The Management of Changing Types in an Object-Oriented Database

Andrea H. Skarra
Stanley B. Zdonik

Brown University
Department of Computer Science
Providence, RI 02912

April, 1986

Abstract

We examine the problem of type evolution in an object-oriented database environment. Type definitions are persistent objects in the database and as such may be modified and shared. The effects of changing a type extend to objects of the type and to programs that use objects of the type. We propose a solution to the problem through an extension of the semantic data model.

A change in the interface defined by a type may result in errors when programs use new or old objects of the type. Through the use of an abstraction of the type over time, timestamping and error handling mechanisms provide support for the type designer in creating compatible versions of the type. The mechanisms are incorporated into the behavior defined by the type and are inherited via the type-lattice.

1. Introduction

The integration of database and programming languages into a single programming system has received increased attention [Bu, CM, PSA, ZW]. The appeal of such an approach is due in large part to the overlap in ideas and issues. Of direct relevance to our research are several object-oriented programming languages based on types which may change during the execution of a program [BS, Fl]. The issue of change, though, is more complex in object-oriented databases for several reasons. Objects in a database are shared and are persistent. For example, several authors may share a document during its preparation over several months.

In a design environment type definitions often change. The evolution of types is necessary in order to accurately model our current view of the behavior of a group of objects. However, changes in a type can cause problems when incompatible versions of the type result. For example, existing application programs which manipulate objects of the type may fail when executed on newly created instances. Moreover, programs written for new instances may fail on instances of the type created before the change.

Methods such as versions have been proposed for managing change [ABBHS, KL, LDEGPWWW]. However, we sought a solution in which changes in a type are hidden such that objects of the type created before and after the change may be used interchangeably by programs.

The solution to the problem is type and change specific. Consequently, we provide a structure and method to accommodate a type designer in creating compatible versions of a type. There are cases for which compatibility cannot be maintained; however, for those cases in which a solution exists, our method facilitates its incorporation into the type structure as an extension of the data model.

2. The System

Our approach will be described in terms of the ENCORE database management system (DBMS). The system embodies a high-level semantic data model in the direction of [Ch, Co, GAL, HM, MBW, SS, Zd]. A brief summary of its most relevant features is presented in order to explain the basic system types and the model of inheritance.

2.1. Objects and Types

An object is a data abstraction with an internal representation known only to the type of which the object is an instance, and an interface consisting of a set of properties, operations, and constraints. The type of an object is itself a database object of type Type that carries the definitions of its instances' properties and operations. Types serve as templates for the form and function of their instances.

The set of instances of a particular type forms a class (*i.e.* the type's class). The term, class, is borrowed from high-level data models such as SDM [HM] and should not be confused with its use in Smalltalk [GR]. The class is represented as an object of type Class, and is implicitly created when the type is defined. As instances of the type are created, they are added to the type's class. Classes may also be explicitly created (*cf.* section 2.3). Throughout the paper we have used the following convention for designating objects. A type definition object is named *Typename*. An instance of the type is referred to as a *typename* object or a *typename*. Finally, the type's class is named *Typenames*. Other classes are named similarly as plural, proper nouns.

Types, like other objects, have an interface consisting of properties and operations. The operations defined on types are distinguished by the inclusion of the *create* operation; types alone are instantiable objects. The The *supertypes* and *subtypes* properties of types are the means by which *multiple inheritance* is defined.

In the system's model of inheritance, every instance of the type is also an instance of and inherits the properties and operations defined by each supertype.[1] The inheritance scheme defined on a group of types is represented as a lattice with arcs that correspond to the *supertypes* and *subtypes* properties of the types; the lattice is rooted at type Entity and is called a *type-lattice* (Fig.1). Alternatively, the scheme is represented using the classes defined by the types. The class of a type is

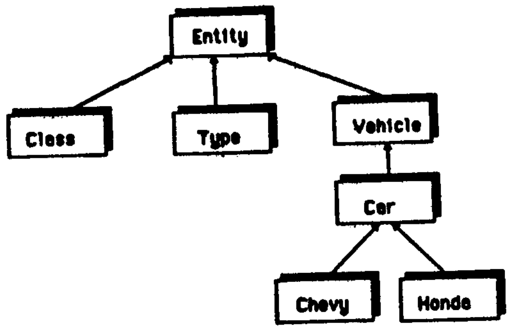

Figure 1. A type-lattice representing the *supertypes* and *subtypes* properties of several types.

subsumed by those of its supertypes (Fig. 2).

As a result, the behavior defined by a type may only represent a *specialization* or a *refinement* of that defined by its supertypes. Every property or operation defined by a type represents either an addition to those defined by supertypes (*i.e.* specialization) or a

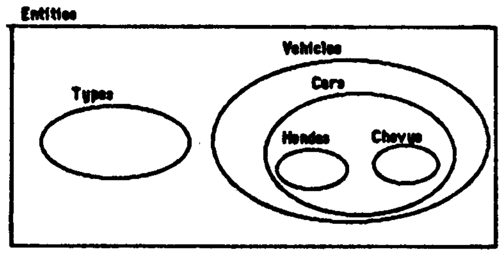

Figure 2. A Venn diagram illustrates that the class of each type is contained by those of its supertypes.

redefinition by way of refinement thereof.[2] In no case may a type nullify or *enrich* the properties and operations defined by any of its supertypes; a contradiction to the supertype's class containing the type's class would result.

2.2. Properties and Operations

The properties and operations of an object are also database objects that are instances of property and operation types. Property and operation types are subtypes and refinements of Property and Operation. Thus, the domain of values defined by a property type for its instances is contained by those defined by its supertypes. Similarly, the domains of both input and output parameters defined by an operation type are respectively contained by those defined by each supertype; the number of parameters in each case are equal.

