BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-89-M10

"Implementing Views in the ENCORE Object-Oriented Database System"

by
Alan N. Ewald

Implementing Views in the ENCORE Object-Oriented Database System

Thesis

Alan N. Ewald
Department of Computer Science
Brown University
December 13, 1989

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University.

Stanley B. Zdonik

Professor Stanley B. Zdonik
Advisor

Implementing Views in the ENCORE Object-Oriented Database System

Alan N. Ewald
Department of Computer Science
Brown University
Providence, RI 02916

December 13, 1989

Abstract

Views are an important feature of database management systems. In this paper, we describe an implementation of views based on the ENCORE data model supplemented by two extensions. The first extension allows for the definition of multiple interfaces for a set of objects in the database. The second extension is a mechanism for collecting objects into sets based on queries formed using ENCORE's query algebra. The extents of these sets are dynamically maintained to always reflect the result of the query used to define the set. We use a combination of function application and procedural attachment to implement the maintenance of these sets. Our extended ENCORE model includes the necessary features to define flexible database views which provide a context in which a user can access an ENCORE database. An important goal of our definition is the maintenance of strong typing of the objects visible within a view.

1 Introduction

Views are an important feature of traditional database management systems. They are used to provide data independence from the physical organization of the database or from changes to the database schema. Views give the user a picture of the database which has been tailored and simplified to meet their needs. Many different views of the same database may exist for different groups of users. Views can also be used to implement data security by hiding portions of the database from the external user. The term view can mean either a picture of a particular database entity or a picture of the overall database. In this paper, the term *view* is used to mean a picture of the overall database. The phrase *object-level view* is used to mean a picture of an individual entity.

In the relational model, views are built from existing relations by means of a query [Ull88]. The result of a query over a set of relations is a new relation built using a combination of project, select, and join operators. The select and project operators describe a new relation which is a row or column subset (respectively) of another relation while the join operator builds a new relation by combining tuples from two existing relations. A view is therefore a relation defined by designating a query as its constructor. The extent of a view is not usually stored in the database to avoid the costs of maintenance in the face of changes to base relations. When an access is made through a view, the query which defines the view is composed with the query being made against the view to produce an equivalent query. In this way, a view relation is always consistent with the base relation or relations it was created from.

Updating a relational view is not as straight-forward as reading one. There are views which are not updateable [Dat88]. Examples include views which do not contain the primary key of an underlying relation, (some) views created with a join operator, and views created with queries involving an aggregate operator (e.g., Sum, Avg). In these cases, a view update may dictate an ambiguous update of the underlying database or may introduce null values into a base relation. View updates are restricted in many relational database systems to those view relations formed using a project and/or select operation which retains the key attribute(s) of the base relation. In order to overcome these restrictions, [Kel86] proposes that the ambiguity of many view updates can be resolved by acquiring information about view semantics from the view definer.

Views are useful to include in object-oriented database systems (OODBs) for many of the same reasons that they are useful in other database systems. Existing object-oriented database systems [MS87, LRV88, ESZ89, AH87, BCG+87, MD86, FBHPC+87] are based upon the concepts of object identity, complex state, abstract types, and inheritance. Since data abstraction is an integral part of object-oriented database systems, views are not necessary to provide independence from physical organization. In fact, data abstraction provides a built-in viewing mechanism in that it limits access of an object to those operations provided by the object's abstract type. An OODB viewing mechanism should allow the definition of new abstract types and objects based on existing abstract types and their instances. Views can be used in addition to data abstraction to simplify the interface of an existing abstract type for a particular group of users. Views can also provide a level of data security and may be useful in addressing problems related to schema modification.

Defining views in an object-oriented database is different from defining views in a relational database. Unlike tuples, objects in an OODB have a behavioral aspect in addition to their state, although behavior is strongly related to state. In a relational database, the inclusion of a tuple in a view is based purely on attribute values within the tuple. There is no notion of specifying the operations that may be performed on a tuple in a relational view [1]. A view model in an OODB must address both behavior and state. Objects to include in a view should be selectable based on their behavior as well as their current state. Furthermore, the behavior of an object within a particular view should also be selectable. The ability to accomplish these goals is dependent on the query mechanism provided by the OODB.

Unlike the relational model, object-oriented database systems support object identity [KC86]. In other words, each object in the database has an identity that does not change over time. This is usually implemented as an immutable object identifier that can always be used to reference an object. Object identity transcends all views of a database. The same object may be viewed in several different ways while maintaining its identity. Updates made to an object in the context of one view are visible in the context of other views which include the object. New objects may be created for the purposes of a view (e.g., as the result of a join operator). In this case, object identity provides flexibility over the relational model in handling view updates with ambiguous interpretations. The identity of an object or objects used to create a new object for the purposes of a view can be stored with that object [RS79, CM84, HZ90]. This allows an update of a view object to explicitly operate on the object(s) that was used to create it. The ability of an object to reference other objects also has implications for data security as provided by a view. Objects that are meant to be protected may be accessible by evaluating references from non-protected objects.

Once a view is defined, it should form a complete context in which a user can access a database. The type of all objects visible within a view should be visible. Similarly, if an operation is visible, the types of its arguments should also be visible (defined as *type-closure* in [HZ90]). Normal database operations such as querying and operation invocation should be available. In short, database access within the context of a view should be indistinguishable from database access in the context of the underlying database schema.

[1] The Rigel system is an exception, see section 6 for details

In this paper, we present a view model and implementation for the ENCORE [ESZ89] database. In ENCORE, objects are strongly typed. All objects that are instances of a type are also members of a collection (class) associated with that type. In order to support database views, we have extended the ENCORE data model with two features that provide object-level views. First we define a mechanism for providing multiple interfaces for a set of objects. This is accomplished by defining new abstract types as possible replacements for an existing abstract type. Each of these new types is a potential interface for the objects which are instances of the existing type. Only one type can be specified for an object within a particular database view. This restriction maintains the strong typing of objects. The second feature allows the definition of new collections of objects by executing queries over existing collections. The resulting collections are dynamically maintained with regards to the query used to create them. Additional behavior may be defined for objects which are members of a collection. A database view is defined by specifying a set of types and object collections which are available to a user of the view. The types included in a view definition specify the behavior of all visible objects. The object collections included in a view definition specify the sets of objects that can be queried. The enumerated collections do not include all objects that are visible to the users of a view since references (possibly transitive) to other objects may be present in the objects which are members of a visible collection.

The next section briefly describes the ENCORE data model and query algebra. Sections 3 and 4 discuss our extensions to the ENCORE model. A definition of database views is given in section 5. A comparison to other work is included in section 6 and a summary discussion concludes the paper.

2 ENCORE

ENCORE is an object-oriented database system which supports abstract types, type inheritance, object identity and collections (classes) of objects. ENCORE includes a basic set of types (e.g., Integer, String, etc.) which may be used in constructing new abstract types. Additionally, ENCORE includes the parameterized types Set[T] and Tuple[$< (A_1, T_1), \ldots, (A_n, T_n) >$] which may be used to *generate* new types and objects. Sets are strongly typed collections of objects which are manipulated using pre-defined operations. Tuples consist of pairs of typed ($T_1, \ldots, T_n$) attributes ($A_1, \ldots, A_n$) and values. Get_Attribute_Value and Set_Attribute_Value operations are provided for each attribute of a type generated from type Tuple.

An abstract type is defined by giving it a name, a set of supertypes, a set of properties, and a set of operations. Together these features describe the interface and implementation of the type being defined. Types are objects which are instances of type Type. Properties are typed objects that make up the state of the objects with whose type they are associated. Properties may be stored or computed with no visible difference to external users of a type. Dot notation is used to denote property access within a query. Since properties are typed, they may themselves have properties and operations. Every property includes Get_Property_Value and Set_Property_Value operations.

The set of operations defined for a type specifies the behavior of instances of that type. At a minimum, the set of operations includes Get_Property which provides access to the property objects defined for a type. Operations are typed objects as well. The operations and properties defined for a type are divided into public, internal, and private interfaces. Features of the public interface are available to external users of the type (e.g. other types, database users), features of the internal interface are available to subtypes and features of the private interface are only available to the type itself.

The supertypes of a type contribute operations and properties to the type's definition. A subtype may override (overload) inherited operations and properties and/or add new operations and properties as long as the substitutability of instances of the subtype for instances of a supertype is maintained. In particular, the domain of an overridden property may not be changed. This is a consequence of the Get_Property_Value operation which is included with every property. If a supertype accesses a property on an instance of a subtype, the value returned by invoking Get_Property_Value on the property may be outside the domain of the property as defined on the supertype. In the case of operations, if the first argument (e.g., self) is ignored, then the signature of an operation may be changed according to the rules of contravariance [Car88] which specify that the domain of an operation may increase while the range may decrease. In summary, a subtype may not constrain the behavior inherited from its supertypes. The subtype relationship between types organize the types into a lattice.

An object is created as an instance of a single type and has a unique identity which may be used to reference the object. The physical implementation of an object includes a reference to its type object as well as storage for references to the instances of the property type objects which correspond to the properties defined by the object's type. The instances of a type t are collected into a set whose type is Set[t]. This set is automatically maintained by the system. Due to ENCORE's subtyping rules (e.g., substitutability), all objects are considered instances of type Object as well as every other type on a path from their declared type to type Object. Consequently, an object is made a member of the instance set of every type on the path.

The abstract types defined for an ENCORE database form a *base schema*. The base schema combined with each type's instance set forms a *base view* of the database. This is further elaborated in section 5.

2.1 Query Algebra

ENCORE's query algebra is described in [SZ89]. The algebra can be divided into operators which produce collections of objects and operators which modify the returned collections. The query algebra supports abstract types in that only features of the interface provided by an abstract type may be used when forming a query. The objects in a collection returned by a query are strongly typed and consequently the collection is strongly typed. This behavior is accomplished using the parameterized types Set and Tuple. Object identity is preserved for existing objects returned by a query. New objects that are created as the result of a query have a unique identity.

