Change Propagation in an Axiomatic Model of Schema Evolution for Objectbase Management Systems

Randal J. Peters*[1] and Ken Barker**[2]

[1] University of Manitoba
Department of Computer Science
Winnipeg, Manitoba, Canada
randal@cs.umanitoba.ca

[2] University of Calgary
Department of Computer Science
Calgary, Alberta, Canada
barker@cpsc.ucalgary.ca

Abstract. Schema evolution is an important component of advanced information systems such as *objectbase management systems*. These systems typically support volatile and complex application domains that include engineering design, CAD/CAM, multimedia, and geo-information systems. The schema of these applications must be able to evolve along with the changing environment. There are two problems to consider in schema evolution: (i) *semantics of change* and (ii) *change propagation*. The first deals with the effects of the schema change on the overall type system. For example, the deletion of a property in a type affects the subtypes inheriting that property. Our previous work has introduced a sound and complete *axiomatic model* to deal with the semantics of change problem. The second problem deals with the techniques for propagating schema changes to the underlying objects. For example, the addition of an attribute to a type requires additional memory to be allocated to the objects so that values for the attribute may be stored. The first step of change propagation is to identify the affected objects. Subsequent steps carry out the actual changes. This paper deals with the first step by extending the axiomatic model with semantics to determine a sound and complete set of objects affected by a schema change. The extended model can be used with any method for carrying out the changes such as the *conversion*, *screening*, and *filtering* approaches proposed in the literature.

* The Natural Science and Engineering Research Council (NSERC) of Canada support this research under operating grant OGP-0173259.

** The Natural Science and Engineering Research Council (NSERC) of Canada supports this research under research grant RGP-0105566.

H. Balsters, B. de Brock, and S. Conrad (Eds.): FoMLaDO/DEMM 2000, LNCS 2065, pp. 142–162, 2001.

1 Introduction

Designers and users of advanced application environments realize the benefits that schema evolution can provide. The main feature is to allow the modification of a design "on the fly." The main requirement is that there is a clear semantics carried out by each schema change. Support for schema evolution is important when advanced information servers such as *objectbase management systems* (OBMSs) are used to develop and run these applications.

Object-oriented computing is emerging as the predominant technology for providing database services in advanced application domains such as engineering design, CAD/CAM systems, multimedia, and geo-information systems, to name a few. A distinguishing characteristic of these applications is that the schema design can become quite complex with many types and inheritance links between them. An important feature to support in these systems is the ability to modify the schema as the application environment evolves. For example, in an engineering design application many components of an overall design may go through several modifications to produce a final product. Dynamic schema evolution within an OBMS can support these requirements.

Table 1. Typical schema change operations.

	Add (A)	Drop (D)	Modify (M)
Type (T)	Type addition	Type deletion	Add Behavior (AB) Drop Behavior (DB) Add Subtype Relationship (ASR) Drop Subtype Relationship (DSR)

Three basic operations are typically performed during schema evolution: *add*, *drop* and *modify*. Table 1 shows the combinations of applying these operations to types. The result is a collection of six complete operations that are performed on types during schema evolution. We use the conjunction of the abbreviations to denote the operations. For example, AT is an "Add Type" operation and MT-DSR is a "Modify Type - Drop Subtype Relationship" operation. A typical schema change affects many aspects of a system. There are two fundamental problems to consider:

Semantics of Change: The effects of the change on the overall way in which the system organizes information (i.e., the effects on the schema), and

Change Propagation: The effects of the change on the consistency of the underlying objects (i.e., the identification of affected objects and the propagation of the changes to these instances).

For the first problem, the basic approach is to define a number of invariants that must be satisfied by the schema and then define rules and procedures for maintaining these invariants for each possible schema change. Orion [1] and

Gemstone [9] are examples of OBMSs that use this approach. Our approach [12] introduces a *formal axiomatic model* for handling the semantics of change. From the schema designer perspective, it is a simple model to use because the designer only needs to specify and maintain two sets for each type: the *essential supertypes* and *essential properties*. The axioms provide an automated means of managing schema changes based on modifications to these two sets. An important characteristic of the model is that it is proven sound and complete.