Therefore, a type definition declares the properties and operations of its instances as objects which are members of a particular class. The class may be the set of all instances of the property or operation type, or it

[1]Notice that although an object may be an instance of multiple types, we will often use the phrase "type of an object" to refer to the object's lowest level type.

[2]When a property or operation originally defined by a supertype has been refined by a type, the system uses the refinement for objects of the type and its subtypes rather than the original definition.

may be a subset thereof. If a type refines a property or operation originally defined by a supertype, the class containing the refined property or operation is a subclass of that containing the original. Furthermore, the type of the refined property or operation is either the same as that of the original or is a subtype of it. For example, in Figure 3 the type Chevy refines the color property originally defined by its supertype Car. Consequently, chevy objects have an object from the class Chevycolors for their color property rather than an object from the class Colors. Chevycolors is a subclass of Colors, and Chevycolor is a subtype of Color.

2.3. Classes and Constraints

A class object represents a set of objects. With each class object we associate a predicate that describes the class' members. The predicate or function (f) is a Boolean combination of basis predicates, takes an object (σ) as an argument, and returns true if the object qualifies for membership in the class (*Classname*) or false if not. That is,

$$f_{Classname}(\sigma) = \begin{cases} \textit{true} & \text{if } \sigma \in \textit{Classname} \\ \textit{false} & \text{otherwise.} \end{cases}$$

The Boolean connectives used in the function are $\wedge$ (*AND*), $\vee$ (*OR*), and $\neg$ (*NOT*), and each basis predicate is of one of the following three forms:

<object value> <op> <constant>
<object value> <op> <object value>
<object value> <op> <object value> + <constant>

where *op* is in the set $\{=, \neq, <, >, \leq, \geq\}$.

Using the predicates associated with two classes, we may compute whether one is subsumed by the other. The class Chevycolors is subsumed by Colors, for example, *iff* all chevycolor objects are also color objects, or the following function, g, is unsatisfiable:

$$g(\sigma): \; f_{Chevycolors}(\sigma) \wedge \neg f_{Colors}(\sigma)$$

The satisfiability problem in the predicate calculus is in general unsolvable [LP]. However, when the basis predicates are defined as above for a fixed number of properties, satisfiability may be decided by a polynomial time algorithm [HR]. Thus, the system is able to prevent the definition of any type that violates the model of inheritance.

When a type is defined, the system constructs a function describing the type's class and a separate function for each of its immediate supertypes and subtypes; it then tests the type's class for containment by its superclasses and of its subclasses. Each function is a conjunction of the *constraints* defined on instances by a type and its supertypes. Constraints in a type definition are manifest as the implicitly or explicitly defined classes that are declared for the properties and operations and as *interproperty constraints.* Thus, the following function describes the set of valid instances of a new type *Chevycolor*:

$$f_{Chevycolors}(\sigma): \begin{array}{lllll} f_{Props_1}(\rho_1) & \wedge & \cdots & \wedge & f_{Props_j}(\rho_j) \quad \wedge \\ f_{Ops_1}(\phi_1) & \wedge & \cdots & \wedge & f_{Ops_k}(\phi_k) \quad \wedge \\ f_{Cnst_1}(\sigma) & \wedge & \cdots & \wedge & f_{Cnst_l}(\sigma), \end{array}$$

where ρ_i is a property object of σ and a member of the class $Props_i$ and ϕ_i is an operation object of σ and a member of the class Ops_i. If *Chevycolor* obeys the system's inheritance model, its definition is allowed. Henceforth, the function associated with the type's class is simply

$$f_{Chevycolors}(\sigma): \; \{type(\sigma) = Chevycolor\},$$

where $type(\sigma)$ is an operation defined by type Entity which maps an object σ to its associated type object.

Besides implicit definition with a type, classes are defined explicitly when a function is provided that describes the class' members. Explicitly defined classes generally contain objects of different types or a subset of the objects of a particular type. For example, we may chose to define the class Chevycolors with the function

$$f_{Chevycolors}(\sigma): \\ \{type(\sigma) = Color \wedge (value(\sigma) = blue \vee value(\sigma) = red)\},$$

where $value(\sigma)$ is an operation defined by type Property that maps a property object to its associated value object. Interproperty constraints are expressed in type definitions with similar functions. Figure 4, for example, shows an equality constraint on the color and carpetcolor of chevy objects as $\{chevy.color = chevy.carpetcolor\}$, where the notation $\sigma.\rho$ stands for the value object of property object ρ of object σ and may be alternatively written as $value(\rho(\sigma))$.

3. The Problem

In this paper we explore the problem of changing type definitions in an environment of persistent objects. The effect of changing a type extends beyond the type object itself. We examine the problem by first enumerating the kinds of change that may occur in a type and then describing the effects on other objects in the database.

3.1. Changes in a Type

The kinds of change which may occur in a type-lattice may be enumerated as follows:

- add or delete a type
- move a type to a new position in the type-lattice
- modify a type
  - add or delete properties, operations, or constraints defined by a type
  - modify properties, operations, or constraints defined by a type
    (*i.e.* modify the function associated with the property, operation, or constraint)

3.2. Effect on Instances

A change in a type definition may affect instances of the type that exist in the database since they're defined by the type. In addition, instances of the type's subtypes may also be affected because of inheritance; instances of subtypes are also instances of the type.

