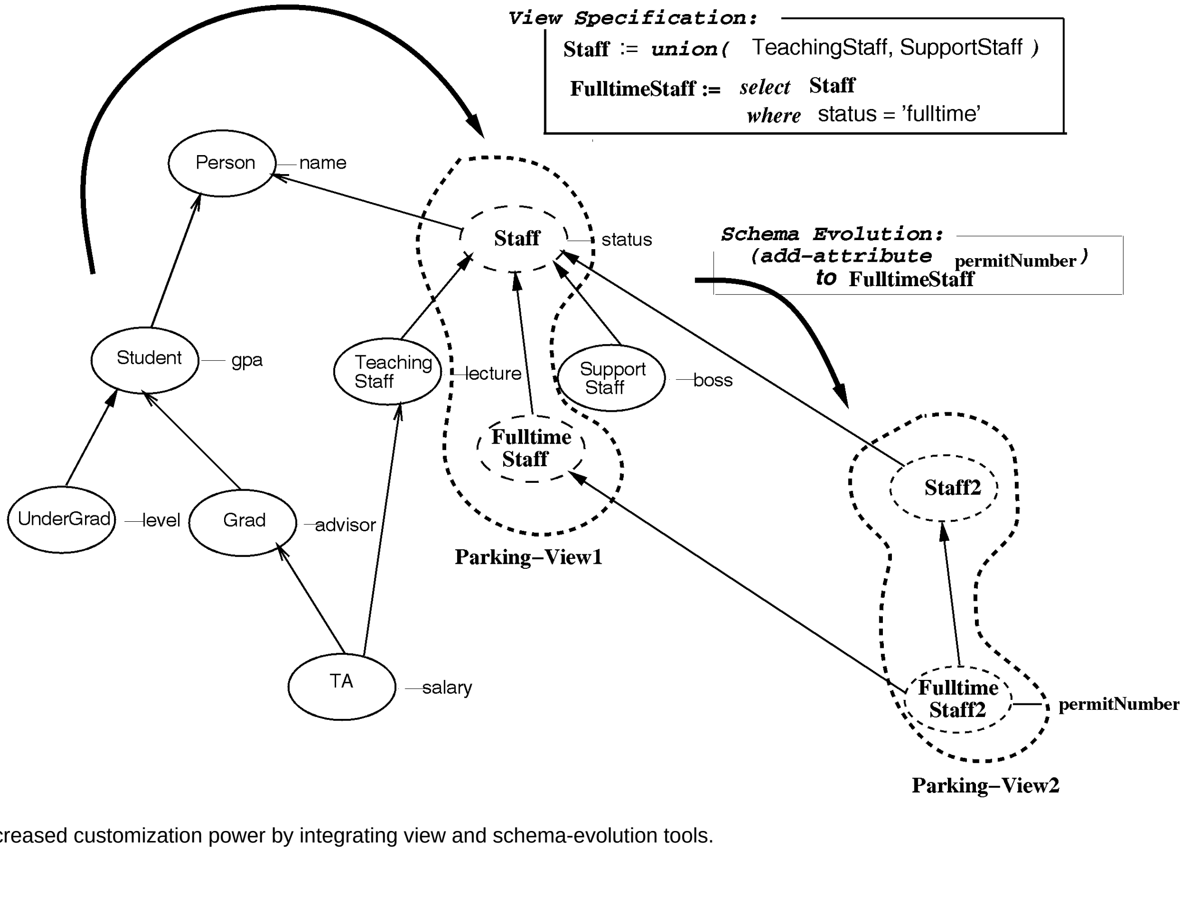


Fig. 4. Increased customization power by integrating view and schema-evolution tools.

initiated to a view, we are able to easily add a new stored attribute *permitNumber* to the virtual-class *FulltimeStaff*. This schema change operation would result in the *Parking-View2* view of Fig. 4.

Note that view or schema evolution alone can't provide restructured schemas like the *Parking-View2* view. This schema can be created only by an integrated system of view and schema evolution. By integration, we mean that schema changes can be applied to a view and that a view can be constructed from the changed schemas. We easily see that this integration supports two difficult problems of schema evolution: *subschema evolution* and *content-based evolution*. Subschema evolution is achieved by selecting desired classes to form a view and applying a schema change operation within the context of the view rather than the entire global schema. Our subschema change mechanism does not involve the classes outside the view regardless of the position of the classes in the global schema, as we will show in Section 6.

Content-based evolution utilizes a query language to restructure relevant objects based on their content rather than just performing a syntax (type)-based transformation as done by most popular current schema-evolution mechanisms [3], [20], [34]. For a simple example, suppose we want to keep track of the parking permit number only for fulltime staff. This is achieved by creating a select virtual class from the *Staff* class (assuming *Person* has an attribute called *status*) with the predicate *status* = *'fulltime'* and by adding a new attribute called *permitNumber* to the virtual class. Consequently, only instances of *Staff* whose status is fulltime have the additional stored attribute *permitNumber*. This is a situation that can't be achieved by schema evolution or by view mechanisms by themselves.

3.3 Version Merging in TSE

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 schema version systems [10], [32]. 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 be determined, and that identical classes with different names as well as distinct classes with the same names be found by this merging process.

These difficulties of version merging can be easily solved in the TSE system. In TSE, instances of different view schemas are never duplicated but are kept as single objects under one global schema. Thus, the merging of object instances is by design taken care of by our system. The integration of two schemas into one is also automatically achieved in our system, since the classification algorithm of our MultiView system integrates all virtual classes into one consistent global schema graph [27]. In our system, it is thus straightforward to determine based on the global schema whether the same named classes are really identical. Similarly, differently named classes of separate schemas are easily found to be identical if they refer to the same class in the global schema. The latter may occur since each view schema maintains its own name space.

4 Requirements for the TSE System

In order to achieve the transparent schema change approach outlined in Section 2.1, the following requirements must be imposed onto our view evolution approach:

1) The schema change applied 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 interoperate 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.[3]
2) 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. This is singled-out here because the majority of relational views have been shown not to be unambiguously updatable, and sometimes are restricted to be read-only. This is clearly not acceptable for our purpose.
3) Both old and new versions of a schema must be able to *share the same* (*persistent*) *data*, independently of the schema through which the data was 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.
4) 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. Thus, all commonly supported operations on classes, such as get-extent and create-instance, must still work as expected. 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 should not be apparent to the user. In other words, a view schema needs to have all properties of a regular base schema, and a virtual class needs to "act" just like a base class.
5) The view 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 a view definition language that pro-

3. This relates to the property of view independence defined in [25] that an OO view in our view system is not affected by global schema classification.

vides capacity-augmenting operators, and second, we need to provide a flexible architectural framework for supporting the dynamic augmentation of the schema and also of the underlying database.

We have developed solutions to the above problems based on object-oriented view mechanisms, as further outlined below.

5 MultiView: The View System Supporting TSE Realization

Since our view schema-evolution approach is based on object-oriented view techniques, we describe below the view system we have developed towards the specification and maintenance of views, called MultiView [25]. Unlike most 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 the class hierarchy. These unique characteristics effectively support TSE, 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 (requirement 4). In addition, MultiView offers the following features that made it suitable as foundation for our schema change approach. First, it generates updatable views [25] (requirement 2). Second, several of the view specification subtasks are already automated, and can be reused in our system, such as virtual-class insertion into the global schema [27] and generation of the view schema [26]. 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 [12], [21].

The MultiView system assumes an object model that corresponds to core object model features generally agreed-upon in the literature (for example, encapsulation, full inheritance, polymorphism, multiple inheritance, etc.) [25]. The glossary of terminology on our object model is given below:

1) **Virtual-Class**: Class derived via an OO query.
2) **Base class**: Class that stores base instances.
3) **Source class**: Class on which the derivation for a virtual class is based (they can be both base or virtual classes).
4) **Global schema**: The schema integrating all base and all virtual classes.
5) **View schema**: The schema containing a subset of base and virtual classes as required by a particular user view.
6) **View class**: The class in a view schema that can be both base and virtual classes.
7) **Attribute**: The state of an object.
8) **Method**: The behavior of an object.
9) **Property**: This refers collectively to both attributes and methods.
10) **Type**: The set of methods and instance variables defined for a class.
11) **Extent**: The set of object instances belonging to a class or its subclasses.

The task of creating a view schema is divided in the following tasks in MultiView:

1) the generation of customized virtual 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 schemas** composed of base and virtual classes.

For the first subtask, MultiView provides the user with an object algebra for class derivation [25], [28]. The second subtask is automated in MultiView by the classification algorithm [27]. The integration of virtual classes into one global schema provides many benefits, including detecting identical and thus redundant classes, sharing methods without code duplication among different views, and enabling efficient view schema generation. For the third subtask, we have developed a view schema generation algorithm for the construction of the view generalization hierarchy [26]. Automatic view generation [26] relieves the user of constructing the is-a hierarchy for each view schema and removes the potential inconsistencies in the view generalization hierarchy due to the potential mistakes of the user. Using a view system as foundation of TSE clearly offers a solution to requirement 3. MultiView has been shown to meet requirement 1 (which essentially corresponds to the view independency property shown in [25]).

5.1 Extending the 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 [25], [28], [27], [26], as foundation of our TSE system. While a complete description of the algebra can be found elsewhere [25], below we briefly introduce the operators (we also discuss how MultiView's object algebra has been extended to be capacity-augmenting as required for TSE (requirement 5)):

- 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*>. Currently, we only allow a simple predicate that is a conjunction of atomic predicates—defined as (*property* θ *value*), where θ is a comparison operator such as $\leq$, $\geq$, $>$, $<$, $=$, and $\neq$. 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 class is a supertype of the input class, as less properties are defined on the output. All objects of the input set are also members of the output set.
- Set operations. As the extent of classes are sets of objects, we can perform set operations as usual [28]. The criterion of duplicate elimination is object identity equality, not value equality as assumed in the relational model. We impose 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*>.

MultiView allows arbitrary queries composed by nesting these object algebra operators to serve as view definitions: **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 Fig. 5, 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.[4] In addition, the *age* attribute is hidden from the instances of *AgelessPerson*.