For the second problem, the objects affected by a schema change must be identified and then the changes must be carried out. *The main contribution of this paper is an extension to the axiomatic model for identifying objects affected by a schema change*. In keeping with the characteristics of the axiomatic model, this extension has a proven soundness and completeness as well. The result of this work can serve as a "front-end" to any of the proposed techniques for carrying out schema changes. A typical technique is to explicitly *coerce* objects to coincide with the new definition of the schema. *Screening* and *conversion* are two approaches for defining when coercion actually takes place. Conversion (e.g., Orion [1]) stops the system and updates the affected objects immediately after a schema change. Screening (e.g., GemStone [9]) coerces objects when they are first accessed after a schema change (i.e., the system is not stopped to update objects). Sometimes a versioning mechanism is used in conjunction with coercion and old representations of objects are maintained. *Filtering* [14] is a change propagation technique based on a versioning mechanism that maintains older versions of updated objects. The purpose is to provide better compatibility between objects as the schema evolves.

The relationships between the various components regarding the axiomatic model are shown in Figure 1. The *semantics of change* component includes schema modifications by the designer as listed in Table 1 followed by a precise semantics for incorporating these changes at the schema level. This part of the model has been shown to encompass the schema evolution operations of several OBMSs including Orion, Gemstone, O2, and Tigukat. The *change propagation* component identifies the objects affected by the schema change and then carries out the changes by coercing the objects. Object identification is addressed in this paper and can be linked to the various approaches for carrying out changes.

The remainder of the paper is organized as follows. Section 2 gives an overview of the axiomatic object model and its uses in the semantics of change problem of schema evolution. The axiomatic model is extended in Section 3 to develop a change propagation model and form a complete axiomatic model for schema evolution. Other work related to schema evolution and the axiomatic model is outlined in Section 4. Finally, conclusions and future research are given in Section 5.

2 Axiomatic Model Overview

This section gives an overview of an axiomatic model for specifying properties and inheritance structures of types in an object-oriented environment. The model

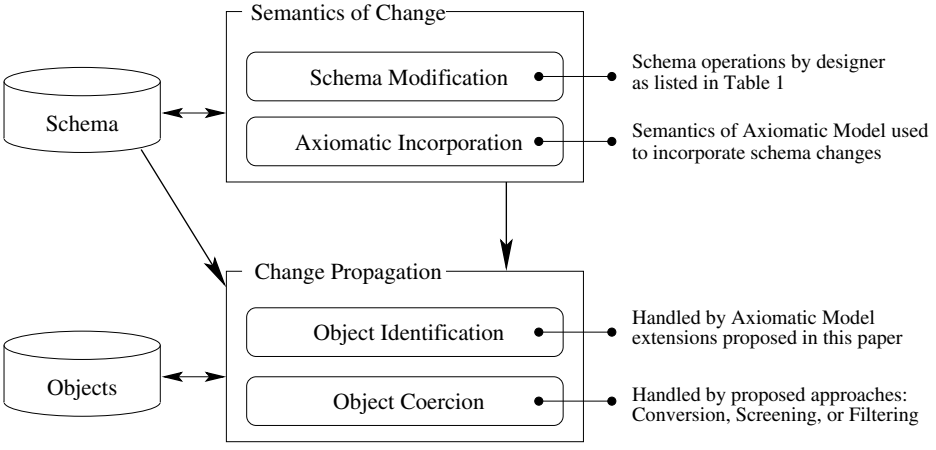

Fig. 1. Schema Evolution Component Relationships of the Axiomatic Model

has been used as a solution to the semantics of change problem in OBMSs [11,12]. It also serves as a basis for a methodology of schema integration in federated objectbase systems [2] and for defining schema evolution in real-time object-oriented database environments [16]. Details of the axiomatic model are given in [11,12]. This section focuses on the notation of the model since they are used in the change propagation extensions presented in Section 3.