There are several type definition changes in which extant instances are rendered illegal representations of the type. Information stored in the instances may be missing, garbled, or undefined according to the current type definition. For example, if the implementation defined by the type is changed by rearranging properties, objects created before the change will be unintelligible. If interproperty constraints are changed, objects in the database may contain illegal values. Moving a type to a different position in the type-lattice has the same effect as removing some properties (*i.e.* those of its former supertypes) while adding others (*i.e.* those of its new supertypes).

3.3. Effect on Users of Instances

A change in a type definition may affect programs that use objects of the type. Moreover, programs that use objects of its supertypes may also be affected since an object of a type may be used in the same context as that of its supertype; an object possesses the properties and operations of its supertypes.

A program manipulates an object via its interface, suppling and receiving values according to constraints defined by the object's type. When the interface defined by the type is changed, errors may occur when a program uses the object. Specifically, a program may fail if a property or operation is no longer defined or if a value is outside the constraints defined by the type or expected by the program.

3.3.1. Property or Operation not Defined

Types may be changed by the addition or deletion of properties and operations. In the case of addition, programs written according to the new interface may not work on objects of the type created before the change; the old objects are missing information. Conversely, if properties and operations are deleted, programs written according to the old interface may not work on objects created after the change. For example, if a mileage property, epa_mpg, is added to the type Car, programs which use the epa_mpg value of car objects will fail on objects created before the addition of the property (Fig. 5).

A similar situation arises when a type is moved to a new position in the type lattice. Properties and operations formerly inherited from supertypes are replaced by those inherited from new supertypes. In addition, name conflicts may occur between the properties and operations provided by the old versus the new supertypes.

3.3.2. Value outside Constraints

A change in a type may strengthen a constraint resulting in refinement of a property or operation, or it may relax a constraint resulting in enrichment. Errors which occur in a program's use of an object after the constraints defined by its type have been modified may be categorized as a reader's problem or a writer's problem.

3.3.2.1. The Reader's Problem

A reader's problem may occur when a property value is read from an object. When a type change strengthens a constraint, the domain of values a property may assume becomes narrower; a program written according to the new type definition may read an unknown value from the property of an object created before the constraint change. Conversely, when a constraint is relaxed, the domain becomes wider; an old program may read an unknown value from an object created from the new type definition.

3.3.2.2. The Writer's Problem

A writer's problem may occur when a property value is written to an object. If a constraint is relaxed by a type change, programs using the new type definition may attempt to write an illegal value to an object created before the change. Conversely, if the constraint is strengthened, old programs applied to new objects fail if they attempt to write a value which is outside the new domain. For example, reader's and writer's problems may occur if the constraint on the color property in the type Car is strengthened (Fig. 6).

A writer's problem may also result from an object's use in a program that expects an instance of one of the object's supertypes, when the type of the object has refined a property or operation originally defined by the supertype. The situation is analogous to the application of an old program to a newly created instance of a type whose recent change involved a strengthening of constraints. In both cases the program expects a wider domain than that defined by the type of the object. A difference between the cases, though, is that refinement by a subtype will not result in a reader's problem since an object is never used in place of an object of a subtype.

4. The Proposed Solution

The foregoing discussion suggests that there are two groups of database objects that are affected when a type definition is changed. Objects of the type and its subtypes are dependent on the type definition for their implementation and interpretation. Objects such as application programs that use objects of the type and its supertypes are dependent on the interface defined by the type. Any solution to the problem of changing types necessarily includes provisions for both groups of dependent objects.

4.1. Solution for Instances

If a type is changed then in order for instances in the database to remain meaningful, either the instances must be *coerced* to the new definition or a new *version* of the type must be created, leaving the old version intact.

4.1.1. Coercion

The system provides an operation defined by Type that converts an instance of one type into that of another. The operation

convert (new_type : Types, instance : Entities)

returns *instance* as an object of *new_type* after any reconfiguration necessitated by a different set of properties or supertypes.

Coercion as the sole mechanism for dealing with type definition changes is limited and has several disadvantages. During the reconfiguration of instances, properties and the information they contain are discarded if not present in the new type. Values for properties which are defined in the new type may not be available for instances of the old type. Indeed, the properties themselves may be meaningless in old instances. Furthermore, the use of coercion alone is not extensible to the problem of dealing with programs that use objects of the type. The new type bears no connection to the old type, and their instances are not interchangeable.

4.1.2. Versions

A more flexible mechanism for dealing with type definition changes is the version set. A version set is an ordered collection of all incarnations of a particular object. A version set is initialized with the first definition of a type, is expanded by a new version with each modification of the type, and is terminated when the type is deleted. Depending on the semantics of the operation *delete_type*, the version set may be removed from the database (*i.e.* delete type and all its instances) or merely rendered non-modifiable and non-instantiable (*i.e.* delete type only). Type versions are treated analogously by the operation *delete_type_version*. Type definitions must remain in the database as long as their instances do.

Operations for version sets support random or sequential access of their version members; one member of each set is distinguished as *current-version*. A new current-version is added to a type's version set when the type is modified. Modification of a type involves choosing a non-deleted version, creating a new version from the chosen version, and adding the new version to the version set. New versions of subtypes of the chosen version are created automatically and are added to their respective version sets; new versions of subtypes differ from old only by virtue of the supertype version referenced. All versions of a type remain instantiable and modifiable until deleted. The linear order of versions within the set is established chronologically, although the version(s) from which a version was derived may also be retrieved.

An object is bound throughout its life to a single version of a type; its properties and operations are defined by that one particular version of its type. When a type is instantiated, the user may specify a version or choose the default, current-version.