While view definition only derives virtual data as a function of existing stored data, we must also support the extension of the database 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 also support *stored attribute* extensions, 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. Our solution to this restructuring problem is outlined in the implementation section (Section 7).

Furthermore, we need to modify the **refine** operator defined for MultiView in order to create a virtual-class refining an existing attribute with a new domain. The new syntax now is:

refine <*attribute-name*>: *newD* **for** *C*.

The type of the output class is either a supertype of *C*, when the domain of <*attribute-name*> is generalized, or a subtype, when the domain is specialized. When the domain is generalized, the extent of the output class is the same as that of the input. But, when the domain is specialized, the extent is lessened by those objects whose attribute values <*attribute-name*> do not meet the new domain constraint *newD*.

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

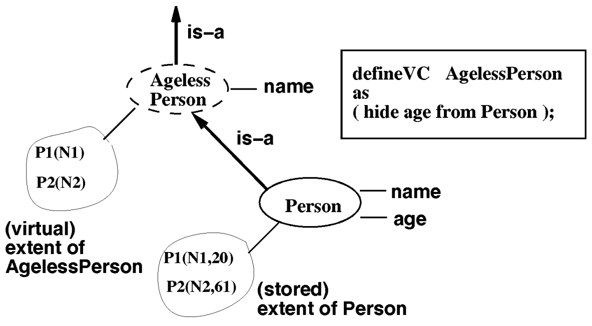

is-a
Ageless Person
name
P1(N1)
P2(N2)
is-a
Person
name
age
(virtual) extent of AgelessPerson
P1(N1,20)
P2(N2,61)
(stored) extent of Person
defineVC AgelessPerson as (hide age from Person);

Fig. 5. Example of virtual-class creation.

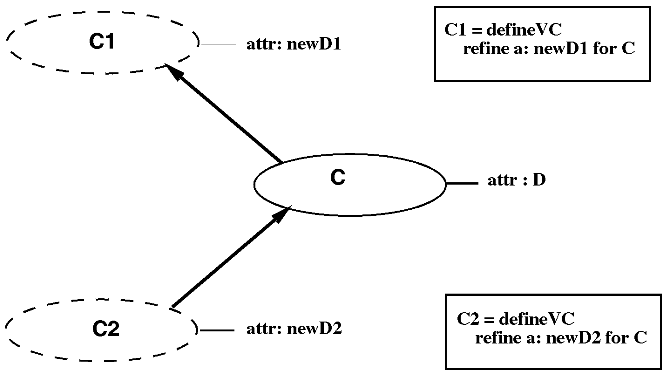

C1
attr: newD1
C1 = defineVC refine a: newD1 for C
C
attr : D
C2
attr: newD2
C2 = defineVC refine a: newD2 for C

Fig. 6. Examples of refining existing attributes with different types.

Fig. 6 shows both cases of the **refine** virtual-class creation. In the figure, *newD1* is more general than the original domain *D* of the *a* attribute in the *C* class. In this case, the **refine** class *C1* is a superclass of *C*. On the other hand, *newD2* is more specialized than *D*. In this case, the type of the **refine** class *C2* is a subtype of *C* and the extent is the same as or more restricted than that of *C*. Thus, *C2* is a subclass of the source class *C*.

5.2 Generic Update Operators

In OODBs, updates are generally performed using *type-specific* update methods. We, however, provide a set of *generic update operations* to extend type-specific updates, similar as proposed in other view systems [31]. Such generic update operations can either be used directly or, if desired, overridden by type implementers 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 define the **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 affected object gains a new type or loses one of its types:

- **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 to which they belong.
- **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*>.

Note that those generic operators can be used by type implementers 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.

5.3 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). This is the requirement 2) from Section 4. It has been shown that virtual classes created by an object-preserving algebra are updatable due to the one-to-one correspondence between base and virtual object instances [31]. 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. Ambiguity on how to realize the update can be characterized and resolved based on the type of virtual class, as discussed in [23].

Notice that virtual classes may, of course, be defined based on other virtual classes. We can now extend the results of updatability of virtual classes defined by one individual algebra operator [31] to virtual classes defined by a possibly complex algebra expression.

PROPOSITION 1. *Any virtual class defined by our object algebra defined in Section 5.1 is updatable in terms of the generic update operators such as* ***create****,* ***delete****,* ***remove****,* ***add****, and* ***set*** *defined in Section 5.2.*

INTUITIVE PROOF. We can view a virtual-class derivation as a DAG with multiple roots where the roots are base classes and all other nodes are virtual classes. The derivation 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 point(s) of the edge. We can label the edges by the name of the object algebra operator that defines the virtual class. An edge labeled as **hide**, **select**, or **refine** has only one starting node, while an edge labeled as **union**, **intersect**, or **difference** has two starting nodes.

Based on the above observation, we can 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. □

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. These base classes, called the *origin classes* of the virtual class, are the ones to which an update on the virtual class eventually will propagate to.

6 ALGORITHMS FOR REALIZING SCHEMA CHANGES IN THE TSE SYSTEM

One of the first object-oriented schema change taxonomies has been proposed by Banerjee et al. [3] for the Orion data model. Note that this taxonomy, adopted in most other schema-evolution research for OODBs such as Encore [32], Goose [9], CLOSQL [19], Rose [18], OTGen [15], and Cocoon [33], still corresponds to the most frequently used set of schema changes. In fact, most commercial OODB systems such as O_2 [34] and GemStone [20] 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. However, some operations in the Orion taxonomy [3] that are specific to the Orion data model are omitted, such as *change the order of inheritance priority*, *add a shared value*, etc. In summary, our set is chosen *comprehensive* enough to cover the Orion taxonomy and *general* enough to be applicable to a standard object model. Below, we present algorithms for translating a schema change into a number of view definition statements for each of these 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-domain* to *C*" augments the types of the class *C* and its subclasses C_{sub} with the new attribute *x* [3]. 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 an inherited existing attribute in class *C* with the same name *x*, the domain of *x*, *attribute-domain*, should be compatible with the domain of the existing attribute. If there exists an attribute with the same name x locally defined in C, 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. Also in this case, the *attribute-domain* should be compatible with the domain of the existing local attribute x in a subclass, i.e., the domain of the local attribute should not be more general than *attribute-domain*. Otherwise, this operation is rejected. For the multiple inheritance conflict, two same named properties can 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, Fig. 2b and Fig. 2c 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-domain* to C" operator to the following view specification:

```
{ if x already exists in C, reject this operation;
defineVC C' as (refine x : attribute-domain for C);
push C onto tmpStack;
while (tmp := pop tmpStack) ≠ 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 $C_{sub}$);[5]
            push $C_{sub}$ onto tmpStack; }
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

Fig. 7 shows the whole process of our schema change approach for this particular operator. Upon the schema-evolution request, "**add_attribute** *register* **for** *Student*", on the view schema of a user (Fig. 7a), the **TSE Translator** generates a set of view specification statements (Fig. 7b) 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 Fig. 7c. During the classification process, if the new virtual classes are augmenting the capacity of the source class, e.g., refining attributes, the objects of the source classes of the virtual classes need to be restructured so that they can carry the new data. This *dynamic restructuring* is performed by the **Dynamic Restructuring** module that takes **refine** virtual classes as input and returns a *status* to indicate the success / failure. If during the classification, 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 classes from the old view unless they have corresponding new *primed* classes.[6] In Fig. 7c, 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 [25]. The result is depicted in Fig. 7d. At last, the system replaces the old 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.

6.2 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'[7] in class C to guarantee the full inheritance invariant.[8] If the attribute x was overriding a same named attribute in class C, then the suppressed attribute is restored and propagated to the subclasses.

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 $C_{sub}$ of the class C, including C
            defineVC $C'_{sub}$ 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 for $C'_{sub}$); }}
```

The above algorithm creates virtual classes that hide the attribute x from the class C and its subclasses. When the attribute x has been overriding 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, Fig. 8a and Fig. 8b show

5. By creating this *refine-attribute* virtual class, every instance of C_{sub} should be restructured in order to carry the additional state for the attribute x.