A *type* in an object model (called a *class* in some models) defines properties of objects. Existing systems use attributes, methods, and behaviors to represent properties of objects. We use the term *property* to generically encompass all of these components. Types are used as templates for creating objects. The set of all objects created from a particular type is called the *extent* of that type.

Subtyping is a facility of object models that allows types to be built incrementally from other types. We use the symbol "$\preceq$" to represent a reflexive, transitive, and antisymmetric *subtype relationship* where $t \preceq s$ means that type t is a subtype of type s, or equivalently, s is a supertype of t. Diagrammatically, we use a directed arrow from a subtype (the tail) to its supertype (the head) to represent a subtype relationship. A *subtype* inherits all the properties of its supertype and can define additional properties that do not exist in the supertype. If a subtype has multiple supertypes, it inherits the properties of all the supertypes. This is known as *multiple inheritance* and results in a graph of subtype relationships.

A *type lattice* (or simply *lattice*) $L = \langle T, \leq \rangle$ consists of a set of types, T, together with a partial order, $\leq$, of the elements of T based on the subtype relationship ($\preceq$). The term *lattice* (or *semi-lattice*) is commonly used in object-oriented literature to denote a typing structure that supports multiple inheritance. This meaning does not correspond to lattice in the strict mathematical sense because the notions of least upper bound and greatest lower bound are relaxed. Regardless, we use the term throughout the paper with the understanding that its object-oriented meaning applies. A type lattice can be represented as a

directed acyclic graph (DAG) with types as vertices and subtype relationships as directed edges.

The notation for the axiomatic model is shown in Table reftable2. The terms denote various arrangements of types and properties that can be represented in virtually any object model. We address each of these terms and use the simple example type lattice in Figure 2 to clarify their semantics. The example is kept simple so that the functionality of the axiomatic model can be more easily presented and understood. It will become apparent from the following discussion that the model scales up to type lattices of more complex application environments such as those mentioned earlier.

Table 2. Notation for Axiomatic Model

Term	Description
T	The set of all types in an application schema design
L	The type lattice of an application schema design $L = \langle T, \leq \rangle$
$s, t, \nabla, \perp$	Type elements of T
$P(t)$	Immediate supertypes of type t
$P_e(t)$	Essential supertypes of type t
$PL(t)$	All supertypes of type t
L_t	Supertype lattice of type t
$N(t)$	Native properties of type t
$H(t)$	Inherited properties of type t
$N_e(t)$	Essential properties of type t
$I(t)$	Interface of type t
$\alpha_x(f, T^*)$	Apply-all operation

The set of types T represents all the types in an application schema design. These types have schema evolution operations applied to them. The set consisting of all types (i.e., vertices) in Figure 2 forms T in this example. A type lattice L is formed from the set T and the subtype relationships between the types of T. Type elements s and t serve as variables while ∇ and $\perp$ are constants denoting the least defined type and most defined type, respectively. In Figure 2, ∇ = T_object and $\perp$ =T_null. Type ∇ serves as a common ancestor of all types and $\perp$ serves as a common descendent. The use of ∇ is popular in many systems as a root with properties that are inherited by all types. For example, it can be used to support object identity or a set of typical comparison operators. The use of $\perp$, while not as widespread, can be favorable as a type that supports all properties and behaviors. One function is to create a number of "error" objects of this type that can then be returned by the methods of other types when errors occur. These error objects have some meaning with respect to the other methods in the system.

The *immediate supertypes*, $P(t)$, of a type t are those types that cannot be reached from t, transitively, through some other type. In other words, their only link to t is through a *direct* subtype relationship. For example, if we let t = **T_teachingAssistant**, then the immediate supertypes of t are **T_student** and **T_employee**. Hence, P(T_teachingAssistant)={T_student,T_employee}. The other supertypes of T_teachingAssistant (i.e., T_person, T_taxSource, and T_object) can be reached transitively through T_student or T_employee.