The query operators which produce collections of objects are UNION, INTERSECTION, DIFFERENCE, SELECT, IMAGE, PROJECT, and OJOIN. The first three operators have the expected semantics for manipulating sets of objects. SELECT produces a subset of a collection of objects where members of the subset satisfy a predicate. IMAGE returns a collection of objects obtained by executing a function on each member of another collection. PROJECT is a generalization of IMAGE which produces a collection of tuple objects where the attribute values of each tuple are the result of executing functions on a member of the collection queried over. OJOIN also produces tuples whose attribute values are pairs of objects from the collections queried over where each pair is related by a specified predicate. PROJECT and OJOIN (and possibly IMAGE) create new objects as a result of their execution. New types which are instantiations of type Set and/or Tuple are also created by these operators.

The FLATTEN, DUP_ELIMINATE, NEST, COALESCE, and UNNEST operators are used to restructure a collection of objects returned as the result of a query. FLATTEN takes a collection of set objects and returns a collection of the unique objects that are members of the set objects in the original collection. DUP_ELIMINATE deletes duplicate objects from the result of a query. NEST provides a way to group

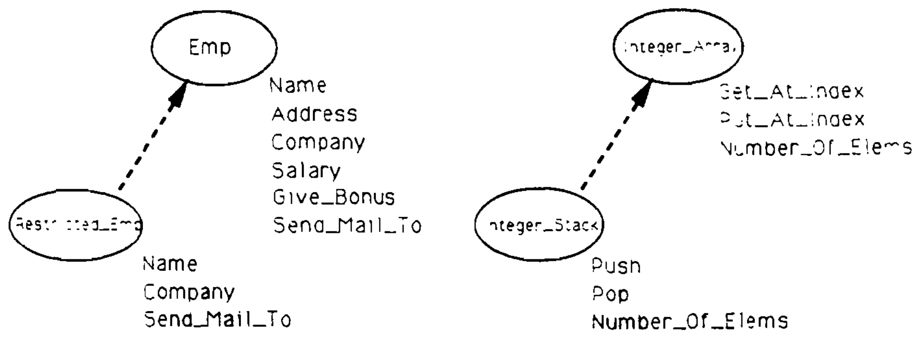

Figure 1: Sample surrogate-types

tuples which share attribute values except for one attribute whose values are collected into a set to form the attribute's value for the nested tuple. COALESCE can be executed on the tuples resulting from a NEST to ensure that the sets created for the attribute NESTed upon are not duplicated. UNNEST provides the opposite functionality to NEST in that it modifies tuples that have a set-valued attribute by creating multiple tuples, each with a different member of the set as the value for the (originally set-valued) attribute.

3 Surrogate-Types

This section describes the first of two extensions to the ENCORE data model. A *surrogate-type* is associated with an existing abstract type (the *base type*). A surrogate-type provides a different interface for objects created as instances of a base type. In other words, surrogate-types are object-level views. Consider the sample types shown in figure 1. Type Emp includes properties name, address, company, and salary and operations Give_Bonus and Send_Mail_To. A different interface to Emp instances might include name, company and Send_Mail_To but exclude company, salary and Give_Bonus. We can provide this restricted interface to Emp instances by creating a surrogate-type (Restricted_Emp) for type Emp which hides some of the features of type Emp while retaining others. As an illustration of the usefulness of surrogate-types for data abstraction consider the Integer_Stack/Integer_Array (adapted from [HZ88]) example also shown in figure 1. Integer_Array includes operations such as Put_At_Index, Get_At_Index, and Number_Of_Elems. These operations manipulate properties (i.e., state) which implement an integer array. A surrogate-type of Integer_Array, called Integer_Stack, is defined which provides a stack abstraction implemented in terms of the Integer_Array representation. The operations provided for Integer_Stack include Push, Pop, and Number_Of_Elems. Push and Pop invoke operations copied from type Integer_Array to manipulate the array representation rather than manipulating it directly. Number_Of_Elems in included in both type definitions with the implementation being defined on Integer_Array.

Objects cannot be created as instances of a surrogate-type within the context of the base view of a database. Instead, a surrogate-type may replace its corresponding base type as the interface of choice for instances of the base type within a database view. When such a replacement occurs, all instances of the base type may only be accessed using operations and properties in the interface of the chosen surrogate-type. In the case of the Restricted_Emp surrogate-type, all properties and operations except salary, address and Give_Bonus are accessible when Restricted_Emp replaces Emp within a view. A surrogate-type always includes the initialization information needed to create an instance of the base type when the surrogate-type is active within a view.

Many surrogate-types may be created for a single base type. A base type and its corresponding surrogate-types form a set of interfaces valid for a particular set of objects. Only one member of this interface set may be active for a particular view of the database. The active member in the context of the base schema is the base type. Since only one interface is available for a particular view, it is possible to statically type-check programs written against the view. This is because the type of any object can be uniquely determined during compilation. Explicitly associating multiple types (interfaces) with an object [Shi81, HZ88, FBHPC+87] requires that runtime checking be performed to determine which types are valid for an object before allowing access to that object.

A surrogate-type is initially specified by including the public and internal properties of its base type, and their associated Get_Property_Value and Set_Property_Value operations [2], in the public and internal interface of the surrogate-type. These properties and operations define the state, and constraints on that state, that is available for use on the surrogate-type. The visibility of properties copied from the base type may be modified by moving them to other levels of interface on the surrogate-type [3]. The operations defined on the base type are not *automatically* included in the definition of the surrogate-type. A potential reason for this is that the operations of the base type may reference properties or operations which are private to the base type and therefore not included in the definition of the surrogate-type. The dynamic binding process used by ENCORE addresses this problem by switching context from the surrogate-type to the base type when an operation copied from the base type is invoked [HZ88, MD86]. This is similar to the binding process which occurs when an operation defined on a supertype (and not overloaded on the subtype) is invoked. In fact, the reason for not automatically including base type operations is that semantic correctness is not guaranteed. For example, consider the Emp and Restricted_Emp types. Suppose that Emp includes an operation called Display_Emp which displays the values of each property of an Emp object. If the salary property is hidden on Restricted_Emp by moving it to the private interface, then the inclusion of Display_Emp in the interface of Restricted_Emp is not semantically correct because the value of the salary property may be displayed. Therefore, the inclusion of base type operations is left to the type definer [4].

The next step in defining a surrogate-type is to specify it as a subtype of some type (base *or* surrogate) on the path from the supertype of its base type to the root of the subtype hierarchy (Object). If no supertype is specified, the surrogate-type defaults to being a subtype of type Object. If there is more than one supertype defined for the base type, the surrogate-type may be specified as a subtype of a type on each path to the root of the subtype hierarchy. The supertypes chosen for a surrogate-type provide an additional set of properties and operations to consider. ENCORE does not permit name conflicts between inherited properties and operations and will raise an error condition if a conflict occurs.

Properties of the proposed supertypes are compared to the properties copied from the base type. Properties of the same name and signature are deemed equivalent and are considered to be inherited from a supertype. Properties of the same name and different signatures are in conflict and result in an error condition that must be resolved by the type definer. All properties copied from the base type which are not inherited from a supertype are eligible to be moved to the any interface of the surrogate-type.

Operations defined on the proposed supertypes must also be defined on the surrogate-type. Supertype operations may be inherited or re-implemented as long as the substitutability of the surrogate-type for each supertype is maintained. Additional operations may be added to the surrogate-type. As alluded to

[2] It is assumed that these operations have no side-effects.

[3] Movement of properties may be restricted by the choice of supertypes for the surrogate-type.

[4] As with properties, the inclusion or exclusion of operations may be restricted by the choice of supertypes.

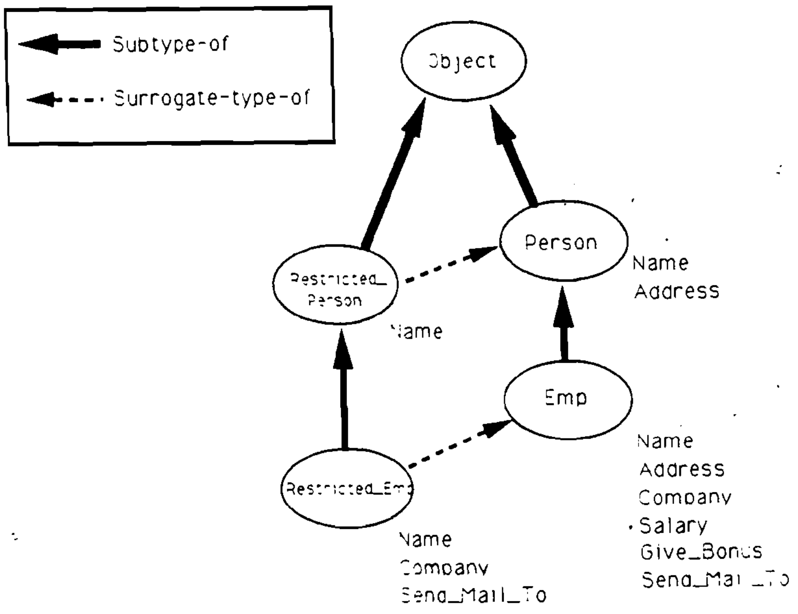

Figure 2: Hierarchy containing Restricted_Emp surrogate-type

previously, operations defined on the base type may be explicitly copied to the surrogate-type. For the purposes of type-checking, a surrogate-type is treated like any other type in the type hierarchy. Additional properties may also be defined for a surrogate-type. If an added property is to be stored, then local storage must be provided on each instance of the base type. This storage is only available when the surrogate-type defining the added property is the current interface of an object. Local storage for a surrogate-type is implemented using the chunking method described in [Zdo88]. Properties local to a surrogate-type are stored in a separate 'chunk' of each object. A new stored property must specify a default value which is propagated to an object when the property is first accessed on that object.

The regular (as opposed to surrogate) types form a type hierarchy in the context of the base schema of a database. Surrogate-types form *parallel* pieces of the type hierarchy, illustrated in figure 2 [5], that can overlay pieces of the base type hierarchy within the context of a particular database view. This restricts the meta-level behavior of a surrogate-type. We have already stated that instances of a surrogate-type cannot be created unless the creation occurs in the context of a view where the surrogate-type replaces its base type. We have also placed restrictions on the definition of properties and operations for a surrogate-type. Further restrictions are a consequence of ENCORE's semantics for the collection of a type's instances.