Coercion may be used in conjunction with version sets to manage type definition changes. It may be desirable to convert instances of one version into those of another version of the same type. For example, a property which is added to a type may be relevant to instances of older versions of the type; instances of those versions may be explicitly updated with the new property by converting them into instances of the new version.

4.2. Solution for Users of Instances

Interobject errors which occur during a program's use of a changed type's object may be detected by either the program or the object. For example, when a program references a property or operation undefined in the object or when a program supplies a value to the object which is outside a constrained domain, the object detects the error. On the other hand, if an object returns a value which is outside the domain expected by the program, only the program can detect the error. In both cases, the errors occur because every object is strictly bound to a single version of its type. Moreover, different versions of the same type may define different interfaces for objects. If an abstraction of the interface defined by a type over time were available an object could be used as though it were bound to the type rather than to a single version of the type. As a result, objects of different versions of the same type could be used interchangeably by programs, provided names of properties and operations were unique and constant across all versions of the type.

As a solution we propose a *version set interface* with an *error handling* mechanism for interobject errors detected by objects of the changed type.

4.2.1. Version Set Interface

Conceptually, the version set interface of a type is an inclusive summary of all the type's versions. Every property and operation ever defined by a version of the type and every value ever declared valid for properties and operation parameters are represented in the interface. Properties and operations which do not appear in the interface are not valid on any version of the type. Similarly, values which are outside domains defined by the interface are not valid for any version.

More formally, a function F is associated with the version set interface which is a disjunction of functions, each of which describes a version in the set. Each version function is the conjunction of functions associated with the properties, operations, and constraints defined by the version. We may define the interface function for type T as follows:

$$F_{Tclass}(\sigma) : \quad f_{Tclass_1}(\sigma) \vee f_{Tclass_2}(\sigma) \vee \cdots \vee f_{Tclass_n}(\sigma),$$

where $f_{Tclass_i}(\sigma)$ is the functional description of the class defined by the ith version of type T. $F_{Tclass}(\sigma)$ thus describes the set of all objects that are of type T.

A version set interface is initialized to a description of the first version of a type. Subsequently the interface is modified whenever a new type version defines a property or operation that's not defined on other versions or that relaxes a constraint on the domain of a property or operation. Modification of the interface includes adding the new version function as a disjunct to the existing interface function. Because the interface represents the union of properties and operations defined by all versions of the type, where each is represented with its least constrained domain, modification of the interface does not occur when a new version strengthens a constraint or removes a property or operation.

The significance of the version set interface is that it establishes a standard interface for all instances of a type, regardless of their version. If a program which reads values from objects sets the expected domains to be those defined by the version set interface rather than the (possibly narrower) domains defined by a single version, the program will not fail on reading from an object of any version of the type. Similarly, if a type version were to carry definitions for all properties and operations in the interface and were to consider legal all values defined in the interface, errors such as the writer's problem and undefined property would not occur when its instances were used. However, a greater level of discrimination is usually necessary in programs and in versions of types, and therefore errors may result from the difference between the version set interface and the interface defined by a single version of a type. The following two mechanisms provide a sieve wherein certain error conditions may be caught and diverted.

4.2.2. Solution for Readers

When a new version of a type defines a less constrained domain for a property than that presently defined by the version set interface, the interface is modified and timestamped. In such a case, instances of the new version may contain values outside the previously defined domain, and programs which had successfully manipulated objects of all versions of the type may fail on reading values from instances of the new version.

In order to cope with this problem, we define an additional set of handlers on each type. These handlers will attempt to resolve the case in which a program tries to read a value from an instance of the type, and that type might provide a value that the program was not expecting.

We define two version set interfaces, one which is the union of all defined behavior, as before, and a new one that is the intersection of behavior defined on all types. We will call the first interface, the *union interface* and the second the *intersection interface*. The intersection interface represents the most constrained type. All versions of the type support this behavior.

The new set of handlers for a type version T are defined for all values that are defined for T but that are not in the intersection interface. That is, there is a handler for any piece of behavior of T that is not defined everywhere.

The use of these read handlers is very similar to that of the write handlers that were defined previously. Since they are a symmetric concept, we will not go into any more detail here. For simplicity, the rest of the examples will not refer to the read handlers.

4.2.3. Error Conditions and Handlers

An error condition is raised by an object when a program attempts to write a value to the object which is outside the defined domain or when a program references an undefined property or operation. The error conditions are *invalid* and *undefined*, respectively; handlers may be included in a type definition for both cases.

The use and substance of error handlers may be explained by first envisioning a comparison between the version set interface and the interface defined by a single version of a type. We attempt to establish a point to point connection between the two interfaces for each domain value of the properties and operations defined in the version set interface (Fig. 7). Each connector represents the means by which the value is read from or written to an object of the version. Connectors which are not fixed at the single version interface correspond to *undefined* or *invalid* error conditions which occur when use of the connector by a program is attempted. Error handlers may be defined with which loose connectors may be affixed to a single version interface. Handlers may consist of an executable routine or merely a more complete error diagnostic. They are evaluated in the same lexical scope as the operation code for the type on which they're defined.