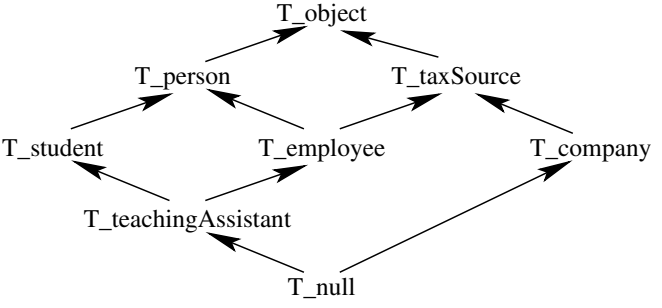

Fig. 2. Simple example type lattice

The *essential supertypes*, $P_e(t)$, are the types identified as being essential to the construction of type t. Essential supertypes must be maintained as supertypes of t for as long as consistently possible during the evolution of the schema. The only way to break a link from t to an essential supertype s is to explicitly remove s from $P_e(t)$ by either dropping the subtype relationship between t and s or by dropping s entirely. Note that $P(t) \subseteq P_e(t)$, which means that the immediate supertypes are essential.

An OBMS can impose constraints that force a newly created type to be a subtype of certain system primitive types. In other words, these primitive types are essential supertypes of every type. For example, many OBMSs define a primitive root type "object" that must be a supertype of all types, either directly or transitively through some other type. Upon creation of a new type t, the system can initialize $P_e(t)$ to {T_object}. In a typical environment, the system would provide essential supertypes based on known constraints and the schema designer would provide essential supertypes based on his/her expertise in the particular application domain being modeled.

In Figure 2, assume the system provides the root type **T_object** and assume the schema designer has specified the remaining essential supertypes of **T_teachingAssistant** resulting in the following:

$$P_e(T_teachingAssistant) = \{T_student, \quad T_employee, \quad T_person, \quad T_object\}$$

If T_student and T_employee were dropped as immediate supertypes of T_teachingAssistant, then T_person would be established as an immediate super-

type because it is essential. However, T_taxSource would be lost as a supertype because it is not declared as essential.

The *supertype lattice* $L_t = \langle PL(t), \leq_t \rangle$ of a type t consists of a set $PL(t)$ which includes t and all supertypes (immediate, essential, or otherwise) of t together with a partial order $\leq_t$ such that $\forall x, y \in PL(t)$ if $x \preceq y$ in [then $x \preceq y$ in $\leq_t$. In other words, if a subtyping relationship exists between supertypes of t in the application lattice, then the subtype relationship also exists in the supertype lattice of t. For example, if we let t=T_employee, then the supertype lattice of t is given as:

PL(T_employee) = {T_employee, T_person, T_taxSource, T_object}

$\leq_{employee}$ = {T_employee $\preceq$ T_person, T_employee $\preceq$ T_taxSource, T_person $\preceq$ T_object, T_taxSource $\preceq$ T_object}

The *native properties*, $N(t)$, of a type t are those properties that are not defined in any of the supertypes of t. That is, they are not inherited from a supertype, but instead are natively defined in t. Note that the native properties of one type may also be defined as native properties of other types that are not in a subtype relationship with one another. For example, the type T_employee may have a native "salary" property that is not defined on any of its supertypes. Moreover, T_person and T_taxSource may both have native "name" properties defined because they are not in a subtype relationship with one another.

The *inherited properties*, $H(t)$, of a type t is the union of the properties defined by all supertypes of t. The native and inherited properties are disjoint. For example, the inherited properties of T_employee are the union of the properties defined on T_person, T_taxSource, and T_object. In contrast, the native properties of T_employee are those defined on employees, but not defined on any of T_person, T_taxSource, or T_object.

When two common properties are inherited from multiple supertypes (e.g., T_employee inherits the "name" property from both T_person and T_taxSource) a *conflict* can arise and some form of *conflict resolution* must be performed. The conflict resolution problem has been addressed in previous work [12]. One element that must be resolved is the representation/implementation of the property. A simple form of resolution used in some systems is to ask the designer to resolve a conflict by choosing one of the conflicting properties as a basis for the representation/implementation or by redefining the property altogether.