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

7. The term '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.

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

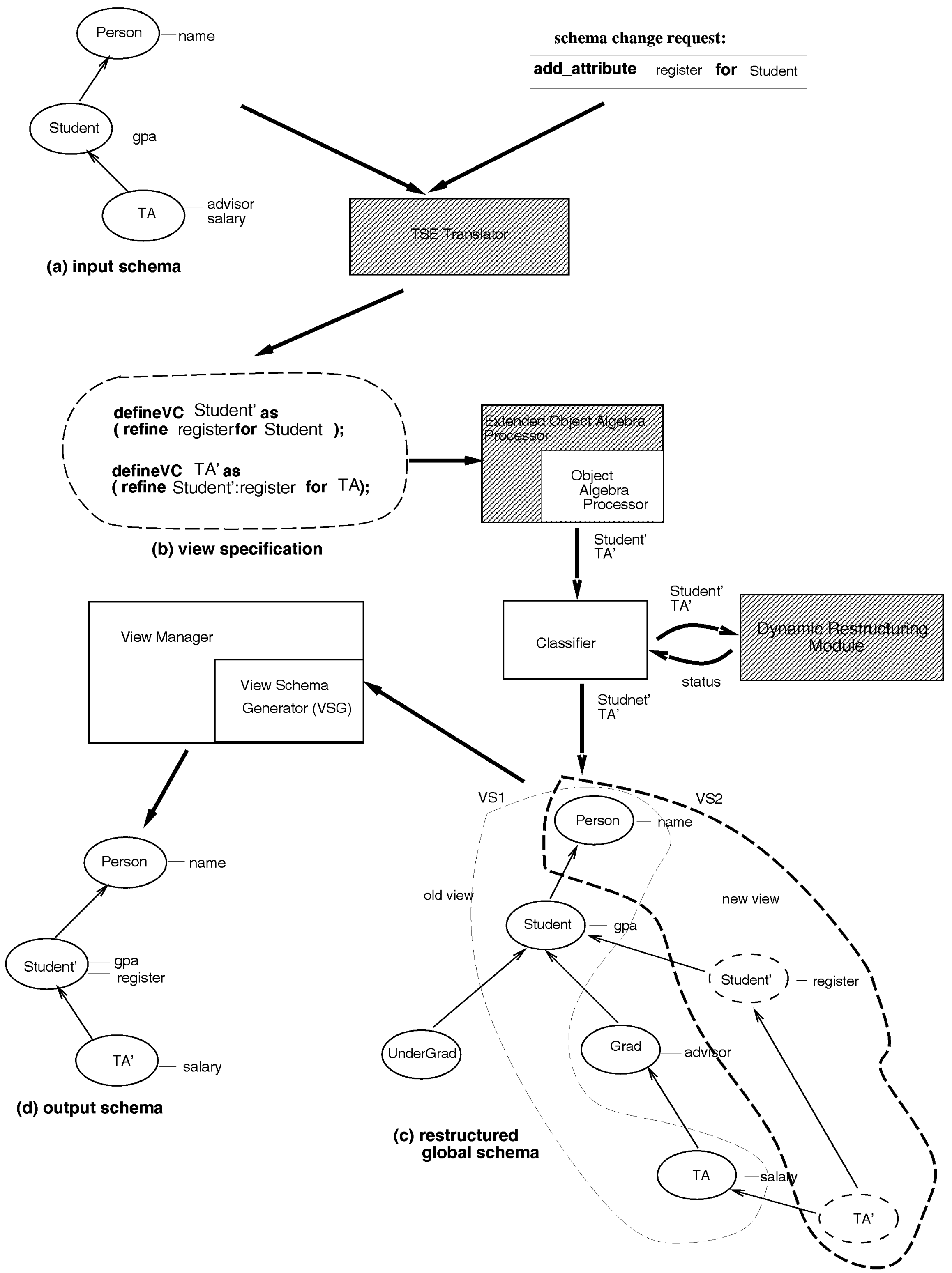


Fig. 7. Complete TSE procedure for a schema change.

an old and a new view schema, and Fig. 8c 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.

6.3 Implementing the Add-Method Schema Change in TSE

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

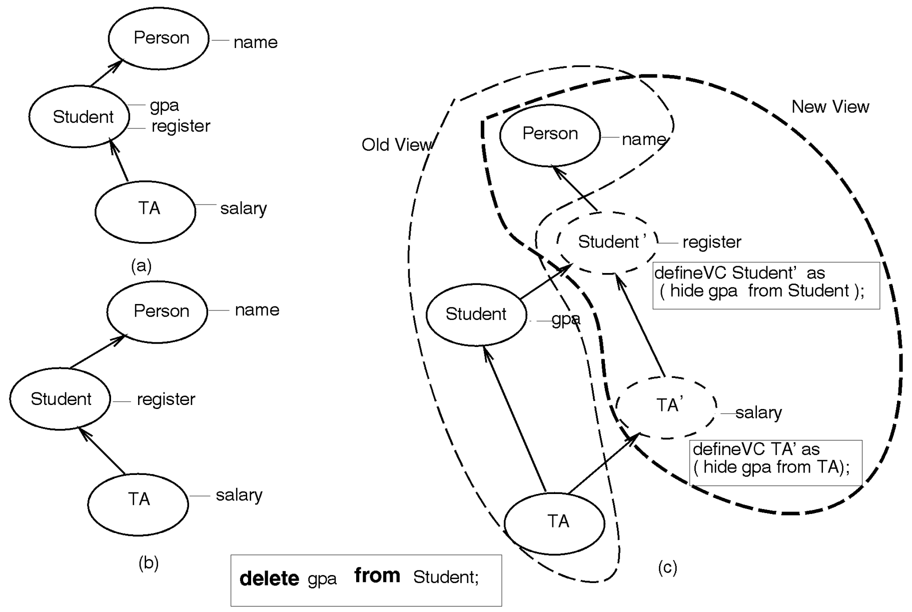


Fig. 8. Schema change for deleting an attribute.

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.[9] The algorithm for this operator is similar to that of the *add-attribute* operator except that the virtual classes created by the algorithm for this operation are refining methods instead of attributes and, thus, no object restructuring occurs. Hence, a detailed discussion is omitted here.

6.4 Implementing the Delete-Method Schema Change in TSE

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. The algorithm for this schema update is the same as that of the *delete_attribute* operator—except for no object restructuring to be performed during the TSE process (Fig. 7).

9. 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.

6.5 Implementing the Change-Domain of an Attribute Operator in TSE

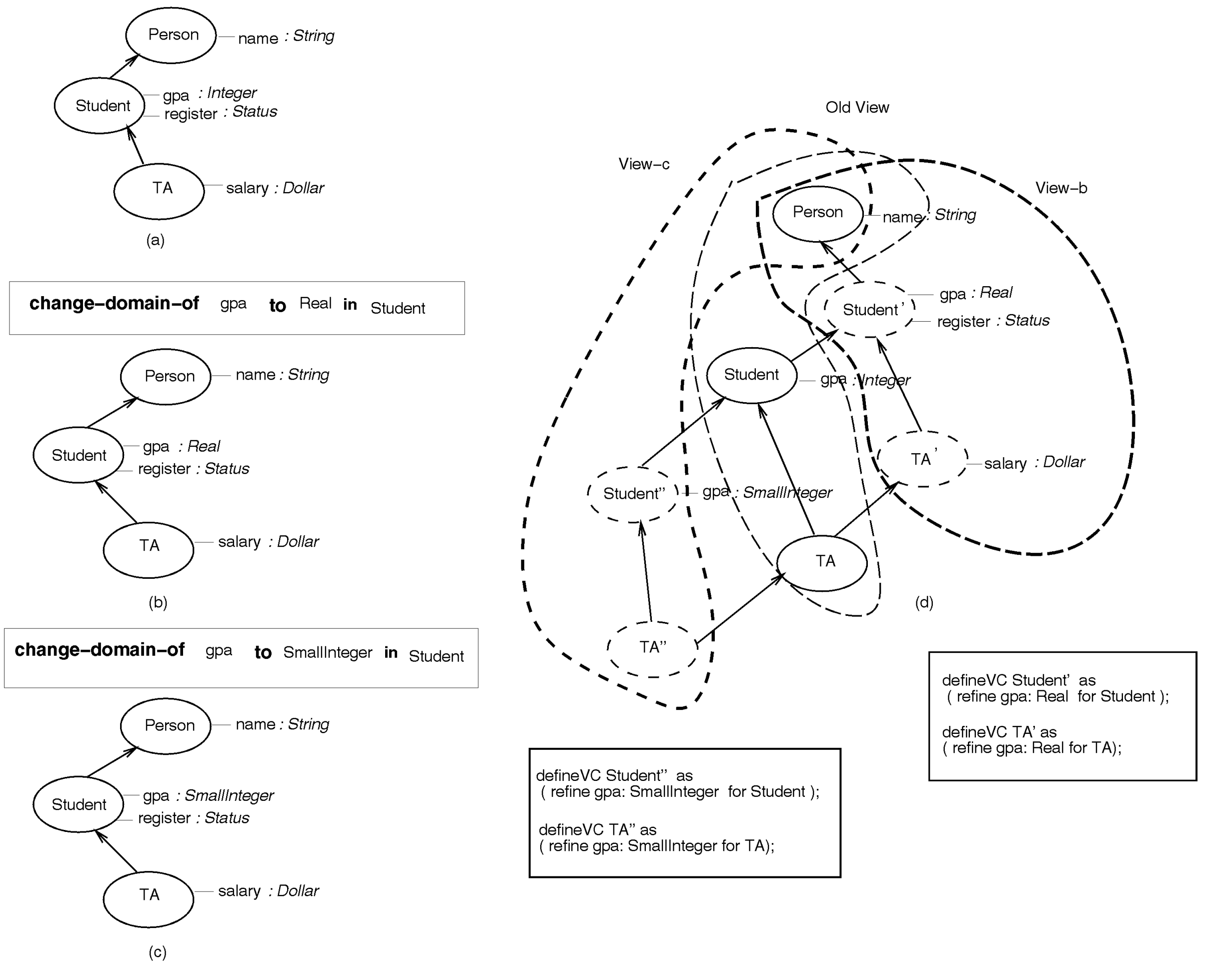
6.5.1 Semantics of the Change-Domain Operator

The schema operation defined by "**change_domain_of** *x* **to** *newD* **in** *C*" changes the domain of the attribute *x* of the class *C* to the new domain *newD*. This domain change is propagated to the subclasses of *C*, unless the domain of *x* is again overridden in some subclass. The domain of the attribute can either be specialized or generalized. However, if the attribute *x* is inherited, it can't be more generalized than the domain of *x* in the superclass of *C* in which *x* is defined.[10] Thus, there are two kinds of domain changes:

1) the new domain class is a superclass of the old one, and
2) the new domain class is a subclass of the old one.

Fig. 9 shows an example of this operation. The example of the first kind operation is "**change_domain_of** *gpa* **to** *Real* **in** *Student.*" This operation is initiated on the view schema shown in Fig. 9a and results in the schema shown in Fig. 9b. This output schema corresponds to the view *View-b* in the global schema Fig. 9d. As illustrated, *View-b* is constructed first by creating the virtual classes *Student'* and *TA'* by refining the *gpa* attribute with the *Real* type for *Student* and *TA* in the global schema, respectively. The example of the second kind operation is "**change_domain_of** *gpa* to *SmallInteger* **in** *Student.*" This operation is initiated in the view schema of Fig. 9a and results in Fig. 9c (or the *View-c* view in the global schema). For this view, we have created the *Student*" and *TA*" virtual classes by refining the *gpa* attribute with the *SmallInteger* type for *Student* and *TA* in the global schema, respectively. For the second case, some existing objects may no longer qualify for the new domain. Then, these objects can't be accessed from the *C* class after the operation.

10. This corresponds to semantics chosen by Orion [3].



```
change–domain–of  gpa  to  Real  in  Student
```

```
change–domain–of  gpa  to  SmallInteger  in  Student
```

```
defineVC Student' as
 ( refine gpa: Real  for Student );

defineVC TA' as
( refine gpa: Real for TA);
```

```
defineVC Student" as
 ( refine gpa: SmallInteger  for Student );

defineVC TA" as
( refine gpa: SmallInteger for TA);
```

Fig. 9. Schema change for changing the domain of an attribute.