The handlers required by a new type version over and above those required by the preceding version correspond to properties and operations that have been removed and domain values that are no longer considered valid by the new version. On the other hand, the definition of a new version may entail the addition

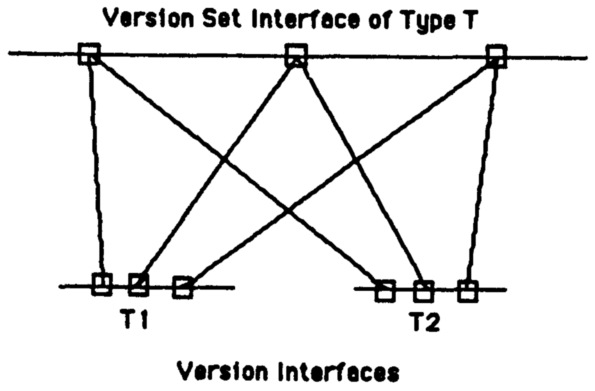

Figure 7. The correspondence between the version set interface of type *T* and the interfaces defined by two versions. The interface points marked by a square (□) represent constraints on three properties or operations defined by the type.

of error handlers to former versions for any properties, operations, and domain values introduced by the new version. That is, if the version set interface is modified

by the definition of a new version, error handlers will be required by former versions of the type.

The reader will remember that objects inherit the behavior defined by their supertypes. Further, a type may refine the inherited behavior. As a logical extension of this notion, error handlers are inherited as well, and they may be refined. Consequently, handlers resulting from a change in a type need be specified only for versions of the type. Subtypes either inherit a handler as is, or they may refine it. An exception occurs, however, when a new type is defined with subtypes or when a type with subtypes is deleted. The properties, operations, and domain values defined by a new type are not defined in existing versions of its subtypes. As a result, handlers are required by existing versions of the type's immediate subtypes, and the type becomes an *auxillary supertype* of those versions. A type inherits only error handlers from its auxillary supertypes. When a type with subtypes is deleted, its properties and operations may either be transferred to subtypes or be replaced in subtypes by error handlers. The type becomes an auxillary supertype for new subtype versions.

5. An Example

We will consider a small world of vehicles as an example domain. The properties defined by types in our example are reduced to a minimum so as to more clearly illustrate the ways in which the types are changed. Operations will include only those implicitly defined for reading and writing properties, named by the system as *read_propname* and *write_propname.*

We begin with the type Vehicle and three subtypes: Boeing aircraft, Honda automobiles, and Chevy automobiles. Vehicles have a single property, num_wheels which is drawn from the set of natural numbers. Hondas and chevys each refine the num_wheels property with the constraint that its value be four, and define a color property; the classes Chevycolors and Hondacolors are distinct and non-overlapping. In addition, chevys have a carpetcolor property (Fig. 8). Versions of types and classes are explicit in a type definition; a new version of a type's class results only from the generation of a new version of the type. A type version may specify only one version of each of its supertypes. In contrast, several versions of a subtype may specify the same type version as a supertype.

Because of the similarity between hondas and chevys a generalization, Car, is defined as a supertype. The definition of a new type is allowed by the system only if the class of the new type is contained by those of its supertypes and contains those of its subtypes. Following the definition of a new type, a version set and interface are created and initialized with the version; the interface is timestamped. If the new type defines properties and operations, they are added to the version set interface of each subtype. Moreover, error handlers are added to existing versions of subtypes for the new properties and operations, and the type becomes an auxillary supertype of those versions. A new version of each subtype is created. In our example, the new type Car assumes the refinement of num_wheels and defines a generalization of the color property. Honda and Chevy now refine the color property. The type Car has not as yet added any new properties to objects of its subtypes; its definition merely constitutes a reorganization of the type-lattice. The type Vehicle is modified by adding Car as a subtype; a new version of Vehicle is not created (Fig. 9).

The type Car is subsequently modified with the addition of two properties, epa_mpg and fuel. When properties and operations are added to a type, error handlers are added to existing versions of the type and the version set interface is updated. New versions are made of the type and its subtypes; version set interfaces of subtypes are updated. In our example, the property object for fuel may assume the values *leaded* or *nolead*; epa_mpg is a mileage value drawn from the set of natural numbers. Cars created before epa_mpg ratings will not have a value for epa_mpg; an error handler can do nothing more than return an error if the property is referenced. In contrast, it may be the case that before the fuel property was introduced, all cars burned leaded fuel. When the fuel property is read from an old object an error handler may return *leaded*; when written, a handler may return *valid* for a value of *leaded* and *invalid* for *nolead.*

The type Car is then modified by deleting the fuel property. When properties and operations are deleted, a new version of the type is created which contains error handlers for deleted properties and operations. New versions of the type's subtypes are created, and any refinements of the deleted properties and operations are removed from the new versions. In our example all cars now burn unleaded fuel, and the new version of Car receives appropriate error handlers (Fig. 10).

Changing a type by relaxing or strengthening a constraint on a property or operation results in a new version of the type, provided the class of the type is still contained by those of its supertypes and still contains those of its subtypes. If the constraint is relaxed the version set interface of the type is updated and timestamped. Error handlers are added to former versions of the type for those domain values defined by the new version but not by former versions. New versions of the type's subtypes are created; if a subtype doesn't define a refinement of the property or operation an update and timestamp is made to its version set interface. On the other hand, if the constraint is strengthened error handlers are added to the new version for domain values which are no longer valid. New versions of the type's subtypes are created. The constraints defined by our example type Car may be strengthened by changing the set from which color is drawn from Colors to Carcolors and for epa_mpg from Naturals to a set ranging from 20 to 50, where 20 may be interpreted as $<=$ 20 and 50 as $>=$ 50. We may include an error handler in the new version for epa_mpg but not for the color property. The handler for epa_mpg enters reasonable values which are outside the defined domain (*e.g.* within 10 mpg of the range limits) as either 20 or 50 and returns an error for others (Fig. 11).[3]

We may consider an example program called *car_task (car : Cars)* which manipulates car objects. The domains of the variables in *car_task()* match those defined by Car_4, a version that is the same as Car_3 except for a refinement of the color property from Colors to Carcolors. Figure 12 illustrates the results of executing the program on objects of different versions of Car. On receiving an error return from the object (*i.e. undefined, invalid*), the program may invoke an error handling routine of its own.