The *interface*, $I(t)$, of a type t is the union of native and inherited properties of t. This term simply serves as a specification of all properties of t to which the object instances of t will respond.

The *essential properties*, $N_e(t)$, are those properties identified as being essential to the construction and existence of type t. Essential properties must be maintained as part of the definition of t for as long as consistently possible during the evolution of the schema. The essential properties of a type consist of all properties natively defined by the type (i.e., $N(t) \subseteq N_e(t)$) and may contain properties inherited from its supertypes. The schema designer has the expertise to understand the properties that types within a particular application domain must support and can declare these properties as being essential to the types by

including them in the appropriate $N_e(t)$ specification. Additionally, the system may require all types to support various primitive properties for object instances such as object identity retrieval and object equality. At type creation time, the system can initialize $N_e(t)$ with the appropriate primitive properties. The synergy between schema designer and system primitives goes hand in hand with the definition of essential properties $N_e(t)$ and essential supertypes $P_e(t)$.

Schema evolution may force inherited properties of a type to be adopted as native properties. This can occur if a type defines an essential property that is currently inherited from a supertype and that property is removed from the supertype or the supertype is removed altogether. For example, assume that a "taxBracket" property is defined on T_taxSource and this property is declared as essential in T_employee. If T_taxSource is deleted, then "taxBracket" would be adopted as a native property of T_employee because it is essential to that type. The axiomatic model automatically handles this adoption process.

We provide an *apply-all* operation in the axiomatic model. This operation, denoted $\alpha_x(f, T^*)$, applies the unary function f to the types $T^* \subseteq T$. The function f is defined over the single variable x, which is shown as the subscript of the α operator. Other variables appearing within the parenthesis of the α operation are substituted with their values prior to evaluation and they remain constant throughout the apply-all operation. The semantics of apply-all will let x range over the elements of T^* and for each type bound to x, f is evaluated and the answer is included in the final result set. If T^* is empty, the empty set is returned. In functional notation, the α operation applies the lambda function $\lambda x.f$to every element of T^* and returns a set containing the results. For example, the expression $\cup\alpha_x(N_e(x), \{T_person, T_student\})$ gives the set of native essential properties specified in T_person and T_student.

Table 3 depicts the axioms of dynamic schema evolution using the various types and properties in Table 2. The derivation of the various sets in the axioms are based on the $P_e(t)$ and $N_e(t)$ terms. To define a schema (i.e., type lattice), one need only specify values for these two sets. They can be initialized as part of type creation or modified during schema evolution. All schema operations are handled as modification to these two terms, which eases the burden on the schema designer and makes the system more manageable. The effects of schema changes on subtyping relationships and property inheritance must be closely scrutinized in order to maintain system integrity, as well as the intentions of the schema designer. The axiomatic model provides a consistent, automatic mechanism for deriving the entire type lattice structure after a change to either $P_e(t)$ or $N_e(t)$. Further, the model has proven soundness, completeness, and termination. The axiomatic model has the flexibility to handle variations on type and property arrangements depending on the defaults imposed by individual systems. This results in a powerful model that can be used to describe dynamic schema evolution in OBMSs that support subtyping and property inheritance. The axiomatization and comparison of Tigukat, Orion, GemStone, and O_2 have been addressed in previous work [2,12].