6.5.2 The Algorithm for Mapping the Change-Domain Operator to Views

TSE translates the "**change_domain_of** *x* **to** *newD* **in** C" operator to the following view specification:

```
{ if $x$ does not exist in $C$, reject this operation;
defineVC $C'$ as (refine x: $newD$ 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 locally defined in $tmp$
        then { defineVC $C'_{sub}$ as (refine $C'$ : x for $C_{sub}$);[11]
            push $C_{sub}$ onto tmpStack; }
endwhile; }
```

This algorithm is almost the same as that of the *add-attribute* operation except that now we refine the classes with an existing attribute with a different type rather than with a new attribute as we do in the *add-attribute* case.

11. The instances of the C_{sub} class are restructured by refining the x attribute.

6.6 Implementing the Add-Edge Schema Change in TSE

6.1.1 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, leaving 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 Fig. 10a and Fig. 10b, 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.[12]

12. The term 'extent' used in this paper is implicitly assumed to be global extent and not local extent.

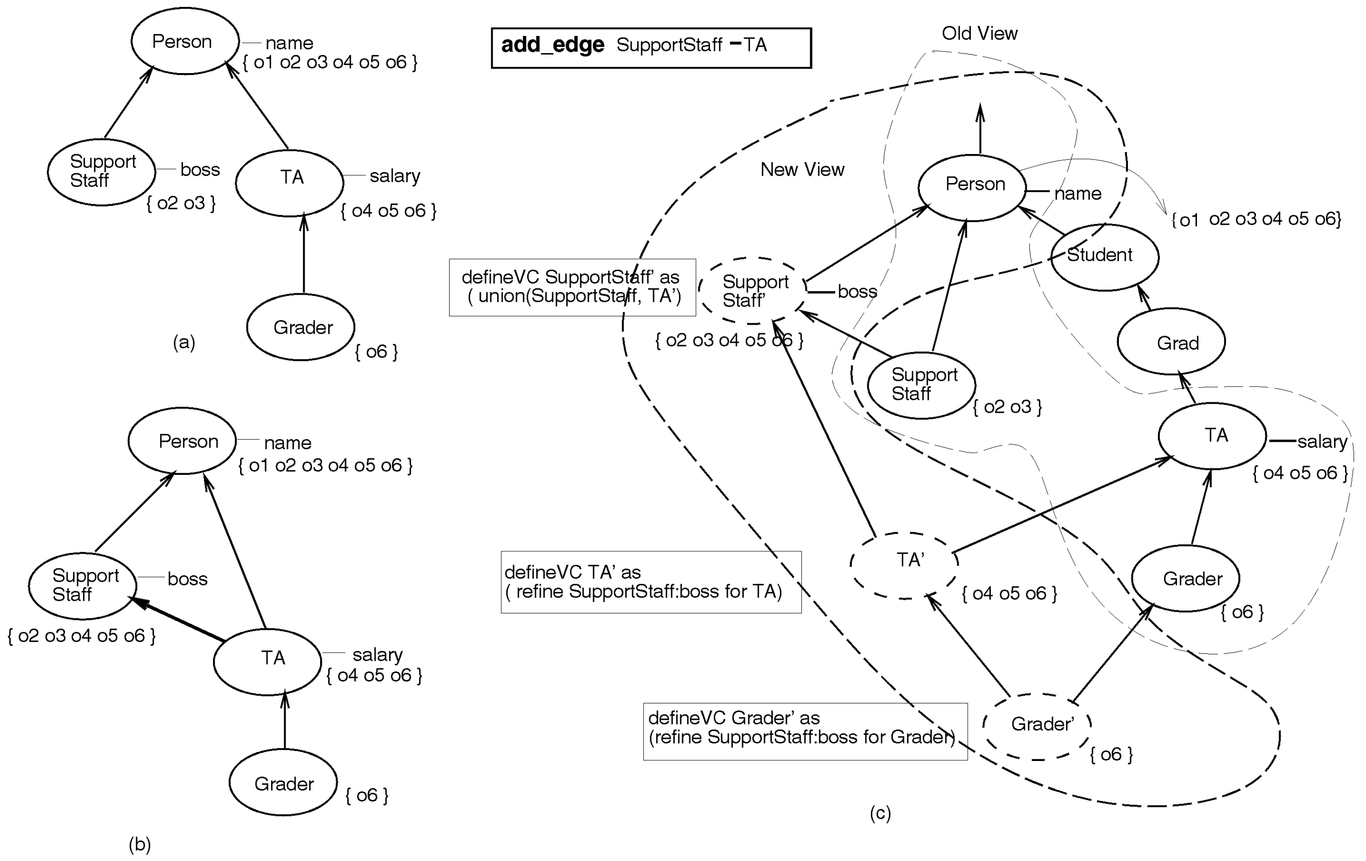


Fig. 10. Schema change for adding a generalization edge.

The extent of the class *SupportStaff* {o2 o3} is expanded to {o2 o3 o4 o5 o6}.

6.6.2 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 $C_{sup}$ for w);[13]
    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 statement adds properties of the class C_{sup} to the classes which now become subclasses of C_{sup} due to the added is-a relationship. In Fig. 10c, the classes *TA*' and *Grader*' are created with their type augmented by the property *boss* of *SupportStaff* (C_{sup}). The second 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 Fig. 10c, 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 P*erson* class before the schema change.

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

6.6.3 Updatability

The **union** operator could possibly be problematic because there are two source classes associated with each unioned virtual class. In Fig. 10c, 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 specified 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 will not cause undesirable side effects. 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.

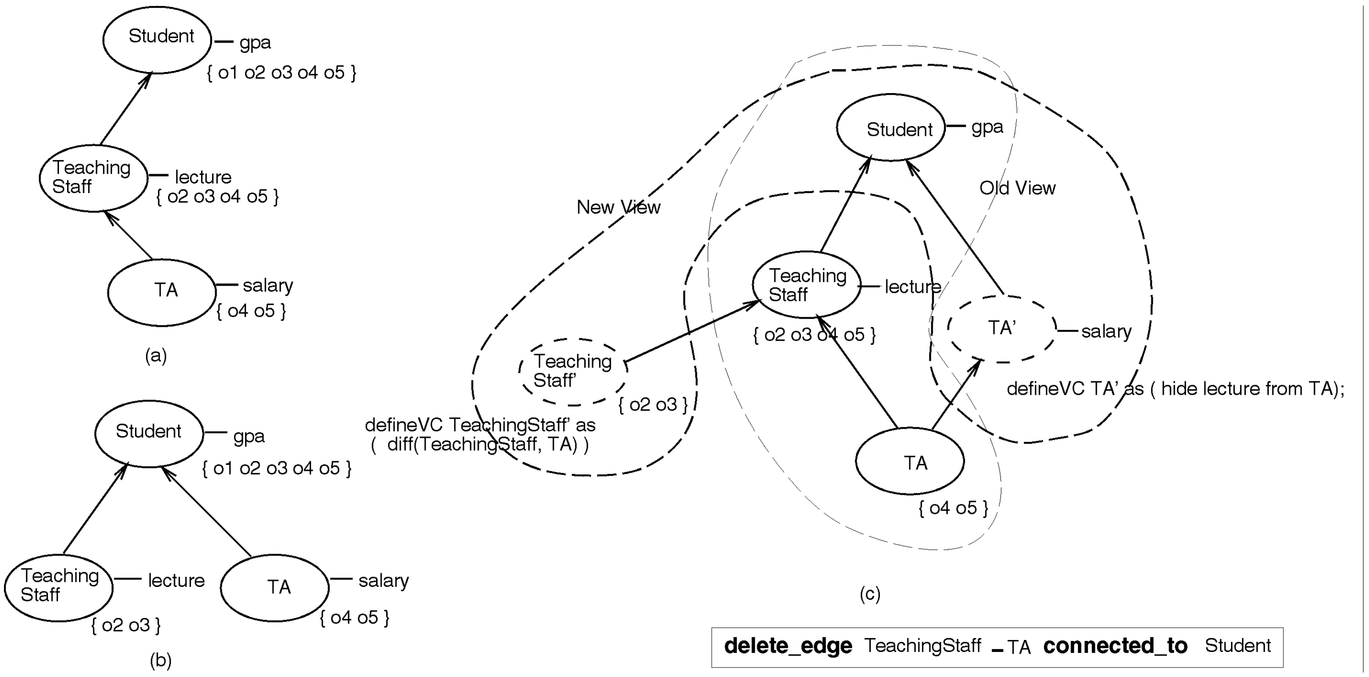


Fig. 11. Schema change for deleting an edge.

6.7 Implementing the Delete-Edge Schema Change in TSE

6.7.1 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**[14] to avoid disconnecting 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.

Fig. 11a and Fig. 11b 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}.

6.7.2 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:

```
{  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 Fig. 12. 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 removed from the class v. The **commonSub** (v, C_{sub}, C_{sup}-C_{sub}) procedure 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} is 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 preserved in spite of the fact that the **union** operation usu-

14. The ROOT class is the root class of the DAG structure.

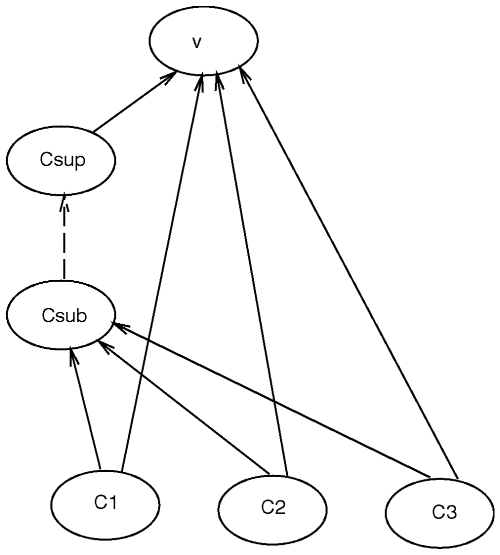


Fig. 12. An example of deleting an edge E.

ally constructs the type of a virtual class, which may possibly be different from any of the source classes. However, the **union** operations in the above algorithm create virtual classes whose types are the same as those of the first argument source classes, because the set of classes returned from **commonSub** are subclasses of the class *v*.

The **findProperties**(*w*, C_{sup}-C_{sub}) procedure returns the properties that have been inherited to the class *w* only through the is-a relationship C_{sup}-C_{sub}.[15] 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.

Fig. 11c 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 **commonSub**(*TA*, *TA*, *TeachingStaff-TA*) procedure returns the *TA* class. The **findProperties**(*TA*, *TeachingStaff-TA*) procedure returns the *lecture* property.

6.8 Implementing the Add-Class Schema Change Operator in TSE

6.8.1 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 C_{add} is a leaf class and that its extent is implicitly empty. 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**.

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

This operation of adding a class is complicated when the superclass is a virtual class rather than a base class, because intentional subclassing is not allowed for the virtual class.

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 $C_{sup}$[16]:
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;
  }
```

The above algorithm creates base classes C_x that 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.