As described above, a surrogate-type must "hook in" to the same path of the specification hierarchy as its base type. Section 2 states that the instances of a subtype are included in the collection of instances of each supertype. In order for a surrogate-type to replace its base type within a view, the subset relationship of the surrogate-type's instance collection to that of its supertypes must be maintained thereby restricting the choice of supertypes. Subtypes of a base type might not be subtypes of a surrogate-type associated with that base type. If this is the case, the subtype and the surrogate-type cannot both exist within the same view because a subset relationship of instances of the two types will exist while a subtype relationship between the two types does not exist. A new surrogate-type of the subtype in question may be defined to be a valid subtype of the original surrogate-type in which case both surrogate-types may exist within a single view. A final restriction is that a regular subtype cannot be created from a surrogate-type. This is because the subtype relationship may not exist between the type being created and the base type of the surrogate-type

[5] Restricted_Emp could have been made a subtype of type Person if the address property were included in its public interface.

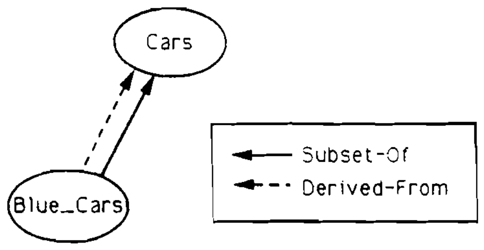

Figure 3: Blue_Cars derived class

chosen as supertype. Consequently, the subset relationship of instances of the subtype to instances of the supertype may not be maintained.

4 Derived Classes

As stated previously, ENCORE maintains a collection of each type's instances. We will now call these collections *base classes*. The name of a base class is the pluralized name of the type it is associated with (e.g., type Emp ⇒ class Emps). We extend the class mechanism by allowing the definition of *derived classes*. A derived class is created by executing a query over an existing class or classes in the context of a DEFINE CLASS statement. A derived class forms an object-level view by grouping related objects based on a query. A derived class may be included in the definition of a database view in which case the derived class is visible to be queried over in the context of that view. The result of a query is a collection of objects (the *query collection*). The query collection returned within the context of a DEFINE CLASS statement forms the initial extent of the derived class being defined. At any point in time, the members of a derived class reflect the result of the defining query as if it had been executed at that point. Therefore, the membership of a derived class fluctuates as a result of changes to the database. Objects cannot be explicitly added to or removed from a derived class, this task is reserved to the database system. The *member type* of a derived class is the type of the members of the query collection used to create the class. This may or may not be the same as the member type of the collection originally queried over. We will call the member type of the original collection the *base type*.

Consider the derived class example shown in figure 3. Type Car has an associated base class, Cars, which includes all instances of Car. A derived class is created from Cars which includes those car objects which are colored blue. This is expressed by the following DEFINE CLASS statement:

$$DEFINE\ CLASS\ Blue\_Cars\ USING\ SELECT(Cars, \lambda c\ c.color = \text{"blue"})$$

The derived class defined is named Blue_Cars and contains Car instances whose color property has a value of "blue". The member type (and base type) of Blue_Cars is Car, therefore Blue_Cars has type Class[Car]. Derived classes are basically sets and support operations such as Member_Of, Number_Of_Members, Subset_Of, Get_Member with the obvious semantics.

The extent of a derived class is stored to avoid re-computation on each reference. Furthermore, derived classes defined with a PROJECT, OJOIN, or (in some cases) IMAGE query will contain new objects which did not exist before the defining query was executed. Re-computation of the extent of these derived classes would not preserve references to class members. Storing a derived class extent requires that maintenance be performed in order to update the extent as changes are made to the database.

Once an object is retrieved from a derived class, no guarantee exists that the object will always be a member of that class. Subsequent updates to the retrieved object may invalidate its membership. Membership in a class can always be checked using the Member_Of function provided for all classes. Changes in class membership do not affect other references to an object. If a car object is retrieved (and therefore locked) from the Blue_Cars class it is safe to assume that the object is a car whose color property has value "blue". If the Paint operation is executed on the retrieved car changing the value of its color property to "red", references to the car object do not change. In other words, membership of a car object within Blue_Cars is mutable while external references to the car object remain valid and unchanged. An alternative functionality is to supply an updated reference which reflects the fact that the membership of a car object in Blue_Cars may change. For example, access through an updated reference could return a warning or error condition if the relevant car object is no longer a member of the Blue_Cars derived class. A subset of this functionality could be implemented using a computed property which embodies the desired query. However, this does not address the need for updated references which are not properties of an object (e.g., database variables, set members). Although it is a useful feature to consider, it is not further discussed in this paper.

In the sections of this paper which describe the different kinds of derived classes, we assume that only base classes exist. The creation of a derived class from another derived class is discussed in section 4.7. In the interest of saving space, we do not include derived classes created with UNION, INTERSECT or DIFFERENCE. The implementation of derived classes formed using these operators is analogous to the derived class implementations that are presented.

4.1 Closure Functions

Logically, the objects that are members of a derived class are instances of the member type of that class (e.g., members of Blue_Cars are Car objects). In fact, they are initially identical to the objects returned in the query collection used to define the class. However, in order to facilitate the maintenance of the extent of a derived class, we chose to physically implement the members of a derived class as separate objects from those which are instances of the class member type. This implementation is hidden from a user of a derived class. An object retrieved from a derived class, by invoking Get_Member, is an instance of the member type of the class. The retrieved object is *generated*, however, from a different object which is physically a member of the derived class.

The mechanism that we apply to represent logical members of a derived class is that of a closure function [GP79] (similar to the Apply function described in [Shi81]). A closure function is of the form:

$$closure(function, value_1, value_2, \ldots, value_n)$$

A closure function takes an arbitrary function as its first argument. The remaining arguments to a closure function are values bound to the arguments of the function referenced in the first argument. The result of executing a closure function is equivalent to executing the function bound to the first argument with its arguments bound to the values specified in the remaining arguments. For example, consider the function Add which takes two integers and produces their sum. A closure function binding function Add to its first argument and the integers two and seven to its remaining arguments will always produce the integer nine when executed.

The objects which are physical members of a derived class are implemented as closure functions. An object is created as a class member which embodies a particular closure function and its arguments. We will distinguish between logical members of a derived class (e.g., instances of the member type of the class) and physical members of a derived class (e.g., objects implemented as closure functions which produce logical members of the class) whenever there is ambiguity. The closure function implementation of a physical class member is executed whenever that member is referenced. The result of the function execution is an instance of the member type of the derived class or an error condition. The functions referenced in the first argument of a class member closure function generally test to see if a particular instance of the class member type is a valid participant in the derived class. For example, the closure function implementation of a member of a derived class created with a SELECT query tests to see if an instance of the class member type meets the requirements of the predicate specified by the SELECT. If it does not, the member type instance is not considered to be a participant in the derived class at that point in time. In other cases, a closure function may take an instance of the base type of a class and produce an instance of the member type of the class.

The physical members of a class are merely place holders for logical members of the class. The visibility of physical class members is obscured by the operations provided for derived classes. For example, if the Get_Element operation is invoked on a derived class, the result is an object obtained by finding a physical class member whose closure function successfully produces an instance of the member type of the class. Similarly, if the Number_Of_Members operation is invoked on a derived class, the closure function of each physical member of the class is executed. Only those executions which complete successfully contribute to the count of logical members of the class.

Closure functions provide a mechanism for testing the inclusion of a particular object in a derived class. Additional functionality is required to add or remove members of a derived class in response to external database changes. Removal of members is actually handled by closure functions in a form of lazy evaluation [Bun82]. We use actions (triggers) associated with specific operations in the database to add members to a derived class as needed. The following sections describe, among other things, the maintenance of different kinds of derived classes using closure functions and actions. Section 4.8 provides details on how actions are formed and what operations they are associated with.

4.2 SELECT Derived Classes

A SELECT query can be used to define a derived class which expresses a constraint on an existing class. The Blue_Cars example illustrates a derived class created with SELECT. Consider the following alternative example:

$$DEFINE\ CLASS\ Highly\_Paid\_Emps\ USING\ SELECT\ (Emps, \lambda e\ e.salary > 50000)$$

Highly_Paid_Emps is a derived class whose instances represent a constrained subset of the collection of Emp instances (Emps). In particular, those Emp instances whose salary property has a value greater than fifty thousand dollars are present in the extent of Highly_Paid_Emps. The member type of Highly_Paid_Emps is Emp. In SELECT derived classes, the base type of a class is always equivalent to the member type. In this example, the semantics of Highly_Paid_Emps are expressed in the SELECT constraint used to create the derived class. Other semantics may be expressed by the addition of behavior to the Highly_Paid_Emps class (see section 4.9).

The closure function implementation of physical members of a SELECT derived class is of the form:

$$closure(Check\_Select\_Predicate : Object \times Function \rightarrow Object, o, f)$$

where Check_Select_Predicate is a system supplied function, o is a member of the class SELECTed from, and f is a function which implements the relevant SELECT predicate. The execution of a closure function of this form in turn executes Check_Select_Predicate on the given object and predicate function. Check_Select_Predicate returns the object sent to it if the predicate function evaluates to true on that object. If the predicate function evaluates to false, Check_Select_Predicate returns an error condition. The object sent to Check_Select_Predicate is an instance of the member type of the class. If the execution of Check_Select_Predicate is successful (e.g., it does not return an error condition), then the member type instance is considered a logical member of the class. If Check_Select_Predicate returns an error condition, then the member type instance is not considered a valid member. In the Highly_Paid_Emps class, the arguments to Check_Select_Predicate are an instance of Emp and the predicate function used in defining the class (e.g., $\lambda e\ e.salary > 50000$) [6].