Finally, the type Chevy has been deleted from our example, leaving Honda as the only remaining subtype of Car. In this case, the type-lattice may be collapsed by deleting Car. When a type is deleted it remains in the database as long as its instances do, but it becomes non-modifiable and non-instantiable. New versions of its subtypes are created with the deleted type serving as an auxillary supertype. If the deleted type refined properties and operations originally defined by its supertype, the refinements are moved to the new version of each subtype unless the subtype already refines the property or operation. If subtypes refine properties and operations originally defined by the deleted type, the refinements are removed from the new versions. Error handlers are added to the new subtype versions for the deleted properties and operations. In our example, Honda re-acquires the refinement of num_wheels and defines an error handler for the property epa_mpg; all hondas have a mileage ≥ 50 (Fig. 13).

A change omitted from our example is the movement of a type within the type-lattice. We may, however, describe the change in terms of previously discussed changes. Movement within the type-lattice corresponds to removing former supertypes and adding new supertypes. Thus for each former supertype the type changes as though the supertype had been deleted; for each new supertype the type changes as though the supertype had been added.

6. Conclusions

We have presented a model for management of change in type definitions. Two groups of objects are affected by a change in a type: objects of the type and its subtypes and program objects which use objects of the type and its supertypes. As a solution for the former group we propose a version set mechanism wherein changes in a type result in new versions of the type and its subtypes. An instance remains bound to a particular version of its type unless explicitly coerced into another version. Properties, operations, and domain values defined on an object consist solely of those defined by the specific versions of the type and supertypes of which the object is an instance.

The mechanisms proposed for the latter group of objects facilitate the interchangeability of instances of different versions of the same type, and use an abstraction of a type over time, the version set interface. The interface represents the union of properties, operations, and domain values defined by all the versions of a type. The problem here is manifest as interobject errors which occur when program objects use objects of a type which has been changed. Errors may be detected by the program when it reads an unexpected value from an object. Alternatively, errors may be detected by the object when the program refers to an undefined property or operation or when the program attempts to write a value to an object which is outside a defined domain. For errors detected by objects of the changed type we propose a mechanism wherein error handlers may be added to versions of a type for properties, operations, and domain values defined in the version set interface but not in the interface defined by the version.

Finally, we presented a constraint language with which classes may be described as Boolean functions that test membership in the class. The class defined by a type version is described by the conjunction of the functions associated with the properties, operations, and interproperty constraints defined by the version. The class defined by a type version set is described by the disjunction of functions associated with its version members. The functional description of a type version's class enables the system to disallow the definition of a new version whose class is not contained by the classes of its supertypes or does not contain those of its subtypes. The functional description of a type version set enables the system to describe the set of properties, operations, and domain values that are defined in the version set interface but not in the interface defined by a version; the version may then define error handlers to cover that set.

The proposed model provides a systematic method for changing type definitions such that minimal impact is felt by objects which are dependent on the type. The system has some limitations in its ability to divert interobject errors primarily because we have adopted an unrestricted view of changes in type definitions. It may not be possible for a program object to accept as valid all domain values defined for an object by its type, or for a type to define error handlers to cover all changes in properties, operations, and domain values. It may be simply meaningless to consider some properties defined on one version as extensible to another. Changes in types may be as radical as the conversion of a frog to a prince, with very few characteristics common between two arbitrary versions of a type. In general, though, we feel that most courses of development will involve relatively minor alterations in type definitions, and for these the model provides support.

The modification of an arbitrary type-lattice could become quite complex. For this reason, a system is currently under development to assist type designers in definition and modification of types. We envision the facilities of the type designer's workbench to include a

[3]Although we have generated two versions of Car in this example, multiple constraints may be changed at the same time so that only a single new version results.

syntax for definition and for change (*e.g.* operations such as merging or splitting types or moving properties from one type to another as demonstrated in our example during the creation and deletion of the type Car) and a disambiguation scheme for property and operation names (*e.g.* when a type is moved moved from one location in the type-lattice to another a name conflict occurs if the names of properties and operations defined by former supertypes are the same as those defined by new supertypes.

References

[ABBHS] M. Ahlsen, A. Bjornerstedt, S. Britts, C. Hulten, L. Soderlund, "Making Type Changes Transparent", University of Stockholm, SYSLAB Report No. 22, February, 1984.

[At] M.P. Atkinson, P. Bailey, W.P. Cockshott, K.J. Chisholm, R. Morrison, "Progress with Persistent Programming" in P.M. Stoker, P.M.D. Gray, M.P. Atkinson (editors), "Databases - Role and Structure", Cambridge University Press, Cambridge, UK, 1984.

[BN] H. Biller, E.J. Neuhold, "Semantics of Databases: The Semantics of Data Models", Inf. Syst. 3 (1978), 11-30.

[BS] D.G. Bobrow, M. Stefik, "The LOOPS Manual", Xerox Corporation, 1983.

[Bu] P. Buneman, "Can We Reconcile Programming Languages and Databases?", in P.M. Stoker, P.M.D. Gray, M.P. Atkinson (editors), "Databases - Role and Structure", Cambridge University Press, Cambridge, UK, 1984.

[Ch] P.P.S. Chen, "The Entity-Relationship Model: Towards a Unified View of Data", ACM TODS 1, 1, March 1976.