Table 3. Axioms of Dynamic Schema Evolution

Name	Axiom
Axiom of Closure	$\forall t \in T, P_e(t) \subseteq T$
Axiom of Acyclicity	$\forall t \in T, t \notin \cup\alpha_x(PL(x), P(t))$
Axiom of Rootedness	$\exists \nabla \in T, \forall t \in T, \nabla \in PL(t) \wedge P_e(\nabla) = \{\}$
Axiom of Pointedness	$\exists \perp \in T, \forall t \in T, t \in PL(\perp)$
Axiom of Direct Supertypes	$\forall t \in T, P(t) = P_e(t) - \cup\alpha_x(PL(x) \cap P_e(t) - \{x\}, P_e(t))$
Axiom of Supertype Lattice	$\forall t \in T, PL(t) = \cup\alpha_x(PL(x), P(t)) \cup \{t\}$
Axiom of Interface	$\forall t \in T, I(t) = N(t) \cup H(t)$
Axiom of Nativeness	$\forall t \in T, N(t) = N_e(t) - H(t)$
Axiom of Inheritance	$\forall t \in T, H(t) = \cup\alpha_x(I(x), P(t))$

The specification and management of P_e and N_e can be a shared responsibility between the system and the user. For example, when a new type is defined, the system may open a dialog with the schema designer to determine all supertypes and properties that are essential to the new type. Alternatively, the system may make a default assumption that all supertypes and properties (including inherited properties) are essential in a given type. Other configurations are possible as well. Current systems vary in the semantics defined for the notions of subtyping, inheritance, and nativeness. The formalization of these concepts into the axiomatic model gives a common basis that allows systems the flexibility to build their own customized notions on top of them, while remaining rooted at the formal model. The flexibility of the axiomatic model has been shown by extending it to support schema integration [2] and real-time database systems [16].

3 Change Propagation Model

This section develops the change propagation extensions to the axiomatic model described in Section 2. Additional notation is introduced along with a list of new axioms to support consistent change propagation. The first extension introduces a new notation for identifying the objects in the extent of the types. For each type t, there are three sets of objects associated with t denoted $E_0(t)$, $E_\infty(t)$, and $E_i(t)$. These sets correspond to the shallow extent, deep extent, and i^{th} level extent, respectively. The shallow extent of a type t is the set of objects created directly from t. The deep extent of t includes its shallow extent along with the shallow extent of all subtypes of t. The i^{th} level extent of t includes its shallow extent and the shallow extent of the subtypes up to depth i. Shallow and deep extents are well known concepts and are used extensively in OBMSs for object query models and query processing. The i^{th} level extent is included for completeness as it is useful in some aspects of objectbase systems [13].

Changes to the schema affect the structure of the types and the organization of the type lattice. This in turn affects the objects in the type extents. The first step in performing change propagation is to identify the affected objects. The subsequent steps carry out the changes to the objects by coercing them into structures that correspond to the evolved types. The literature reports three basic approaches for object coercion: *conversion* [9], *screening* [1], and *filtering* [14]. Conversion stops the system and updates the affected objects immediately after a schema change. This is a straightforward and simple approach, but can suffer from performance problems if many changes occur and the system is halted frequently. Screening coerces objects when they are first accessed after a schema change. The system does not have to be stopped to update objects and so concurrent operations can proceed with greater transparency. Filtering is based on a versioning mechanism that maintains older versions of updated objects. The approach requires more space overhead and greater processing demands since translations between older and newer versions may be required on a continuous basis. The purpose is to provide better compatibility between objects and their method implementations as the schema evolves.

The change propagation model presented in this paper is the first of its kind that clearly and formally identifies the objects affected by a schema change. The model can be adopted as a "front end" to any of the coercion approaches. The model is *complete* in the sense that it guarantees to determine all objects affected by a schema change. This is a minimal requirement of any change propagation mechanism. The model is *sound* in the sense that it only determines objects that are guaranteed to be affected by a schema change. In other words, it guarantees a minimal set of objects affected by a schema change. This translates into greater efficiency because some objects of certain subtypes may not have to be coerced. Other approaches are more liberal and may coerce objects that did not require an update. For example, if a type t is affected by a schema change, some systems simply coerce all the objects of t and its subtypes (i.e., the deep extent of t). There are cases where many objects in the deep extent are not affected by the change. Our model will automatically exclude these from being coerced. Another property of the model is that it only uses schema information to determine the objects affected by a schema change. It does not have to access the objects themselves. Finally, the model provides a precise semantics for identifying the objects affected in change propagation. Other schema evolution approaches give informal explanations of the change propagation operations. The axiomatic model can provide a formal basis for comparing the various techniques.