The logical extent of a SELECT derived class may be effected by updates made to the database. The most obvious examples are changes made to instances of the class member type. Creation of an instance of the member type requires the inclusion of a new member in the logical extent of a SELECT class if the relevant predicate evaluates to true on the created object. Similarly, an update may cause a predicate to become true on a member type instance requiring its inclusion in the logical extent of a SELECT derived class. An update could also falsify a predicate on an object requiring its removal from a SELECT derived class. Deletion of an instance of the member type requires that the object be removed from all derived classes as well as the base class. Updates to instances of types other than the member type of a SELECT derived class may also affect the extent of the class. For example, consider a SELECT derived class defined on Emps called Boston_Emps which includes the predicate function $\lambda e\ e.dept.city =$ *"Boston"*. An update to an instance of Dept might affect the extent of Boston_Emps by requiring insertion (in the case that a department's city is changed to "Boston") or deletion (in the case that a department's city is changed to something other than "Boston") of a class member if the updated instance is referenced by at least one Emp instance. The effect of a Dept instance update on the extent of Boston_Emps is not autonomously computable [BCL89] since objects which are not included or referenced in the extent of Boston_Emps must be consulted in order to determine the update's effect. In our model, autonomous computability bounds the number of objects that have to be referenced in order to maintain the extent of a derived class. If the effect of an update on a derived class is autonomously computable then fewer objects need to be referenced. Note that the creation or deletion of an instance of Dept does not affect the extent of Boston_Emps unless it is referenced by an Emp instance.

Maintenance of a SELECT derived class can be handled in two different ways. All of the events described in the previous paragraph are at the logical level. Therefore, one approach to class maintenance is to include an object in a derived class for each instance of the member type. Each physical member of the class will produce, via closure function execution, an instance of the member type if the instance is a valid logical member of the class. Using this implementation, the number of objects in the derived class is always equal to the number of instances of the member type. At any particular time, the number of logical class members is most likely less than the number of physical members of the class. This approach is not space efficient if the logical extent of the derived class is a small subset of the member type's base class. However, the insertion and deletion of physical class members in response to external database updates is minimized. In particular, objects need only be inserted into a SELECT class when instances of the member type are created.

The alternative approach is to immediately add or remove physical members of a SELECT derived class in response to external updates. This minimizes the number of objects that are members of the derived

[6] Note that the Check_Select_Predicate function and the predicate function are both objects and therefore are shared by all physical members of a derived class.

class at any point in time but requires more processing to reach a consistent database state when an update occurs. Additionally, this approach requires that more actions be generated and associated with operations in the database since more database events must be monitored in order to maintain a derived class extent. The choice between the two approaches can be made based on the cardinality of the query collection initially used to define a SELECT derived class versus the cardinality of the member type's base class.

In either approach, the deletion of a derived class member which corresponds to a *deleted* instance of the member type can be delayed until next reference of the class member. The next reference will perform a closure function execution to determine if the referenced member type instance is a valid logical member of the class. Check_Select_Predicate returns a distinct error condition if the object sent to it is deleted. The derived class operations which (indirectly) execute Check_Select_Predicate respond to this error condition by deleting the corresponding physical class member. A query over the extent of a derived class qualifies as a next reference so that query results are always consistent with the current state of the database.

4.3 IMAGE Derived Classes

An IMAGE query produces a new collection of objects by executing a function on each member of an existing collection. There is a bijective correspondence between objects in the original collection and objects in the resulting query collection. An IMAGE query can be used to define a derived class which maps the members of a class to the members of a derived class using function execution. Consider the class definition:

$$DEFINE\ CLASS\ Emp\_Roster\ USING\ IMAGE\ (Emps, \lambda e\ Name\_Of(e))$$

Assume that the Name_Of function returns String objects. The IMAGE query shown takes the collection of Emp instances (Emps) and produces a collection of String objects where each member of the collection represents the name of an employee. The member type of Emp_Roster is String.

The physical members of a derived class created with IMAGE are implemented using closure functions of the form:

$$closure(Update\_Image : Object \times Function \rightarrow Object, o, f)$$

where Update_Image is a system supplied function. o is an instance of the type queried over and f is the function used in the query. Executing a closure function of this form is equivalent to executing Update_Image on object o and function f. Update_Image returns the object which results from applying function f to object o. The object returned may be different at different times depending on the state of o (and possibly other objects that o references). Update_Image is executed by the closure function rather than executing f directly to handle the case that o is a deleted object without requiring f to do so. In the Emp_Roster class, the arguments to Update_Image are an instance of Emp (o) and the Name_Of function (f). The result of executing Update_Image in this case is the same as executing the Name_Of function on the referenced Emp instance. If the name of the employee has changed since the last time the function was executed, the resulting String object is different. Each physical class member is a place-holder for a String object which may change over time.

As with SELECT derived classes, the process of obtaining logical members of an IMAGE derived class involves the execution of closure functions. The difference is that instances of the base type of the class are not produced by the executions. The closure functions reference an instance of the base type, but they produce an instance of the IMAGE function return type (the member type). This process maintains the logical extent of an IMAGE derived class with regards to the extent of the class it was created from. It also allows the determination of the effect of an update to a base type instance on the logical extent of a corresponding IMAGE derived class to be deferred until next reference. Due to the bijective correspondence of an IMAGE derived class to the base class it was created from, the creation or deletion of base type instances always requires a corresponding update to the extent of the derived class. Insertion of physical derived class members is handled by attaching an action to the Create_Instance operation of the base type (as described in section 4.8). Deletion of a derived class member is deferred until next reference as shown for SELECT derived classes.

4.4 PROJECT Derived Classes

A PROJECT query is similar to an IMAGE query in that it applies functions to a collection of objects to produce another collection of objects. A PROJECT query, however, always produces a collection of tuple objects. This may be desirable to provide a common interface (e.g., Get_Attribute and Set_Attribute) for objects of numerous types. It is also possible that the well understood interface of tuples will provide better opportunities for query optimization (e.g. relational algebra transformations) [SZ89]. There is a bijective correspondence between objects in the original collection and the collection of tuples produced by a PROJECT. Like IMAGE, PROJECT can be used to define a derived class which maps the members of an existing class to the members of a derived class. An example of a PROJECT derived class is:

$$\begin{aligned} &DEFINE\ CLASS\ \ Emp\_Directory\ USING\\ &\qquad PROJECT(Emps, \lambda e < (Name, Name\_Of(e)),\\ &\qquad\qquad (Phone, Phone\_Number(e)) >) \end{aligned}$$

Emp_Directory is an abstraction of Emps which stores the name and phone number of each employee in tuple form. The query used to define Emp_Directory produces tuples with attributes Name and Phone. The values for these attributes are obtained by executing the Name_Of and Phone_Number functions on instances of Emp. The member type of Emp_Directory is a type generated from the parameterized type Tuple (e.g., Emp_Directory_Type). Note that the definition of the new type also defines a corresponding base class which includes all instances of the type.

The physical members of a derived class created by PROJECT include closure functions of the form:

$$\begin{aligned} closure(&Update\_Project\_Tuple : Object \times Tuple \times Function_1 \times \ldots \times Function_n \rightarrow Tuple,\\ &o, t, f_1, \ldots, f_n) \end{aligned}$$

where Update_Project_Tuple is a system provided function which updates a tuple (t) by executing the functions $(f_1, \ldots, f_n)$ on the given object (o) to obtain values for each attribute of t. Update_Project_Tuple returns an appropriate error condition if o is a deleted object in which case the corresponding derived class member and tuple object are deleted. The arguments to Update_Project_Tuple in the case of Emp_Directory are an instance of Emp (o), a tuple of type Emp_Directory_Type to update (t), the Name_Of function (f_1) and the Phone_Number function (f_2).

The process of accessing members of a PROJECT derived class involves the execution of closure functions. As with IMAGE derived classes, the closure function execution produces an object whose type is different from the type originally queried over. In particular, the objects produced are tuple objects. This process keeps the extent of a PROJECT derived class consistent with regards to the class originally queried over. The updating of a class member is delayed until the next reference of that member. For example, if an employee's phone number changes then the tuple object corresponding to that employee in the extent of Emp_Directory is updated when it is next retrieved from the class. This will not affect external references to the tuple if it was previously retrieved from the derived class since a new tuple object is not created when an update occurs. As with IMAGE classes, there are no predicates involved in defining a PROJECT class. Therefore, it is not necessary to immediately determine the effect of a base type instance update (other than creation or deletion) on the extent of a PROJECT derived class. Creation or deletion of instances of the base type always requires updating the extent of a PROJECT derived class. As previously, creations are handled immediately by an action attached to the Create_Instance operation of the base type of the class. Deletions are handled when the corresponding class member is next referenced.

Once a tuple object is retrieved from a PROJECT derived class, it must behave like other tuple objects in the database. In particular, the Get_Attribute and Set_Attribute operations for each attribute of the tuple must be available. Since the functions which produce attribute values on a particular PROJECT tuple may not be invertible, it is not possible to automatically provide Set_Attribute operations which update an instance of the base type when an update is made to a tuple object. For this reason, the default Set_Attribute operations provided by the system for tuple types generated as a result of a PROJECT derived class definition are null operations. These operations may be overridden by the class definer. In order to facilitate access to the object which to create a tuple in a PROJECT class, a property is defined in the private interface of the tuple type whose value is the relevant object.

4.5 OJOIN Derived Classes

OJOIN queries express new relationships between objects by constructing tuples whose attributes contain objects related by a predicate. The result of an OJOIN query is not guaranteed to be in one-to-one or onto correspondence with either of the two collections queried over. An OJOIN derived class may be defined to express a relationship between the members of two existing classes. The following is an example of an OJOIN derived class:

$$\begin{aligned} &DEFINE\ CLASS\ \ Emp\_Phones\ USING \\ &\qquad OJOIN(\ Emps, Phones, A_e, A_p, \\ &\qquad\qquad \lambda e \lambda p\ p \in e.dept.phones \land p.loc = e.loc) \end{aligned}$$

Emp_Phones is a derived class which contains pairs of employees and the phones that are in the employee's department and at the same location as the employee. The logical extent of Emp_Phones is made up of tuple objects. The A_e attribute value of each tuple is an Emp instance and the A_p attribute value is a related Phone instance. Note that there may be several members of Emp_Phones which have the same Emp instance as a value for attribute A_e. Similarly, multiple members may have the same Phone instance as a value for attribute A_p. The member type of Emp_Phones is generated from type Tuple (e.g., Emp_Phones_Type).