Fig. 13a and Fig. 13b show an example of this change of the view schema. In this figure, the class *HonorParttime-Student* is created as subclass of the class *HonorStudent*. Fig. 13c 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 is coincidence 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*.

6.9 Implementing the Delete-Class Schema Change Operator in TSE

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)[17] 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

15. 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 procedure returns the property.

16. 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 5.3.

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

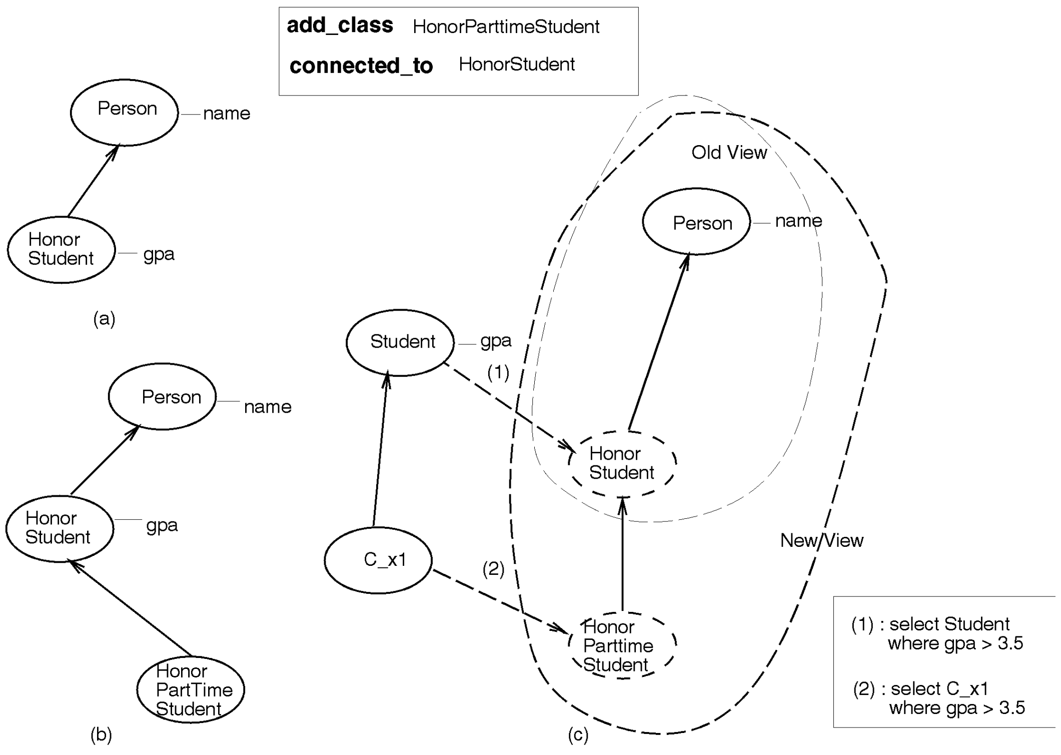


Fig. 13. Schema change for adding a class.

already provided by a command of the view specification language, **removeFromView** *aClass*, of MultiView [25].

6.10 Implementing Rename Operators in TSE

In our system, each view and global schema have their own name space. Furthermore, each view keeps a mapping between the names in its name space and names in the global schema name space. Via this mapping, views and the global schema can interoperate with each other without duplicating objects with different names. The mapping could be implemented as a mapping table that associates names in a view to names in the global schema. By default, the same name in the global schema space is used in the view unless specified.

The schema change operator defined by "**rename_attribute** *x* **to** *y* **in** *C*" renames the attribute *x* in the class *C* to it *y*. This operation is propagated to the subclasses of *C*. If the attribute *x* was overridden with a same named attribute in the class *C*, then the suppressed attribute is restored and propagated to the subclasses. The operand attribute *x* might be either locally defined or inherited. This operation can be achieved by modifying the association for the attribute *x* in the mapping of the view to *y*.

The schema change operator defined by "**rename_class** <class name> **to** <string>" changes the name of a class while preserving the object identity representing the class. This operation can be achieved by modifying the association for the <class name> in the mapping of a view.

7 The Implementation of the TSE System

We have implemented the TSE system to demonstrate the feasibility of our approach, as described below.

7.1 Object Model Requirements for Capacity-Augmenting Views

As mentioned in Section 4, MultiView has to be extended for TSE such that it can support *capacity-augmenting* views. In turn, this extension imposes the following three requirements on the object model (as explained in an example below):

1) *multiple classification*,
2) *dynamic reclassification*, and
3) *flexible* re*structur*ing.

In Fig. 14, two views VS1 and VS2 are defined such that VS1 consists of *Person*, *Student*, and *UnderGrad* classes, and VS2 of *Student*, *Grad*, and *TA* classes. Both view schemata share the *Student* class. Suppose that the attribute *credit* is added to the *Student* class in VS1 and the attribute *register* to the *Student* class in VS2. Following our TSE approach, new virtual classes *Student'* and *UnderGrad'* are created for VS1 by refining *Student* and *UnderGrad*, respectively, with the stored attribute *credit*. The new virtual classes and the *Person* class are selected to form a new version of view schema VS1'. For the view schema VS2, *Student*'', *Grad*', and *TA'* are created and each refines *Student*, *Gad*, and *TA*, respectively, with the attribute *register*.

Considering object instances of *Student*, they now should be able to carry the information associated with both attributes *credit* and *register*. In other words, they are *dynamically reclassified* to become instances of both *Student'* and *Student"* classes. This situation of an object being classified as instance of more than one class even though the classes are not super/subclass of each other is commonly called *multiple classification*. To the best of our knowledge, current OODB systems do not support multiple classification. The only

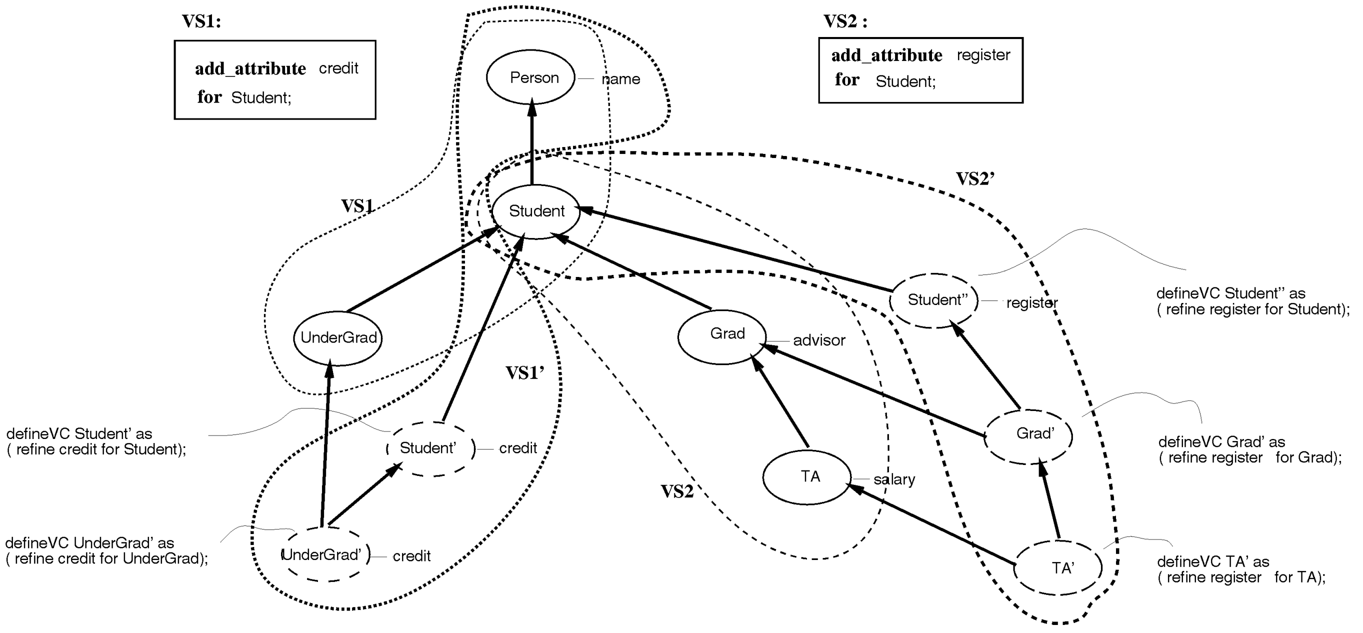

VS1:
add_attribute credit for Student;
VS2 :
add_attribute register for Student;
Person
name
Student
VS1
VS2'
UnderGrad
Grad
advisor
Student''
register
defineVC Student'' as (refine register for Student);
VS1'
defineVC Student' as (refine credit for Student);
Student'
credit
Grad'
defineVC Grad' as (refine register for Grad);
TA
salary
VS2
defineVC UnderGrad' as (refine credit for UnderGrad);
UnderGrad'
credit
TA'
defineVC TA' as (refine register for TA);

Fig. 14. Example for multiple classification.

exception is IRIS [8] 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 [1]. This clearly is no longer acceptable as foundation for our object-oriented view and schema change tools—as demonstrated by the example above.