[CM] G. Copeland and D. Maier, "Making Smalltalk a Database System", Proceedings of the ACM SIGMOD, Boston, Mass., June, 1984.

[Co] E.F. Codd, "Extending the Database Relational Model to Capture More Meaning". ACM Transactions on Database Systems 4, 4 (December 1979), 397-434.

[Fl] "Introduction to the Flavor System", in Reference Guide to Symbolics-Lisp, Symbolics Inc., 1985.

[GAL] A. Albano, L. Cardelli, and R. Orsini, "Galileo: A Strongly Typed Interactive Conceptual Language", Technical Report 83-11271-2, Bell Laboratories, Murray Hill, New Jersey, July, 1983.

[GR] A. Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[HM] M. Hammer, D. McLeod, "Database Description with SDM: A Semantic Database Model", ACM TODS 6, 3, September 1981, 351-387.

[HR] H.B. Hunt, D.J. Rosenkrantz, "The Complexity of Testing Predicate Locks", Proceedings of the ACM SIGMOD, Boston, Mass., May-June, 1979.

[KL] R. Katz and T. Lehman, "Storage Structures for Versions and Alternatives", Computer Science Department, University of Wisconsin - Madison, Technical Report No. 479, July, 1982.

[LP] H.R. Lewis, C.H. Papadimitriou, "Elements of the Theory of Computation", Prentice-Hall, 1981.

[LDEGPWWW] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, J. Woodfill, "Designing DBMS Support for the Temporal Dimension", Proceedings of the ACM SIGMOD, Boston, Mass., June, 1984.

[MBW] J. Mylopoulos, P.A. Bernstein, H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications", ACM Transactions on Database Systems, Vol 5, No. 2, June, 1980, pages 185-207.

[PSA] Persistent Programming Research Group, "PS-Algol Reference Manual", University of St. Andrews, Department of Computational Science, Persistent Programming Research Report 12.

[Sh] D.W. Shipman, "The Functional Data Model and the Data Language DAPLEX", ACM TODS 6, 1 (1981), 140-173.

[Sc] J.W. Schmidt, "Type Concepts for Database Definition", in Schneiderman, B. (editor), Databases: Improving Usability and Responsiveness, Academic Press, 1978.

[SFL] J.M. Smith, S. Fox, and T. Landers, "ADAPLEX: Rational and Reference Manual", second edition, Computer Corporation of America, Cambridge, Mass., 1983.

[SK] A. Shepherd and L. Kerschberg, "PRISM: A Knowledge Based System for Semantic Integrity Specification and Enforcement in Database Systems", Proceedings of the ACM SIGMOD, Boston, Mass., June, 1984.

[SS] J.M. Smith, D.C.P. Smith, "Database Abstractions: Aggregation", CACM 20, 6 (1977).

[TL] D.C. Tsichritzis, F.H. Lochovsky, "Data Models", Prentice-Hall, 1982.

[WM] H.K.T. Wong, J. Mylopoulos, "Two Views of Data Semantics: A Survey

[WMy] H.K.T. Wong, J. Mylopoulos, "Two Views of Data Semantics: A Survey of Data Models in Artificial Intelligence and Database Management", INFOR 15, 3 (1977), 344-382.

[Zd] S.B. Zdonik, "Object Mangement System Concepts", Proceedings of the Second ACM-SIGOA Conference on Office Information Systems, Toronto, Canada, June, 1984.

[ZW] S.B. Zdonik and P. Wegner, "Language and Methodology for Object-Oriented Database Environments", Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, January, 1986.

Type Definitions

```
define type Chevy
  property definitions
    refine color as
      color : Chevycolors†
    carpetcolor : Chevycolors‡
  operation definitions
    refine readcolor as
      readcolor (car : Chevys, result : Chevycolors)†
    readcarpetcolor
      (car : Chevys, result : Chevycolors)‡
  supertypes
    Car
  subtypes
    —
```

```
define type Car
  property definitions
    color : Colors
  operation definitions
    readcolor (car : Cars, result : Colors)
  supertypes
    Entity
  subtypes
    Chevy
```

Figure 3. A type and its supertype, where the type has both refined (†) and added to (‡) the properties and operations defined by the supertype.

Type Definition

```
define type Chevy
  property definitions
    refine color as
      color : {type(color) = Color  ∧  (value(color) = blue  ∨ value(color) = red)}†
    carpetcolor : {type(color) = Color  ∧  (value(color) = blue  ∨ value(color) = red)}†
  operation definitions
    .
    .
    .
  constraints
    {chevy.color = chevy.carpetcolor}‡
  supertypes
    Car
  subtypes
    —
```

Figure 4. Type definitions may constrain the domain of a property by declaring a Boolean function rather than a class name (†). Interproperty constraints may be also declared in a type definition (‡).

Type Definition

```
define type Car
  property definitions
    color : Colors
    epa_mpg : Naturals
  operation definitions
    .
    .
    .
  supertypes
    Entity
  subtypes
    Chevy
```

Undefined Property

```
compare_cars (c1 : Cars, c2 : Cars)
  m1, m2 : Naturals
  .
  .
  .
  m1 := c1.epa_mpg
  m2 := c2.epa_mpg
  .
  .
  .
```

Figure 5. The type Car is changed by adding the property epa_mpg. The function compare_cars will fail if either of the cars passed as arguments was created before Car was changed.

Type Definition	Reader's Problem	Writer's Problem
define type Car		
property definitions	copy_car_description (c : Cars)	paint_car (c : Cars, p : Colors)
color : Carcolors	color : Carcolors	.
epa_mpg : Naturals	m : Naturals	.
operation definitions	.	.
.	.	c.color := p
.	.	.
.	color := c.color	.
supertypes	m := c.epa_mpg	.
Entity	.	
subtypes	.	
Chevy	.	