Physical members of a derived class created with OJOIN contain closure functions of the form:

$$\begin{aligned} &closure(Check\_Ojoin\_Tuple : Object_1 \times Object_2 \times Tuple \times Function \rightarrow Tuple, \\ &\qquad o_1, o_2, t, f) \end{aligned}$$

where Check_Ojoin_Tuple is a system provided function which applies a predicate function (f) to two objects (o_1, o_2) to determine if the predicate is true. If the predicate is true, then the tuple sent (t) is returned as the value of Check_Ojoin_Tuple. If the predicate does not hold *or* either of o_1 and o_2 are deleted, then Check_Ojoin_Tuple returns an error condition in which case the corresponding tuple and physical class member are deleted. The arguments to Check_Ojoin_Tuple in the case of Emp_Phones are an instance of Emp (o_1), an instance of Phone (o_2), an Emp_Phones_Type tuple to update (t), and the predicate function f: $\lambda e \lambda p\ p \in e.dept.phones \wedge p.loc = e.loc$. The result of executing Check_Ojoin_Predicate with these arguments is a Emp_Phones_Type tuple which contains e and p as the values for A_e and A_p respectively or an error condition. This maintains the relationship between the extent of Emp_Phones and the classes Emps and Phones.

The logical extent of an OJOIN class may be affected by updates to the database. Creation or update of an instance of either base type may require insertion of a class member if there are instances of the other base type which are matched by the relevant OJOIN predicate. An update to an instance of the either base type may also require the deletion of an OJOIN derived class member if there is no longer a match between the updated object and an instance of the other base type. Note that in order to determine the effect of a database update on the extent of an OJOIN derived class, it is usually necessary to consider sets of objects rather than individual objects. Deletion of an instance of either base type requires deletion of the derived class members, if any exist, that reference the deleted object. As with SELECT classes, the update of an object other than a base type instance may require a modification to the extent of an OJOIN derived class. In the Emp_Phones example, the OJOIN predicate includes the expression $p \in e.dept.phones$. If the phones property of an instance of Dept is modified, an Emp_Phones member may have to be inserted or deleted. If an instance of Dept is created or deleted it does not affect the extent of Emp_Phones unless the create or delete is followed or preceded by an update to an Emp instance.

Maintaining an OJOIN derived class is analogous to maintaining a SELECT derived class because a predicate is involved in both cases. There were two approaches described for maintaining a SELECT derived class. The first involves the creation of a physical class member for every member type instance. The second approach involves more processing but keeps the number of physical class members at a minimum. In the case of OJOIN derived classes, it is not feasible to create physical class members which correspond to every combination of instances of the types queried over. Clearly the cardinality of the resulting class could be quite large and would include many unnecessary members. Therefore, the second approach must be adapted for use with OJOIN classes. The cost of maintaining the minimum number of physical members of an OJOIN derived class is higher than for SELECT classes. The effects of the updates described in the previous paragraph on the extent of an OJOIN derived class (with the exception of the deletion of a base type instance) are not autonomously computable while the effect of many updates on a SELECT derived class extent are autonomously computable. Therefore, more objects must be referenced to determine the effect of an external update on an OJOIN class extent. Furthermore, the instances of *two* base types must be monitored which increases the chance that a database update will affect an OJOIN class. As with SELECT classes, the deletion of derived class members may be delayed until next reference.

A tuple object retrieved from an OJOIN derived class must behave like other tuple objects. Once again, the Set_Attribute operation defined for a type generated by an OJOIN class definition is a null operation. The class definer may replace the default operations with operations which affect the objects used to create a tuple in the class. There is no need for a hidden property which contains the objects used to create an OJOIN tuple since both of the relevant objects are available as attribute values on the tuple.

4.6 Query Modifiers

As described in section 2, the query operators DUP_ELIMINATE, COALESCE, FLATTEN, NEST, and UNNEST re-structure a query collection. They can be thought of as *query modifiers*. Query modifiers destroy the one-to-one correspondence between a query collection and the members of the collection(s) originally queried over. There may be several members of the original collection that contribute to the inclusion of a particular object in the query collection. In the case of UNNEST or FLATTEN, a single member of the original collection may contribute multiple objects to the query collection.

Needless to say, query modifiers complicate the task of maintaining a derived class extent. In previous sections, the concern was whether or not to include a particular object in the logical extent of a derived class based soley on a predicate or function executed on objects *external* to the class. Query modifiers require that additional processing be done which may involve comparison to other class members (e.g., DUP_ELIMINATE) or re-formating (e.g., UNNEST). In order to accomplish this, we supplement the closure function execution performed when a physical class member is accessed by additional function executions which are dictated by the query modifiers being used. These functions take the result returned by a closure function execution and produce a partial logical extent of the class which reflects the effect of the query modifiers on the result. If no query modifier is present, the result of a closure function execution is inserted into the logical extent being constructed. The implementations of derived class operations (e.g., Member_Of, Number_Of_Members) use the temporary logical extent as necessary to perform their designated tasks.

Using the DUP_ELIMINATE modifier as an example, the inclusion of an object in the logical extent of a derived class must be coupled with a test to determine if a duplicate object is already present. If the Number_Of_Members operation is invoked on a derived class whose definition includes DUP_ELIMINATE, each physical member of the class is considered for its contribution to the logical extent of the class. Previously, those physical members whose closure function produced an instance of the class member type contributed to the count of logical members of the class. In this case, successful closure function execution must be followed by a test to see if the member type instance produced is already present in the logical extent created by executing the closure functions contained in previous physical members of the class. If the object is already present, it is not added to the logical extent and therefore does not contribute to the count of logical members of the class. The test for duplication is on the objects resulting from closure function execution rather than on the physical class members themselves since the physical members are only place-holders for the objects that they generate.

Similar functionality is provided for other query modifiers. Nested query modifiers are handled by composing the functions needed to maintain the logical extent of the class. For example, an UNNEST modifier nested within a DUP_ELIMINATE modifier requires that the initial logical class member be UNNESTed and that each resulting object is testing for duplicity. Only those UNNESTed tuples which are not already present in the logical extent of the class will be added. The processing required by query modifiers is performed every time a derived class is referenced.

4.7 Nested Queries

The preceding sections have used simple queries to illustrate the various kinds of derived classes that can be defined. It has been assumed that only base classes were available for use in defining derived classes. It is certainly desirable to be able to create derived classes using nested queries and other derived classes. The two cases are similar since an existing derived class represents the query that was used to define it. Creating a new derived class from an existing one is analogous to nesting the query used to define the existing class within the query being used to define the new class.

We categorize the classes created with nested queries into those that do not involve a query modifier on an inner query and those which do. An outermost query modifier does not affect the categorization. In the first category, it is possible to create a derived class from a nested query by simply nesting the closure functions that are necessary to represent each query operator. This is because the membership of an object in the derived class can be determined using a single fixed object (or a pair of objects in the case of OJOIN). Note that the process of executing a closure function is now more complicated since it is necessary to handle nested executions.

An example of a nested query which does not have an inner query modifier is the following:

$$SELECT(IMAGE(Emps, \lambda e\ Car\_Of(e)), \lambda c\ c.color = \text{“}blue\text{''})$$

This query first executes Car_Of on each instance of Emp and then selects those cars whose color property has a value of blue. The nested closure function needed to represent this query is:

$$closure(Check\_Select\_Predicate, closure(Update\_Image, e, Car\_Of), f)$$

where e is an instance of Emp and f is the predicate function $\lambda c\ c.color = \text{“}blue\text{''}$. Clearly, it is possible to start with an employee and determine if a corresponding blue car is available to include as a member of the derived class without considering other employees or cars.

If the derived class is created from an existing derived class (which also does not involve a query modifier), the closure function representation used in the existing class can be nested within the closure function representation of the new class. For example, suppose the following pair of derived classes were defined:

$$DEFINE\ CLASS\ Emp\_Cars\ USING\ IMAGE(Emps, \lambda e\ Car\_Of(e))$$

$$DEFINE\ CLASS\ Blue\_Emp\_Cars\ USING\ SELECT(Emp\_Cars, \lambda c\ c.color = \text{“}blue\text{''})$$

The closure function representations corresponding to these two classes are:

$$closure(Update\_Image, e, Car\_Of)$$

$$closure(Check\_Select\_Predicate, closure(Update\_Image, e, Car\_Of), f)$$

The second representation is equivalent to the one formed by defining a single derived class using the nested form of the query to obtain blue employee cars.

The second category of nested queries are those which contain inner query modifiers. Creating a derived class from an existing derived class which involves a query modifier is considered to be equivalent. Assume the nested query used to define a derived class is the following:

$$SELECT(DUP\_ELIMINATE(IMAGE(Emps, \lambda e\ Dept\_Of(e))), \lambda d\ d.loc = \text{“}Boston\text{''})$$

This query retrieves all unique department objects which are referenced by Emp objects and are located in "Boston". A first approximation of the nested closure function needed to represent a logical member of the collection returned by this query is:

$$closure(Check\_Select\_Predicate, closure(Update\_Image, e, f_1), f_2)$$

where f_1 is the Dept_Of function and f_2 is the function $\lambda d\ d.loc = \text{“}Boston\text{''}$. Note that this does not include a test for the duplicity of the object returned by the nested closure. Therefore, assuming that many employee objects reference the same department object, the resulting derived class will contain many duplicate members. The membership decision cannot be made without considering the entire collection created by the inner IMAGE and DUP_ELIMINATE query.