In the above example, when creating the new *refine* virtual class *Student'*, each object of the *Student* class has to be physically restructured to accommodate storage for the new attribute *credit*. Thus, the underlying system must also support the *flexible restructuring* of object instances to cope with the creation of the virtual classes refining new stored attributes. In this section, we have identified three important requirements for the MultiView extension with capacity-augmentation capabilities, which are *multiple classification*, *dynamic reclassification*, and *flexible restructuring of the object representation* (more details found in [22]). Below, we present the object model paradigm that we have developed to address these requirements.[18]

7.2 Solution: The Object-Slicing Paradigm

We have selected the object-slicing paradigm as solution [17], [11]. Assume that given the schema in Fig. 15a, we want to create a new car object *o*1 that is both of type *Jeep* and of type *Imported*. We cannot find a class in which to store *o*1, without violating the invariant that an object belongs to exactly one class. To resolve this dilemma, one could either create a new intersection class *Jeep&Imported* which is subclass of both *Jeep* and *Imported* classes. It then would create *o*1 as member of the new class (Fig. 15b). Instead, the object-slicing approach (Fig. 15c) would implement multiple classification by creating three objects to represent the *o*1 object, each of which carries data and behavior specific to its corresponding class. As we can see, the *o*1 object corresponds to a hierarchy consisting of the $o1_{Car}$, $o1_{Jeep}$, and $o1_{Imported}$ objects. When the current class of the *o*1 object is Jeep, the $o1_{Jeep}$ object represents the *o*1 object. We call the *o*1 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 most specific class.

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 [11]. In the worst case, the number of intersection classes could grow exponentially with respect to the number of user-defined classes. Also dynamic reclassification may require the creation and/or removal of intersection-

18. 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.

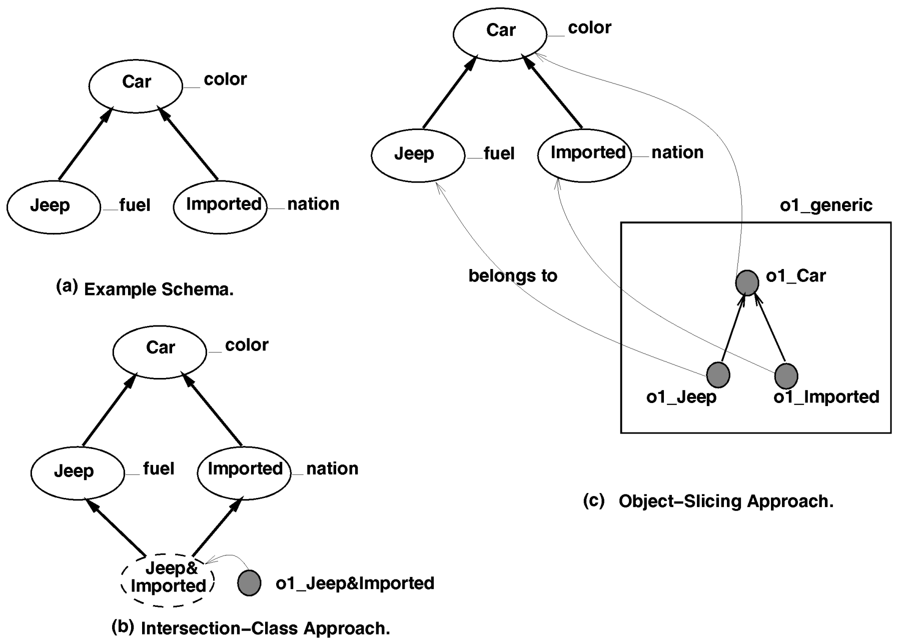


Fig. 15. Two approaches for implementing multiple classification.

classes on the fly. Note that our TSE system is independent of this choice of implementation architecture as long as the underlying object model supports the required properties of multiple classification, dynamic reclassification, and dynamic restructuring.

7.3 Implementing the Object-Slicing Approach Using GemStone

7.3.1 The Basic Representation

Fig. 16 describes the implementation approach of realizing this object-slicing model on top of GemStone.[19,20] Every user-defined class in our system is created as a direct GemStone subclass of `MVClass`, a metaclass providing TSE functionalities. One of the main functionalities provided by the `MVClass` is the message forwarding mechanism, which enables the development of an inheritance mechanism for the object-slicing approach [21].

Every user-defined class *C* inherits the class variables of `supers` and `subs` from `MVClass`, which keep track of the direct superclasses and subclasses of *C* within our object-slicing model, respectively (e.g., between *Imported* and *Car* in Fig. 16). We keep track of subclasses for three reasons:

1) the *global extent* of a class C can be collected efficiently,
2) the class hierarchy can be restructured efficiently for virtual-class integration, and
3) method polymorphism is supported.

Let us explain the implementation approach using an example. Object O1 in Fig. 16 consists of a conceptual object and four implementation objects of type: *Car*, *Jeep*, *Imported*, and *Hyundai*, respectively. The implementation object of the *Car* class carries the data for the *color* attribute, etc. Each implementation object is implemented as an instance of a user-defined class, which is subclass of `MVClass`. Each carries a system-defined attribute `concepLink` pointing to its conceptual object O1 as well as class-specific data.

The conceptual object *O*1 does not carry any state but rather holds the references to its implementation objects (`implLink`) and their respective classes[21] (depicted as a table inside the conceptual object in Fig. 16). So, each implementation object can be referenced by its class through the dictionary associated with its conceptual object. For example, the implementation object of the *Imported* class for O1 can be found by sending the message "`implLink`: *Imported*" to the conceptual object of O1. In summary, an object is represented in our system by a hierarchy of implementation objects linked to each other via a conceptual object.

7.3.2 Reclassification and Restructuring for Object Instances

We can achieve *dynamic reclassification* and *dynamic restructuring* with the help of the two methods: `getType:` and `loseType:`. Suppose an object O1 is created as instance of *Hyundai* by the command "*Hyundai* **create**" (Fig.17a). The object O1 can also be made an instance of *Jeep* by sending the message "`getType`: *Jeep*" to the object O1. This changes the representation of O1 depicted in Fig. 17a into that of Fig. 17b. The method "`getType`: <*class*>" also creates the implementation objects of classes lying in the path from <*class*> to *Root* if they do not already exist. In this example, the implementation object $O1_{Car}$ already exists. Thus, the creation of the implementation objects terminates at the *Car* class. New implementation objects are connected to

19. More specifically, we use GemStone version 3.2, using the Opal interface. GemStone is a registered trademark of GemStone Inc.
20. In the figure, the thick dashed line arrows represent is-a relationships of GemStone, the thick solid arrows depict the is-a relationships of our system and the thin dotted lines link the conceptual object with its implementation objects.

21. This dictionary is a set of the associations indexed by class names. Thus, associations can easily be added to or removed from the dictionary.

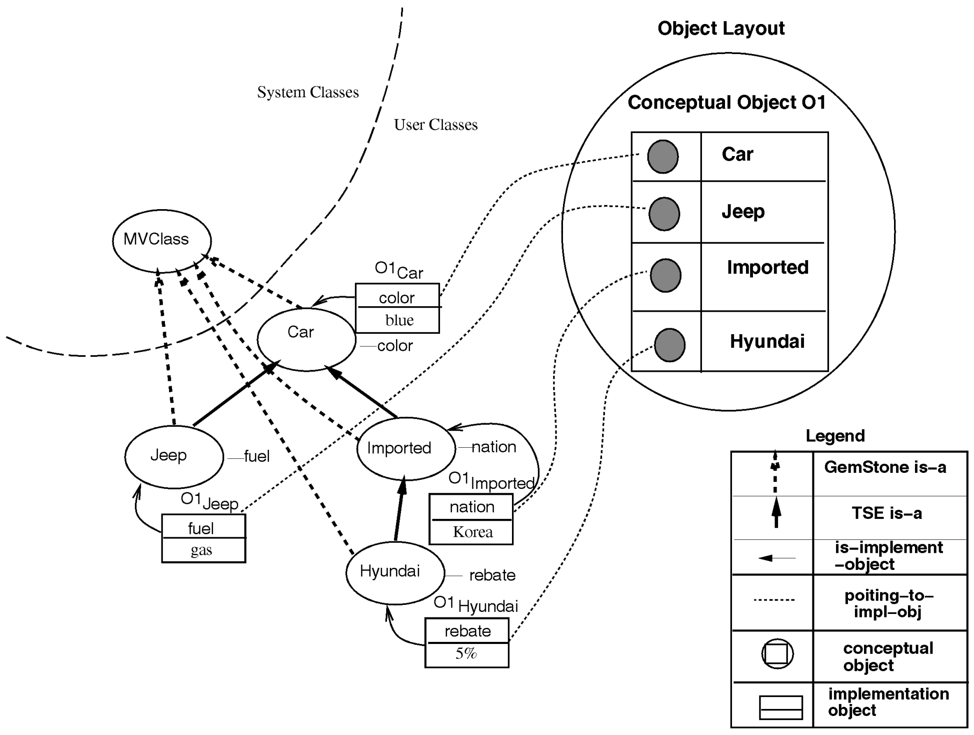


Fig. 16. Implementing the object-slicing approach using GemStone.