Two theorems regarding the completeness and soundness of the change propagation axioms are constructed. Only sketches of the proofs are given. The complete proofs are omitted due to page limitations.

Theorem 1. *The change propagation axioms are complete.*

Proof. By induction on *maximal paths* from the types affected by the schema evolution operation. A *maximal path* is the largest number of direct supertype links between two types. For example, in Figure 2 the maximal path between

T_teachingAssistant and T_person is two, and the maximal path between T_null and T_taxSource is three. The proof proceeds by showing the completeness of each axiom in Table 4. The base cases for the induction focus on the types involved as parameters of the schema change operation. The induction step is straightforward and shows that given the assumption that the axioms are complete for types with maximal path n, the axioms are complete for types with maximal path $n+1$. □

Theorem 2. *The change propagation axioms are sound.*

Proof. By induction on maximal paths from the types involved in the schema evolution operation. The proof is constructed similar to Theorem 1 above. □

Table 4. Notation for Axiomatic Model

Axiom Name	Axiom
Add Behavior	$MT-AB(t,b) = \cup\alpha_x(E_0(x, \{r \mid r \in T \wedge t \in PL(r) \wedge b \notin I(r)\})$
Drop Behavior	$MT-DB(t,b) = \cup\alpha_x(E_0(x), \{r \mid r \in T \wedge t \in PL(r) \wedge$ $\neg\exists v(v \in PL(r) - \{t\} \wedge b \in N_e(v)\})$
Add Subtype Relationship	$MT-ASR(t,s) = \cup\alpha_x(E_0(x), \{r \mid r \in T \wedge t \in PL(r) \wedge$ $I(s) \neq \emptyset \wedge I(s) \neq (I(s) \cap I(r))\})$
Drop Subtype Relationship	$MT-DSR(t,s) = \cup\alpha_x(E_0(x), \{r \mid r \in T \wedge t \in PL(r) \wedge$ $\neg\exists v(v \in PL(r) - \{t\} \wedge s \in P_e(v)) \wedge$ $I(s) \neq \emptyset \wedge I(s) \cap \cup\alpha_y(N_e(y), PL(r) - \{s\})\})$
Add Type	$AT(t, \{s_1, \ldots, s_n\}, \{r_1, \ldots, r_m\}, \{b_1, \ldots b_j\}) =$ $\cup\alpha_y(\cup\alpha_x(MT-ASR(x,y), \{r_1, \ldots, r_m\}), \{t\} \cup \{s_1 \ldots s_n\})$
Drop Type	$DT(t) = \cup\alpha_x(E_0(x), \{r \mid r \in T \wedge t \in PL(r) \wedge I(t) \neq \emptyset \wedge$ $I(t) \neq I(t) \cap \cup\alpha_y(N_e(y), PL(r) - \{t\})\})$

The axioms of change propagation are shown in Table 4. They determine the sound and complete set of objects affected by the schema change operations on types outlined in Table 1. The purpose of each axiom is to return a sound and complete set of objects affected by the schema change associated with the axiom.

The semantics of each axiom is explained below along with an example of its action. Refer to Figure 2 for the type lattice structure of the examples and Table 5 for the essential properties and essential supertypes of each type in the example. All examples below use Figure 2 and Table 5 as a starting point - the examples are not cumulative.