We need a slightly different approach to represent the members of a derived class created with a query in the second category. The first step is to create a derived class using the inner query which includes the query modifier. For example, we can create the class Emp_Depts to represent the query:

$$DUP\_ELIMINATE(IMAGE(Emps, \lambda e\ Dept\_Of(e)))$$

The second step is to create a derived class which corresponds to the original query. The representation of this derived class refers to the members of the intermediate derived class. In particular, the closure function representation for the second derived class is of the form:

$$closure(Check\_Select\_Predicate, d, f)$$

where d is an instance of type Dept (the member type of Emp_Depts) and f is the function $\lambda d\ d.loc = ``Boston" \wedge Member\_Of(Emp\_Depts, d)$. This representation reflects the fact that it is necessary to consider the entire set of objects produced by the inner query in order to obtain the correct result for the outer query. It is not sufficient to say that d is defined as a member of Emp_Depts since the membership of a derived class fluctuates. Object d must be an instance of the member type of Emp_Depts which is then explicitly tested for membership in the Emp_Depts class. This is not surprising since every predicate implicitly includes a test of the form $Member\_Of(base\ class, o)$ which is not included since the membership of an object in a particular base class is guaranteed as long as the object exists.

Similar approaches can be used for the other query operators and modifiers. Some modifiers (such as UNNEST) may need only a subset of the collection produced by the inner query, but the same general approach can be used. It is important to mention that the construction of closure function representations for derived class members is based only on the syntax of the query involved. Two equivalent queries which are syntactically different will have different closure function representations.

4.8 Derived Class Maintenance using Actions

Derived class extents are maintained using a combination of closure functions and actions attached to specific operations in the database. The utility of procedural attachment for performing database maintenance is described in [Day88]. Actions associated with an operation are executed after an invocation of that operation has completed. There may be several actions associated with any given operation. Actions are executed in the order in which they are attached to an operation. The placement of actions is determined by the kind of derived class being defined. In all cases, actions must be associated with the Create_Instance operation of the base type or types of the derived class. For example, the action associated with the Create_Instance operation of type Emp as required by the derived class Emp_Roster is:

```
e = <new Emp object>
Insert_Member ( Emp_Roster,
   Create_Closure_Function_Object ( Update_Image, e, Name_Of))
```

A derived class definition which involves a predicate (e.g., SELECT, OJOIN) requires testing the relevance (as defined in [BCL89]) of a database update on the extent of the class. In the case of a base type instance creation, a member of the derived class should only be created if the base type instance created meets the predicate used to define the derived class [7]. For example, in the derived class Highly_Paid_Emp the action associated with the Create_Instance operation on type Emp is:

```
e = <new Emp object>
if Get_Property_Value ( e, salary ) > 50000 then
   Insert_Member ( Highly_Paid_Emps,
      Create_Closure_Function_Object ( Check_Select_Predicate, e, predicate))
end
```

Additionally, derived classes involving a predicate require parsing of the predicate to determine which aspects of a type's interface are referenced. This is necessary in order to attach actions to operations in the database which respond to updates other than base type instance creation. For example, in the Emp_Phones derived class defined in section 4.5, the OJOIN predicate references the dept and loc properties of Emp, the phones property of Dept, and the loc property of Phone. If a property of a type is referenced in a predicate, then an action must be attached to the Set_Property_Value operation associated with that property. In the case of Emp_Phones, an action must be attached to the loc property of Phone (among others) which determines if instances of Emp_Phones must be added to correspond to the update of a phone's location [8]. The necessary action is shown below:

```
p = <updated Phone object>
for each instance of Emp do
   e = <current Emp object>
   if p is in Get_Property_Value(Get_Property_Value(e,dept),phones) and
      Get_Property_Value (p, loc) =
         Get_Property_Value(<Emp object>, loc) then
      insert ( Emp_Phones,
         Create_Closure_Function_Object ( Check_Ojoin_Tuple, e, p,
           Create_Instance ( Emp_Phones_Type, e, p), <predicate function>))
   end
```

If one of the operations defined on a type is referenced, then actions must be attached to all Set_Property operations on that type. This is because there is no way to determine, short of code inspection or explicit declaration, which properties of a type an operation accesses. An update to any of the properties associated with a type of a particular object may affect the result of invoking an operation on that object. For example, consider a derived class whose member type is Person which includes a predicate with the expression $Assets(e) > 10000$. The Assets operation on type Person may reference several properties of person including car, salary, and bank_account. A change to the value of any one of these properties on a Person object could affect the result of invoking Assets on that object. It is possible for an operation used in a predicate to reference properties or operations on object of other types. This is regarded as a degenerate case: associating actions with the operations of these types is left to the class definer.

[7] In the case of SELECT derived classes, this is only relevant if the second approach to class maintenance is used.
[8] Deletion of members of Emp_Phones is handled by closure execution.

4.9 Class Behavior

The operations and properties defined by a type describe inherent features of an instance of that type. We consider it useful to be able to define additional behavior for an object based on membership within a particular class. This behavior is dynamic in that it is only available on an object as long as that object is a member of the class that the behavior is associated with. This is analogous to the notion of roles in data modeling [Che76, Haw84, Shi81] or to attaching behavior to individual objects [LRV88]. Defining class behavior is different from defining multiple types for an object because a fixed set of behavior is guaranteed, as described by an object's type, regardless of additional class behavior. For example, a tax-consultant property and a Make_Charity_Contribution operation may be relevant to Emp instances which are also members of the Highly_Paid_Emps class. In this case, the added features are available only when an employee plays the role of a highly paid employee.

Class behavior is separate from and in addition to the type specification of members of a class. In particular, added class behavior is not relevant to the substitutability of one type for another. Class behavior is, by default, collected and stored with the member type object of the class it is defined for. Class behavior may also be specifically associated with a surrogate-type of the member type of a class. Distinct behavior from more than one derived class associated with a particular type must have distinct names. A class feature may be shared among several derived classes associated with a type in which case the name and signature of the feature is also shared. Since class behavior is associated with a type, class operations may access public, internal, and private features of that type. Class properties may require local storage on the objects that are logical members of the class. Once again, this is implemented using the "chunking" method described in section 3. Class properties or operations may not have the same name as a feature of the type that they are associated with. If a class property or operation is specified which has the same signature as a private property or operation, it is assumed that the private feature is being "promoted" to a class interface. The usefulness of this feature will become apparent when we describe security as provided by a view (see section 5.2).

In order to invoke a class operation or assign a value to a class property on a particular object, a runtime check must be made to determine if the object is a member of the relevant derived class. If it is, then the invocation or assignment is allowed to continue. If not, a runtime error occurs. The membership of an object in a class may change as a result of database updates. In other words, the class membership set for a particular object is dynamic. This introduces a level of runtime checking into the system which did not previously exist. The runtime testing does not involve type-checking per say since the type of any object can be statically determined and therefore checked at compilation. Rather, the runtime testing is for class membership (e.g., categorization).

5 Views in ENCORE

We are now ready to provide a definition for database views in ENCORE. Surrogate-types and derived classes are extensions which provide specific functionality in the form of object-level views. A database view is a framework for combining these extensions with base types and classes in a manner which creates a context in which a user can access the database.

Formally, a view is a pair $V = (T, C)$ where T is a set of types and C is a set of classes. T enumerates the types (interfaces) that are visible in V. It is assumed that ENCORE's basic types (e.g., Object, Type, Integer, String, etc.) are always included in the set T. A base type and its surrogate-types form a set of types T_i. Only one member of any given T_i may be present in T. This guarantees that the type of all objects visible in V can be statically determined. It type t is included in T, then one of two conditions must hold. Either t's declared supertype, t_s, is a member of T or none of the types in the set T_i which includes t_s are visible. This maintains the consistency of the subtype relationship between types in T and the subset relationship between classes in C. For example, refer back to the type hierarchy shown in figure 2. If Restricted_Emp is a member of T, then either Restricted_Person is also a member of T or neither Restricted_Person or Person is a member of T. In other words, Restricted_Emp and Person cannot both be members of T since the subset relationship exists between the two respective base classes while the subtype relationship does not exist between the two types.

C enumerates the classes that are visible for the purposes of querying. A class c may be a member of C if its member type, or one of its surrogate-types, is a member of T. Objects that are not members of any $cinC$ might be obtained as the result of operation invocation on an object which is a member of some $cinC$. The types of these objects must also be members of T if further access to the objects is to be allowed. The types may not be present in T if access to the objects is to be denied (see section 5.2). In general, the argument and return types of all operations defined on $t \in T$ must also be members of T in order for V to be considered *complete*. The class behavior available for members of a class $cinC$ is dictated by the member type (either the original member type of one of its surrogate-types) for the class. The argument and return types for class operations and properties must also be included in T in order for V to be complete.

A view definition is an object in the database and includes properties which enumerate the members of T and C. It also includes operations to add or remove type and/or classes from the view and to check the completeness of the view.

A view affects the availability of objects and describes the interface of visible objects at the *user* level. User programs and queries are compiled or interpreted in the context of a view. A user may not change database views within a program or query session or operate in the context of two views. Other database operations which occur (e.g., invoking a supertype or base type operation. actions to update derived classes, etc.) may switch to execute in the context of the base view of the database. This ensures that all references by these operations are statically resolved. As stated previously, the base view includes all base and surrogate-types (although surrogate-types are not instantiable within the context of the base view), all classes, and all class behavior. Changes to the definition of types or classes is always done in the context of the base view.

The types and classes shown in figure 4 form the base schema of a sample database of people and vehicles. In the interest of saving space, basic types such as Type, Class, Integer, String, etc. are not shown in the figure. A view can be created for this database which restricts query access to the Boston_Emps and Blue_Cars classes and applies the Restricted_Emp surrogate-type as a replacement for base type Emp. The view definition which meets these restrictions is $V = ((Restricted\_Emp, Car), (Boston\_Emps, Blue\_Cars))$. A user of this view may issue queries over the Boston_Emps class and may invoke operations on the resulting Emp objects. The result of operation invocations may remove a particular Emp object from the Boston_Emps class, but existing references within the context of the view are still valid. The interface provided for all Emp objects in the context of this view is specified by Restricted_Emp. Queries may also be executed on the Blue_Cars class to obtain a Car object which can be accessed using all of the operations defined on type Car. A car object which is not a member of Blue_Cars may be obtained by getting the value of the car property on an Emp object. Membership of the resulting Car object in the Blue_Cars class can be tested using the Member_Of function. Car objects obtained in this fashion are also accessed using the operations defined on type Car.