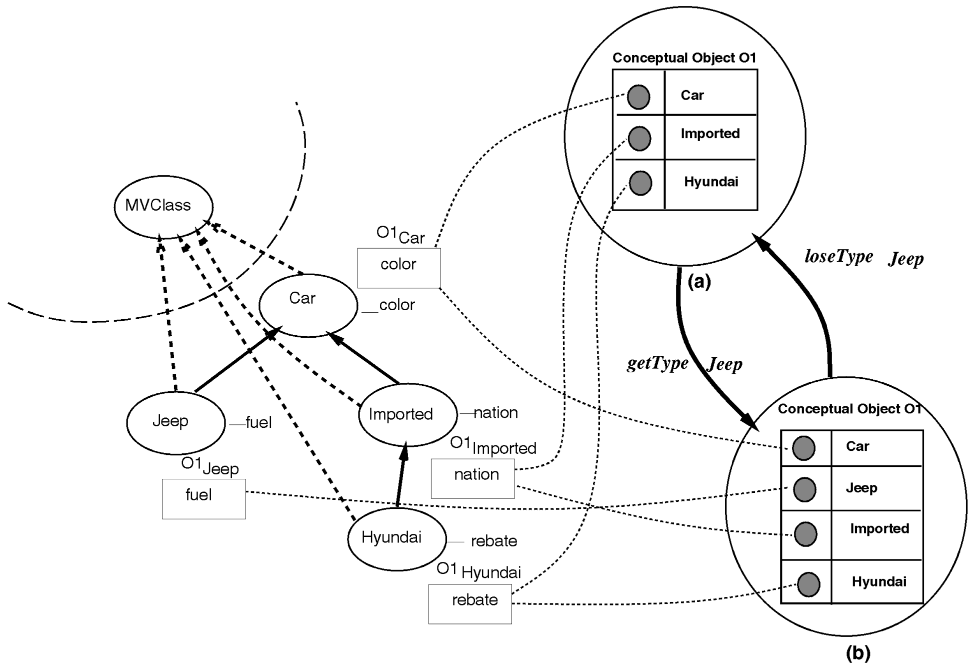


Fig. 17. Getting and losing a type.

the conceptual object of O1 (see the thin dashed lines in Fig. 17).

The method “**loseType**: <*class*>“ destroys the implementation objects of <*class*> and its subclasses. For example, the conceptual object O1 is changed from Fig. 17b to Fig. 17a by sending the message “**loseType**: *Jeep*” to O1. It now can easily be seen that we can achieve *dynamic classification* by combining **getType** and **loseType** methods. For example, the O1 object of the class *Hyundai* can be reclassified as instance of *Jeep* by executing the two methods “**getType**: *Jeep*” and “**loseType**: *Hyundai*.”

7.3.3 Restructuring Due to Virtual-Class Insertion

The dynamic restructuring capability due to the unique object-slicing architecture enables us to easily insert a *capacity-augmenting* virtual class into the global schema hierarchy. Suppose we create a virtual-class *Jeep*’ that refines a new attribute *horsepower* for the *Jeep* class in Fig. 17. Then, by the definition of the *refine* operator, the extent of the *Jeep*’ class should be the same as that of the *Jeep* class. This is achieved by making every instance of the *Jeep* class get the additional type of the *Jeep*' virtual class as done in the following procedure:

for each instance (*inst*) of *Jeep* {*inst* **getType**: *Jeep*'}.

The execution of the above procedure results in the creation of implementation objects of *Jeep*' that carry the state information for the new attribute *horsepower*.

7.4 The TSE System Architecture

Fig. 18 describes the overall architecture of the TSE system.[22] As shown in the figure, we have built the **MultiView object model** on top of GemStone to support the necessary features of multiple inheritance, multiple classification, and dynamic restructuring. The **Global Schema Manager** is directly constructed using the **MultiView model**, providing a layer between the database and all other system modules. Each module is briefly explained below:

- The **Transparent Schema-Evolution Manager** takes a user input (a schema change) and calls appropriate modules to complete the requested schema change (using algorithms defined in Section 6).
- The **TSE Translator** translates a requested schema change into a set of object algebra statements that define all necessary virtual classes (Section 5.1).
- The **Extended Algebra Processor** takes a set of algebra statements as input and creates appropriate virtual classes (also see Section 5.1 and [12]). This module is an extended version of the **MultiView Algebra Processor** supporting object-restructuring when an algebra operator (e.g., *refine-attribute*) requires additional state for the input objects.
- The **MultiView Classifier** integrates the newly created virtual classes into the consistent global schema hierarchy, allowing for upwards inheritance for both base and virtual classes (see [27]).
- The **MultiView View Manager** takes a set of classes as an input and generates a consistent view schema including the appropriate generalization hierarchy (further described in [26]).

8 Related Research

8.1 Comparison Criteria

The continued support of old programs when performing schema evolution has been recognized as a key issue in the literature [32], [10], [9], [18]. This section compares proposals addressing this issue using the following set of criteria for comparison (see Table 1):

- **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.
- **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 [32]. Others require the user to keep track of class versions for each valid schema [9].
- **flexibility** to build a new schema from class versions: It refers to the capability of composing various schemas by combining class versions.
- **subschema evolution**: It refers to the capability to confine the effect of schema evolution to a subschema 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, it allows a user to specify the necessary schema change directly through a customized user view. Second, the content-based derivation power of views could be exploited to support more complex schema customizations as possible by pure schema evolution (See Section 3.2).

22. In Fig. 18, the shaded modules are specific to the TSE system, and are not available as subsystems of MultiView.

One important issue not addressed by other systems is 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 in TSE by specifying the schema change directly on a view rather than on the global schema.

8.2 Versioning and Other Approaches for Schema Evolution

Skarra and Zdonik approach towards type changes in the Encore System [32] 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. 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. The user thus must 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 [10] 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. This approach does however not 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

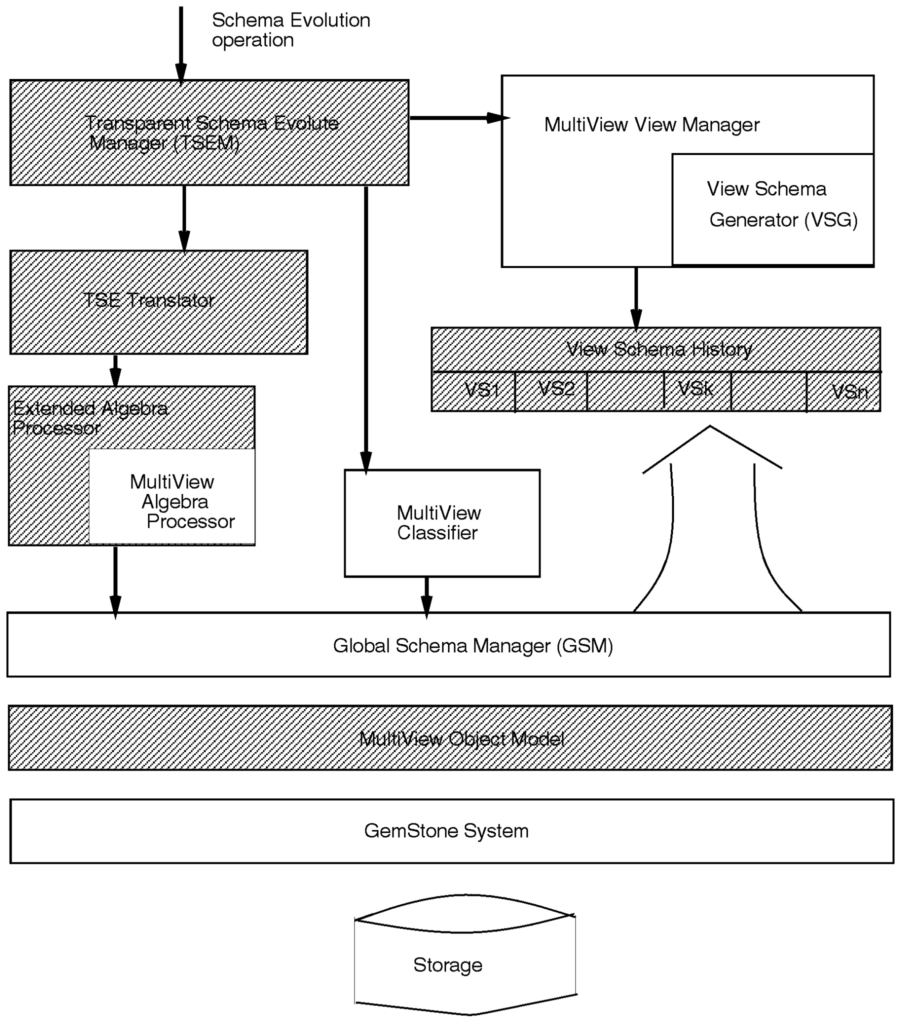

Schema Evolution operation
Transparent Schema Evolute Manager (TSEM)
MultiView View Manager
View Schema Generator (VSG)
TSE Translator
View Schema History
VS1
VS2
VSk
VSn
Extended Algebra Processor
MultiView Algebra Processor
MultiView Classifier
Global Schema Manager (GSM)
MultiView Object Model
GemStone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

Fig. 18. The system architecture for transparent schema evolution.

TABLE 1
COMPARISON OF RELATED WORK

	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
Clamen's	yes	keep track of class versions for each schema	yes	no	no
Bertino's	yes	keep track of class versions for each schema	yes	no	yes
Bratsberg's	yes	keep track of class versions for each schema	yes	no	yes
TSE	yes	nothing particular	no	yes	yes

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. [9] 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.

Mehta et al. propose a unique approach that enables the different schema versions to share the objects in the *Rose* system [18]. Instead of exception handlers for type mismatches between the object and the schema, an *intelligent interface* and *object manager* are used to resolve mismatches. The interface compares the schema of the persistent object and the schema defined by an application program and maps the persistent object into main memory. An object manager keeps track of necessary information to be used by the interface to resolve type mismatches. This approach is powerful for resolving type mismatches, but the performance of the interface may become the bottleneck of an application which requires lots of secondary storage access.

The class versioning approach CLOSQL, proposed by Monk and Sommerville [19], 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, and extensions for handling new stored attributes appear not to have been dealt with.

Clamen [7] also presents an alternative solution for the transparency issue. His work is a generalization of Skarra and Zdonik's type change management [32] described above. In Clamen's scheme, each instance is represented as a disjoint union of the representation of each version, while in Skarra and Zdonik's, each instance is represented as an *interface type*, which is a minimal cover of every instance version. This generalization of the instance representation allows for more flexible instance adaption between instance versions. For example, attributes of an instance of a new version may be shared with, derived from, or dependent on attributes of a previous version of the instance. However, this work is restricted to individual classes rather than hierarchically interrelated classes.

8.3 Schema Evolution Using Views

Tresch and Scholl [33] also advocate views as a suitable mechanism for simulating schema evolution. This article is based on examples, while no precise algorithms are given. They state that schema evolution can be simulated using views if they are not capacity-augmenting. In this paper, we now take the opposite stand of exploring what extensions to view technology must be done to support such described schema changes.