Figure 6. The type Car is changed by strengthening the constraint on the color property from Colors to Carcolors, a subclass of Colors. Programs written after the change may experience a reader's problem if their object arguments were created before the change. Programs written before the change may experience a writer's problem with newly created object arguments.

Type Definitions		
define type Vehicle$_1$	**define type** Honda$_1$	**define type** Chevy$_1$
property definitions	**property definitions**	**property definitions**
num_wheels : Naturals$_1$	**refine** num_wheels **as**	**refine** num_wheels **as**
supertypes	num_wheels : 4	num_wheels : 4
Entity$_1$	color : Hondacolors$_1$	color : Chevycolors$_1$
subtypes	**supertypes**	carpetcolor : Chevycolors$_1$
Boeing, Honda, Chevy	Vehicle$_1$	**supertypes**
	subtypes	Vehicle$_1$
	—	**subtypes**
		—

Figure 8. Type definitions for the first versions of Vehicle, Honda, and Chevy.

Type Definitions		
define type Car$_1$	**define type** Chevy$_1$	**define type** Chevy$_2$
property definitions	**property definitions**	**property definitions**
refine num_wheels **as**	**refine** num_wheels **as**	**refine** color **as**
num_wheels : 4	num_wheels : 4	color : Chevycolors$_1$
color : Colors$_1$	color : Chevycolors$_1$	carpetcolor : Chevycolors$_1$
supertypes	carpetcolor : Chevycolors$_1$	**supertypes**
Vehicle$_1$	**supertypes**	Car$_1$
subtypes	Vehicle$_1$	**subtypes**
Honda, Chevy	**auxiliary supertypes**	—
	Car$_1$	
	subtypes	
	—	

Figure 9. Type definitions for Car and two versions of Chevy.

Type Definitions

```
define type $Car_1$
  property definitions
    refine num_wheels as
      num_wheels : 4
    color : $Colors_1$
  handlers
    for undefined read_fuel
      return leaded
    for undefined write_fuel
      if value(fuel) = leaded
        return valid
      else return invalid
  supertypes
    $Vehicle_1$
  subtypes
    Honda, Chevy
```

```
define type $Car_2$
  property definitions
    refine num_wheels as
      num_wheels : 4
    color : $Colors_1$
    epa_mpg : $Naturals_1$
    fuel : $Fueltypes_1$
  supertypes
    $Vehicle_1$
  subtypes
    Honda, Chevy
```

```
define type $Car_3$
  property definitions
    refine num_wheels as
      num_wheels : 4
    color : $Colors_1$
    epa_mpg : $Naturals_1$
  handlers
    for undefined read_fuel
      return nolead
    for undefined write_fuel
      if value(fuel) = nolead
        return valid
      else return invalid
  supertypes
    $Vehicle_1$
  subtypes
    Honda, Chevy
```

Figure 10. The versions of Car resulting from a sequence of first adding the properties epa_mpg and fuel and then removing fuel.

Type Definition

```
define type $Car_5$
  property definitions
    refine num_wheels as
      num_wheels : 4
    color : $Carcolors_1$
    epa_mpg : $\{type(epa\_mpg) = Natural \land value(epa\_mpg) \geq 20 \land value(epa\_mpg) \leq 50\}$
  handlers
    for undefined read_fuel
      return nolead
    for undefined write_fuel
      if value(fuel) = nolead
        return valid
      else return invalid
    for invalid write_epa_mpg
      if $value(epa\_mpg) < 20 \land value(epa\_mpg) \geq 10$
        car.epa_mpg := 20
      else if $value(epa\_mpg) > 50 \land value(epa\_mpg) \leq 60$
        car.epa_mpg := 50
      else return invalid
  supertypes
    $Vehicle_1$
  subtypes
    Honda, Chevy
```

Figure 11. The version of Car resulting from strengthening constraints first on the property color and then on epa_mpg.

Program	Object Response				
	car$_1$	car$_2$	car$_3$	car$_4$	car$_5$
car_task (car : Cars)					
fuel : Fueltypes					
cc : Carcolors					
mpg : Naturals					
num : Naturals					
fuel := car.fuel	leaded	leaded/nolead	nolead	nolead	nolead
cc := car.color	color†	color†	color†	carcolor	carcolor
if cc = red					
mpg := 5					
car.epa_mpg := mpg	undefined	valid	valid	valid	invalid
else					
mpg := 55					
car.epa_mpg := mpg	undefined	valid	valid	valid	valid‡
num := 1					
car.num_wheels := num	invalid	invalid	invalid	invalid	invalid

Figure 12. The program *car_task()* as executed on objects of different versions of Car. †value may be outside the class Carcolors. ‡car.epa_mpg actually receives the value 50.

Type Definitions	
define type Honda$_6$	**define type** Honda$_7$
property definitions	**property definitions**
refine color **as**	**refine** num_wheels **as**
color : Hondacolors$_1$	num_wheels : 4
supertypes	color : Hondacolors$_1$
Car$_5$	**handlers**
subtypes	**for undefined** read_epa_mpg
—	return 50
	for undefined write_epa_mpg
	if value(epa_mpg) $\geq$ 50
	return valid
	else return invalid
	supertypes
	Vehicle$_1$
	auxiliary supertypes
	Car$_5$
	subtypes
	—

Figure 13. Two versions of Honda illustrating the effect of removing the supertype Car.