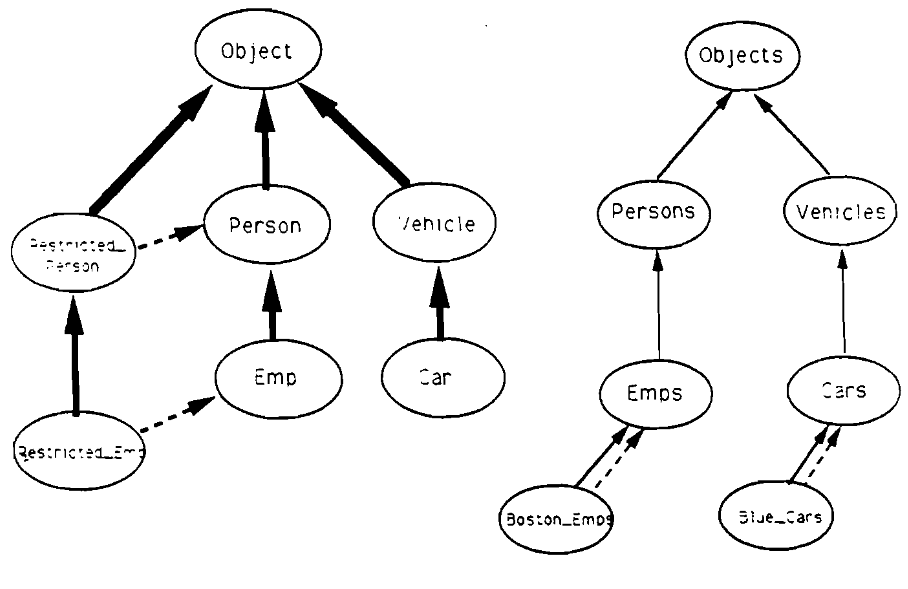

Figure 4: Sample type and class hierarchies

5.1 View Updates

Multiple views of a database may be active at the same time. Updates that occur in the context of one view are visible to users accessing the database in the context of a different view. In our model, views are defined in terms of types and classes. First we consider the case that updates are made to an object through different interfaces in different views. Updates cannot be made to an object through different interfaces in the *same* view due to the restriction that only a base type or one of its surrogate-types may be visible in a single view. Each interface of an object shares a common set of properties (e.g, those defined on the base type) whether visible on the particular interface or not. A surrogate-type may include additional local properties. If an update is made to the value of a property which is shared among all of the interfaces of an object, the update is visible to all interfaces and therefore is available to all users of that object (either directly or indirectly depending on the visibility of the property). Updates made to properties local to a particular interface are only visible through that interface and are therefore available to all users of that interface.

We now consider updates made to objects which affect class membership. Base class membership can only be affected by deleting an object in which case the deleted object is removed from the database and is not visible in any view. The membership of an object in a derived class can change when the object (or some other object) is updated. When an object is removed from a derived class, this change is visible in all views which include the derived class. The removal of an object from a derived class affects the ability to obtain that object using a query in a particular view. It does not affect other references (e.g., as a property value) to the object within a view. Conversely, an object may be updated such that it becomes a member of a derived class, in which case it is available to be queried in all views which include the derived class. In summary, changes in class membership of an object occur in a uniform fashion regardless of the visibility of a class within a view.

An important feature of a view is its consistency. [GPZ88] defines a view as being consistent if an update made in the context of the view uniquely determines an update to the underlying database. Consistency is important in order to understand and therefore reason about the semantics of updates made at the view level. It is based on the notion that a translator exists which can map view updates to underlying database updates without ambiguity. Views in Encore are consistent. Updates made through different interfaces of an object unambiguously update the state of that object. This is because the necessary translators are encoded in the operations provided on each interface. The operations are specified by the type/view definer and will, by definition, produce un-ambiguous results. Similarly, the rules for inclusion of an object in a derived class are deterministic and apply across all views. Therefore, updates made to an object which affect its class membership will affect it in the same way regardless of the view in which the update was made.

5.2 Security

Views can be used to provide a level of security for the objects stored in the database. This is accomplished by creating derived classes which hide objects and surrogate-types which hide features of objects and then combining them in a view. Security, as implemented by views, is complicated in an OODB by the ability of an object to reference other objects using their identity.

A view defines all of the types that are available to a user of the view. All references to the database are made in the context of a view. Objects whose type is not included in the current view may not be operated upon as long as that view remains the context of reference. For example, assume Emp is a type in the current view and Resume is a type not included in the current view. If Emp has a property called Current_Resume whose value is of type Resume, then the value of the Current_Resume property for an Emp instance may be retrieved but not accessed. In other words, an operation invocation is not allowed on a retrieved Resume object. A better solution would be to create a surrogate-type of Emp which does not include the Current_Resume property in its public interface. The surrogate-type can then replace Emp in a view to restrict access to resumes.

Another example of providing data security with a view is illustrated by the following; assume Emp is a type with properties name, address, manager, and salary where the manager property is of type Emp. It might be desirable to allow access to an employee's manager but not to allow access to that manager's salary. This can be accomplished in our view model with the following sequence of steps. First a derived class of Emps is created using SELECT to include all Emp instances which are also managers (e.g., Managers). Another derived class is created using SELECT which includes all Emp instances which are not managers (e.g., Non_Manager_Emps). A surrogate-type is then defined on type Emp which provides the same interface as Emp except the salary property is not included (e.g., Restricted_Emp). Note that the type of the manager property on Restricted_Emp remains Emp. The property salary is specified as class behavior for Non_Manager_Emps and explicitly associated with the Restricted_Emp surrogate-type. In this case, salary is a private property of Restricted_Emp which is promoted to the class interface for members of Non_Manager_Emps. This allows access to the salary property despite its being part of the private interface of Restricted_Emp at the cost of a runtime check for membership in Non_Manager_Emps. Finally, a view can be defined which includes the surrogate-type Restricted_Emp in place of Emp and the classes Managers and Non_Manager_Emps.

Note that the issue of user ownership of objects is not addressed by our view model. For example, it might be desirable to allow a particular user to only see the objects that they own. This can be accomplished at some level by creating separate views for each user which include only the objects that the user owns. This approach does not address access to other objects which may be obtained by following references from visible objects. It also would require a great deal of view definition maintenance to add and remove objects belonging to a particular user. Therefore, we consider ownership of objects to be an orthogonal issue to views.

6 Other Work

Other systems provide features which are similar to our definition of surrogate-types, derived classes, and database views.

Postgres [SHH88] and Rigel [RS79] are extended relational systems which include view definitions. In order to define a view in Postgres, a rule (query) is supplied which defines the extent of a view relation. If the rule defining a view is evaluated as needed, then the extent of a view relation is re-created each time query access is required. If the rule is evaluated early, then the view extent is stored and re-created each time a relevant update is made to the base relations. Updates to Postgres views are not allowed since the result of the updates cannot be automatically and un-ambiguously specified using the rule mechanism currently provided by the system. Rigel takes a more object-oriented approach in that relation definitions are partitioned into modules which include relation schemes and operations which may be performed on the relations defined by these schemes. A view is defined by creating new module which includes a query for defining the extent of a view relation, the scheme for that relation, and a complete set of operations which may be performed on the relation. These operations are supplied by the module definer and are used to address the problem of view updates. Only those operations which are specified by the module definer may be used to update the relations defined in the module. The definition of these operations is simplified by the ability to directly reference the base tuples responsible for the inclusion of a tuple in the view relation. This is unusual for a view model in the context of a relational system.

SDM [HM81] includes subclasses which are defined over base classes using predicates or grouping expressions. Subclasses are analogous to our SELECT derived classes and a subset of IMAGE derived classes. The SDM model does not include behavioral features of objects. New state can be added to objects which are included in a subclass. We also include new behavior for members of a class. The combination of subclasses and subclass features provides a basic form of viewing in SDM although the *visibility* of a subclass and the features described by a subclass cannot be controlled.

The DAPLEX [Shi81] functional data model is based on entities (objects) and functions. Types are defined in terms of the functions that can be applied to entities which are to be associated with a type. An entity can have more than one type associated with it. The set of types that an entity has may change dynamically. Subtypes can be defined which inherit function definitions from existing types. A new type may be derived from the definition of an existing type by defining functions which use those provided by the existing type. A subset of the entities which exist in the database can be associated with the new type by using a query. Views provide a new name space in which to define new types and collections of entities which are visible to a user of the view. Our model maintains a stronger notion of types; multiple types may be defined for an object, but only one may be active at any given time. Furthermore the types (base and surrogate) that are associated with an object can only be added to, no types can be removed. We separate the definition of new collections (derived classes) from the definition of new types. This is advantageous because the type of an object is fixed regardless of which collections that it belongs to. In DAPLEX, an entity may have a particular type by virtue of being in a particular collection. If an update to the entity removes it from the collection, it also removes the type. This creates a problem in viewing. If a type is removed from an entity such that no type associated with the entity is visible within a view, then the entity becomes undefined in the context of that view. In other words, the entity leaves the view. This cannot happen in our model. If the type of an object is included in a view definition, then the object is *always* defined in the context of the view. Once again, this is a consequence of our separating the definition of types from collections.

[HZ88] describes a viewing model in the context of the FUGUE data model which is based on DAPLEX. FUGUE's definition of a database view is similar to ours in that it includes a pair of sets, one specifying types and one specifying objects. They describe how views (at the object-level) can be used to implement data abstraction and inheritance. Function execution is accomplished by changing context from one type to another dynamically in order to execute the necessary functions to accomplish a particular task. This is analogous to our definition of surrogate-types where each surrogate-type is an abstraction on its underlying base type. FUGUE has a similar problem to DAPLEX in that objects may leave a view as the result of an update which invalidates an object's membership in the extent of a type visible within the view. A later FUGUE paper [HZ90] addresses this issue by not allowing updates which would remove an object from the extent of a view type; a restriction which they define as "value closure".