Bertino [4] also presents a view mechanism and indicates that it can be utilized to simulate schema evolution. Like our TSE system, the proposed mechanism is *capacity-augmenting* in that new stored attributes can be added to a view. However, the paper focuses on the evolution of individual classes rather than the schema. Also, implementation issues such as how to deal with the requirements of *multiple classification* and *flexible restructuring* are not discussed.

Bratsberg [5] presents a design of a schema-evolution system also based on object-oriented views. In his system, existing and new applications coexist and share the same set of objects. However, this work is directed toward class evolution rather than schema evolution. In other words, in this approach, the unit of change is a class, while in our TSE system, it is a view schema. In contrast to TSE, his work is specific to an object-oriented data model that separates type and extent hierarchies, and violates the *full inheritance* invariant.

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, we present algorithms to compute the new view reflecting the semantics of the desired schema change 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 [25] is chosen because it generates updatable views and complete view schemas rather than individual view classes. 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—a feature not provided by current OODB systems. We have successfully addressed this problem using an object-slicing approach. The extended MultiView object model has been implemented on top of Gemstone, providing the necessary features of multiple classification and dynamic data restructuring to the TSE system [21].

We have demonstrated the TSE approach on a comprehensive set of schema-evolution operators, i.e., those typically supported by OODBs [32], [10], [9], [18], [19], [2], [15], [34], [20]. 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. This is important since it assures that the resulting view schemas are updatable, because views generated by an object-preserving algebra have been shown to be updatable [31].

Some more complex schema-evolution operators, such as transforming values to an object, partitioning a class, and coalescing classes, can't easily be simulated by an object-preserving algebra. Since they may require an object-generating query language, the updates on views will have to be defined carefully to ensure that the virtual classes generated to simulate schema evolution behave properly as base classes. Thus, this issue of updatability of such object-generating views represents a challenging open problem. Another extension of TSE we are considering concerns the treatment of a more powerful data model, e.g., *has-part* aggregation and others such as semantic relationships, or even general constraints [29]. While we are currently developing algorithms for incremental maintenance of materialized views in MultiView [13], [14], we also want to develop specialized 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 chains of dependent classes.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation RIA Grant No. IRI-9309076, NSF NYI Grant No. IRI-94-57609, and the University of Michigan Faculty Award Program. We also thank our industrial sponsors, including Intel, AT&T, and IBM.

REFERENCES

[1] S. Abiteboul and A. Bonner, "Objects and Views," *SIGMOD*, pp. 238–247, 1991.
[2] J. Andany, M. Leonard, and C. Palisser, "Management of Schema Evolution in Databases," *VLDB*, pp. 161–170, Sept. 1991.
[3] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," *SIGMOD*, pp. 311–322, 1987.
[4] E. Bertino, "A View Mechanism for Object-Oriented Databases," *Proc. Third Int'l Conf. Extending Database Technology*, pp. 136–151, Mar. 1992.
[5] S.E. Bratsberg, "Unified Class Evolution by Object-Oriented Views," *Proc. 12th Int'l Conf. Entity-Relationship Approach*, pp. 423–439, 1992.
[6] P. Butterworth, A. Otis, and J. Stein, "The GemStone Object Database Management Systemversion," *Comm. ACM*, vol. 34, no. 10, pp. 64–77, Oct. 1991.
[7] S.M. Clamen, "Type Evolution and Instance Adaptation," Technical Report CMU-CS-92-133R, School of Computer Science, Carnegie Mellon Univ., 1992.
[8] D. Fishman, "Iris: An Object Oriented Database Management System," *ACM Trans. Office Information Systems*, vol. 5, pp. 48–69, Jan. 1987.
[9] H.J. Kim, "Issues in Object-Oriented Database Systems," PhD thesis, Univ. of Texas at Austin, May 1988.
[10] W. Kim and H. Chou, "Versions of Schema for OODBs," *Proc. 14th VLDB*, pp. 148–159, 1988.
[11] H.A. Kuno, Y.G. Ra, and E.A. Rundensteiner, "The Object-Slicing Technique: A Flexible Object Representation and its Evaluation," Technical Report CSE-TR-241-95, Univ. of Michigan, 1995.
[12] H.A. Kuno and E.A. Rundensteiner, "Developing an Object-Oriented View Management System," *Proc. Center for Advanced Studies Conf.*, pp. 548–562, Oct. 1993.
[13] H.A. Kuno and E.A. Rundensteiner, "Materialized Object-Oriented Views in MultiView," *ACM Research Issues Data Eng. Workshop*, pp. 78–85, Mar. 1995.
[14] H.A. Kuno and E.A. Rundensteiner, "Incremental Update Propagation Algorithms for Materialized Object-Oriented Views in MultiView," to appear in *Proc. ICDE '96*, 1996.
[15] B.S. Lerner and A.N. Habermann, "Beyond Schema Evolution to Database Reorganization," *Proc. OOPSLA*, pp. 67–76, 1990.
[16] S. Marche, "Measuring the Stability of Data Models," *European J. Information Systems*, vol. 2, no. 1, pp. 37–47, 1993.
[17] J. Martin and J. Odell, *Object-Oriented Analysis and Design.* Prentice Hall, 1992.
[18] A. Mehta, D.L. Spooner, and M. Hardwick, "Resolution of Type Mismatches in an Engineering Persistent Object System," technical report, Computer Science Dept., Rensselaer Polytechnic Inst., 1993.
[19] S. Monk and I. Sommerville, "Schema Evolution in OODBs Using Class Versioning," *SIGMOD Record*, vol. 22, no. 3, Sept. 1993.
[20] D.J. Penney and J. Stein, "Class Modification in the GemStone Object-Oriented DBMs," *Proc. OOPSLA*, pp. 111–117, 1987.
[21] Y.-G. Ra, H.A. Kuno, and E.A. Rundensteiner, "A Flexible Object-Oriented Database Model and Implementation for Capacity-Augmenting Views," Technical Report CSE-TR-215-94, Univ. of Michigan, 1994.
[22] Y.-G. Ra and E.A. Rundensteiner, "OODB Support for Providing Transparent Schema Changes," *Proc. Center for Advanced Studies Conf.*, 1994.
[23] Y.-G. Ra and E.A. Rundensteiner, "A Transparent Object-Oriented Schema Change Approach Using View Schema Evolution," Technical Report CSE-TR-211-94, Univ. of Michigan, 1994.
[24] Y.-G. Ra and E.A. Rundensteiner, "A Transparent Object-Oriented Schema Change Approach Using View Schema Evolution," *IEEE Int'l Conf. Data Eng.*, pp. 165–172, Mar. 1995.
[25] E.A. Rundensteiner, "MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases," *Proc. 18th VLDB Conf.*, pp. 187–198, 1992.
[26] E.A. Rundensteiner, "Tools for View Generation in OODBs," *Proc. Int'l Conf. Information and Knowledge Management*, pp. 635–644, Nov. 1993.
[27] E.A. Rundensteiner, "A Classification Algorithm for Supporting Object-Oriented Views," *Proc. Int'l Conf. Information and Knowledge Management*, pp. 18–25, Nov. 1994.
[28] E.A. Rundensteiner and L. Bic, "Set Operations in New Generation Data Models," *IEEE Trans. Knowledge and Data Eng.*, vol. 4, pp. 382–398, Aug. 1992.
[29] E.A. Rundensteiner, L. Bic, J. Gilbert, and M. Yin, "Set-Restricted Semantic Groupings," *IEEE Trans. Data and Knowledge Eng.*, vol. 6, no. 2, pp. 193–204, Apr. 1994.
[30] C. Santos, S. Abiteboul, and C. Delobel, "Virtual Schemas and Bases," *Proc. Int'l Conf. Extending Database Technology* (*EDBT*), 1994.
[31] M.H. Scholl, C. Laasch, and M. Tresch, "Updatable Views in Object-Oriented Databases," *Proc. Second DOOD Conf.*, Dec. 1991.
[32] A.H. Skarra and S.B. Zdonik, "The Management of Changing Types in Object-Oriented Databases," *Proc. First OOPSLA Conf.*, pp. 483–494, 1986.
[33] M. Tresch and M.H. Scholl, "Schema Transformation without Database Reorganization," *SIGMOD Record*, pp. 21–27, 1993.
[34] R. Zicari, "A Framework for O_2 Schema Updates," *Building an Object-Oriented Database System: The Story of O_2*, F. Bancilhon, C. Delobel, and P. Kanellakis, eds., Morgan Kaufmann, 1992.

Young-Gook Ra received a BS degree in electrical engineering from Seoul National University in 1987 and an MS degree in computer engineering from Pennsylvania State University. He is now a doctoral student at the University of Michigan. Ra is interested in databases and their applications. His major interests are object-oriented databases, especially scheme evolution and view constructing tools. He has applied object-oriented view technology to create a conventional schema evolution transparency from other views or exiting application programs. He is a member of the IEEE Computer Society.

Elke A. Rundensteiner received a BS degree (Vordiplom) from Johann Wolfgang Goethe University, Frankfurt, Germany, in 1984; a master's degree from Florida State University, Tallahassee, in 1987; and her PhD degree from the University of California at Irvine in 1992—all in computer science. Her PhD dissertation was on the development of object-oriented view technology and its application to design-tool integration. Dr. Rundensteiner has received numerous honors and awards, including the Fulbright Scholarship, the Young Investigator Award in databases (1994) from the National Science Foundation, and the Intel Young Investigator Engineering Award from the Engineering Foundation.

Dr. Rundensteiner is currently with the Department of Computer Science at Worcester Polytechnic Institute in Massachusetts. Her main interests lie in the application and development of object-oriented database technology for nontraditional applications—such as computer-aided design, manufacturing, and scientific applications. She is leading a project aimed at developing object-oriented extensible view support systems for tool integration and for the on-line evolution of databases. She previously served as an assistant professor in the Department of Electrical Engineering and Computer Science at the University of Michigan. As an investigator on the University of Michigan Digital Library project, Dr. Rundensteiner developed database tools for agent-based digital library systems—including registry database systems and integration services of heterogeneous multimedia library systems. Her research interests also include active and fuzzy database systems for workflow management and control in manufacturing applications, multimedia databases, and spatio-temporal data models. She is a member of the IEEE and the ACM.