Database systems which apply a functional model [FBHPC+87, MD86] include many of the features described above. PDM does not allow an object to have more than one type, but an object may change type. IRIS allows more than one type to be associated with an object. An interesting addition in both cases is the ability to store the extent of a function. A stored function is represented as tuples which pair values of the input arguments with corresponding output arguments. This is analogous to a derived class created using a combination of PROJECT and OJOIN. PDM allows the definition of a new type whose instances are the tuples in a stored function extent. IRIS restricts stored functions to being in first normal form and does not associate a type with the tuples in a stored function extent.

Most of the functional models are not strongly typed in that an object may have multiple types associated with it and/or can change type dynamically. We now consider models which have a notion of typing that is closer to that of ENCORE. Galileo [ACO85] is a database programming language which includes viewing features which are similar to those in our model. The definition of a type may be modified within different "environments" by dropping or adding features which is analogous to our definition of surrogate-types. Galileo maintains collections of objects associated with a type (classes). Subclasses can be defined from an existing class using several selection methods. A subtype of the member type of a class may be associated with objects which are members of a subclass created from that class. Objects may therefore have multiple types by being members of multiple classes. Galileo maintains strong typing by not allowing an object to change types. This is accomplished by restricting the definition of a subclass to reference only immutable aspects of an object. We consider approach to classification to be restrictive and allow the movement of objects between classes. Our objects have only a single type in a given view but may add additional behavior as a consequence of class membership. We feel that this approach is more flexible and natural.

The viewing model described in [SS89] deals with defining multiple interfaces for an object. It does not consider the visibility of objects in a view. The approach outlined is to provide multiple interface definitions based on a single (base) type. Each new interface is a modification of the base type produced by including, excluding or re-implementing methods and instance variables (properties). The ability to read or write properties is also addressed. An access made to an object must specify which of the multiple interfaces associated with the object's type is the desired interface for that access. Multiple copies of the state of an object are kept; one for each "instantiation" or usage of the object. Each instantiation is a pairing of object and interface which is dictated by the current view. This has the disadvantage that updates made to an object in the context of one instantiation are not visible to other instantiations of the object. In other words, instantiations are distinct and can be thought of as versions of the original object which operate using one of the available interfaces for the object.

ORION [BCG+87], GemStone [MS87], and VBASE [AH87] are database systems which are similar to ENCORE. Objects have a single type associated with them which is not changeable. VBASE includes the notion of a union type which can be thought of as an object-level view where the actual interface to an object referenced via a union type is not fixed. In general, however, the viewing capability provided by these systems is limited to making changes to a type's definition within the database schema. This does not address the need for multiple views of an object or database to be available at the same time for different users. [KC88] proposes a schema and type versioning model which extends the data model of ORION. Type versioning could begin to support some of the viewing capabilities that we have described. However, objects in the proposed model may only have a single type version associated with them. This does not provide the flexibility of multiple interfaces for the same object.

Many of the systems described include the ability to create collections of objects based on queries or function application. The maintenance of these collections in response to database updates is a problem in all cases. The maintenance of computed extents is orthogonal to the problem of determining the affect of a view update on the underlying database. This problem is handled in most systems by user written operations or functions. Rigel and Postgres avoid the issues of extent maintenance by re-creating the extent of a view relation (which is analogous to a derived collection) for each reference. DAPLEX and PDM include rules attached to database operations which update derived collections of objects. In this paper, we describe a method for maintaining derived classes using closure functions and procedural attachment. Our contribution is a comprehensive description which describes the different kinds of derived classes and how they must be maintained. Furthermore, our approach uses a combination of early and late evaluation to decrease the overhead of class maintenance.

7 Summary

We have described a flexible implementation of database views based on the ENCORE data model extended to include two kinds of object-level views. The viewing capability described is at least equivalent in modeling power to the viewing mechanisms provided by relational database systems. Surrogate-types provide multiple interfaces which may replace a type within a database view. This allows the view definer to simplify or abstract the interface of a set of objects. Surrogate-types provide functionality similar to relational project by dropping features from the interface provided by the base type. Surrogate-types are much more powerful, however, because new features can also be added to those specified by the base type. Additionally, surrogate-types handle behavior as well as state. Derived classes provide the ability to create dynamic sets of objects which correspond to queries over the database. This allows the view definer to limit query access to those objects which are members of specific classes. Derived classes provide functionality similar to a relational select, project or join. They are more powerful because the query operators provided in the ENCORE query algebra may produce new types and objects as well as execute arbitrary functions to obtain query results. Derived classes may also include additional behavior for the objects which are members of a class. View updating is a difficult problem for relational database systems. Our model addresses the problem of view update by applying object identity and user defined operations to deterministically map view updates to the underlying database.

An interesting feature of our model is te ability to specify type and class behavior for objects. We feel that it is useful to distinguish between inherent and dynamic features of an object. A type defines static features of an object that are always available and may be statically checked. Additional features may be provided by the inclusion of an object in a class. These features are considered to be dynamic or short-term and therefore require that dynamic checking be performed before they are accessed. This maintains static typing while allowing the flexibility of dynamic behavior. Our approach is more efficient than assigning multiple types to an object since a runtime check for the validity of an object access need only occur in some cases rather than every time.

A difficult problem for OODBs, and database systems in general, is how to handle changes in the definition of a type (e.g., schema modification). Our view model may provide some leverage into addressing these problems. For example, multiple versions of a type produced by changing a type's definition over time can be modeled using surrogate-types of the original type definition. New properties or operations to be added to a type's definition can be included in a surrogate-type's definition. Similarly, properties and operations removed from a type's definition can be dropped in a surrogate-type's definition. The most difficult problem is how to handle changes in the domain of a property or the domain, range *and* implementation of an operation. A simple way to handle these changes is to drop the old definition of a property or operation and include the new definition, using a different name, on a surrogate-type. This method does not allow old programs written using a previous version of a type to access objects created using a new version of the type. Some sort of mapping from the old definition to the new definition would have to be provided in order to accomplish this. In the future, we will be looking at the applicability of views in addressing the problems of schema modification.

8 Acknowledgements

The author wishes to thank Professor Stan Zdonik for his direction in forming and refining the model presented here. Page Elmore and Gail Shaw also greatly contributed by participating in many useful discussions on this and related topics and by reviewing drafts of the paper. Thanks to both of them.

References

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM TODS*, 1985.

[AH87] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *Proc. of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*. ACM SIGPLAN, October 1987.

[BCG+87] J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H. Kim. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 1987.

[BCL89] J. Blakeley, N. Coburn, and P. Larson. Updated derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, September 1989.

[Bun82] P. Buneman. An implementation technique for database query languages. *ACM TODS*, June 1982.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 1988.

[Che76] P. Chen. The entity-relationship model: Towards a unified view of data. *ACM TODS*, 1976.

[CM84] G. Copeland and D. Maier. Making smalltalk a database system. In *Proc. of International Conference on Management of Data*. ACM SIGMOD, June 1984.

[Dat88] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, 1988.

[Day88] U. Dayal. Active database management systems. In *Proc. of Conference of Data and Knowledge Bases*, 1988.

[ESZ89] P. Elmore, G. Shaw, and S. Zdonik. The ENCORE object-oriented data model. Technical report, Brown University, November 1989.

[FBHPC+87] D. H. Fishman, D. Beech, E. C. Chow H. P. Cate, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 1987.

[GP79] S. W. Galley and G. Pfister. *The MDL Programming Language*. Laboratory for Computer Science, MIT, 1979.

[GPZ88] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM TODS*, December 1988.

[Haw84] I. T. Hawryszkiewycz. *Database Analysis and Design*. Science Research Associates, Inc.. 1984.

[HM81] M. Hammer and D. McLeod. Database description with a semantic data model: SDM. *ACM TODS*, 1981.

[HZ88] S. Heiler and S. Zdonik. Views, data abstraction, and inheritance in the FUGUE data model. In *Proc. of 2nd International Workshop on Object-Oriented Database Systems*, September 1988.

[HZ90] S. Heiler and S. Zdonik. Object views: Extending the vision. To be published in Proc. Of 6th International Workshop on Data Engineering, 1990.

[KC86] S. Koshafian and G.P. Copland. Object identity. In *Proc. of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*. ACM SIGPLAN, September 1986.

[KC88] W. Kim and H. Chou. Versions of schema for object-oriented databases. In *Proc. of 14th VLDB Converence*, 1988.

[Kel86] A. Keller. The role of semantics in translating view updates. *IEEE Computer*, pages 63–73, January 1986.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proc. of ACM International Conference on the Management of Data*. ACM SIGMOD, May 1988.

[MD86] F. Manola and U. Dayal. PDM: An object-oriented data model. In *Proc. of International Workshop on Object-Oriented Database Systems*, September 1986.

[MS87] D. Maier and J. Stein. Development and implementation of an object-oriented dbms. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. MIT Press, 1987.

[RS79] L. Rowe and K. A. Schoens. Data abstractions, views, and updates in RIGEL. In *Proc. of International Conference on Management of Data*. ACM SIGMOD, May 1979.

[SHH88] M. Stonebraker, E. Hanson, and C. Hong. The deisgn of the POSTGRES rules system. In M. Stonebraker, editor, *Readings in Database Systems*. Morgan Kauffmann, 1988.

[Shi81] D. Shipman. The functional data model and the data language DAPLEX. *ACM TODS*, March 1981.

[SS89] J. Shilling and P. Sweeney. Three steps to views: Extending the object-oriented paradigm. In *Proc. of Conference on Object-Oriented Programming Systems. Languages and Applications (OOPSLA '89)*. ACM SIGPLAN, October 1989.

[SZ89] G. Shaw and S. Zdonik. An object-oriented query algebra. In *Proc. of Second International Database Programming Languages Workshop*, 1989.

[Ull88] J. Ullman. *Princpiles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Inc., 1988.

[Zdo88] S. Zdonik. Object-oriented type evolution. Technical report, Brown University, 1988.