Add Behavior (MT-AB): Adds behavior b as an essential property of type t (i.e., $N_e(t) = N_e(t) \cup \{b\}$). If b was not part of t then the objects of t must now support the new behavior b. This may require an update to the objects of t. For example, if b is implemented as a stored attribute then the objects of t

Table 5. Example Types

Type	Essential Supertypes (P_e)
T_object	
T_person	T_object
T_taxSource	T_object
T_student	T_person, T_object
T_employee	T_person, T_taxSource, T_object
T_company	T_taxSource, T_object
T_teachingAssistant	T_student, T_employee, T_person, T_object

Table 6. Example Properties

Type	Essential Properties (N_e)
T_object	
T_person	Name, Age
T_taxSource	Name, TaxBracket, GovID
T_student	Name, Grade
T_employee	Name, Salary, Age, TaxBracket
T_company	Name, Revenue, Phone
T_teachingAssistant	Hours, TaxBracket, GovID, Salary, Age

require additional memory to store the value of the attribute. Furthermore, the subtypes of t could inherit b as a new behavior and so their objects may be affected. The axiom determines all the types (r) that are a subtype of t and do not have b as part of their interface. The extended union over the shallow extent of these types gives the set of objects affected by the schema change.

Example: Suppose a Government ID (GovID) property is added to type T_person for identification purposes. This would add GovID to $N_e\,(T_person)$. Since this is a new property for T_person, the objects in the shallow extent of T_person are affected (i.e., $E_0\,(T_person)$). Furthermore, the objects in the shallow extent of T_student are also affected, but not the objects in the deep extent of T_employee because T_employee inherits the GovID property from T_taxSource. It may appear that a conflict has arisen for GovID between types T_person and T_taxSource. This can be averted because the representation/implementation of GovID in T_employee has been previously decided by its prior subtyping relationship with T_taxSource.

Drop Behavior (MT-DB): Deletes behavior b from the essential properties of type t (i.e., $N_e\,(t) = N_e\,(t) - \{b\}$). This will affect the objects of t only if b is natively defined on t (i.e., t does not inherit b from some other type). This may also affect the subtypes of t that only inherit b from t. The variable r is used to build a set of types that consist of the subtypes of t that inherit b only from

t. This is accomplished by specifying there does not exist a v such that v is a supertype of r (not including t) and b is an essential behavior of v. The union over the shallow extent of these types gives the set of objects affected by the schema change.

Example: If a schema operation drops property Age from the essential properties of T_person, then the objects in the shallow extent of T_person and T_student are affected. However, the deep extent of T_employee is not affected because it declares Age as an essential property that now becomes native to T_employee, so the interface of T_employee is not affected.

Add Subtype Relationship (MT-ASR): Adds a new subtype relationship between type t and type s ($t \preceq s$) by adding s as an essential supertype of t (i.e., $P_e(t) = P_e(t) \cup \{s\}$). The type t and its subtypes can be affected by this change if s introduces new properties that are inherited. The objects of a subtype r are affected by the schema change if the interface of s is not empty and it is not a subset of the interface of r. Again, the union over the shallow extent of these types gives the set of objects to be coerced.

Example: If T_taxSource is added as an essential supertype of T_student, then the objects in the shallow extent of T_student must have changes propagated to them because of the newly inherited behaviors taxBracket and GovID. However, the objects in the deep extent of T_teachingAssistant are not affected because this type has another link (or path) to T_taxSource through T_employee.

Drop Subtype Relationship (MT-DSR): Drops an existing subtype relationship between type t and type s by removing s as an essential supertype of t (i.e., $P_e(t) = P_e(t) - \{s\}$). The type t and its subtypes can be affected by this change if they inherit some behaviors only from s and all links to s are lost by dropping the subtype relationship from t. The variable v is used to determine that there are no other subtype links to s. The second line of the axiom determines if the behaviors of s are inherited from some other type even if all the subtype links to s are lost.

Example: Suppose the subtype relationship from T_teachingAssistant to T_employee is dropped. Now, T_teachingAssistant has no other link to T_employee or T_taxSource. However, the objects in its deep extent are not affected because T_teachingAssistant has declared a set of essential properties that include all the properties that it previously inherited from T_employee and T_taxSource. Since the properties are essential, they are kept with T_teachingAssistant. Thus, no objects are affected by change propagation in this case.

Add Type (AT): Add the type t to the schema with $\{s_1, \ldots, s_n\}$ as the essential supertypes of t, $\{r_1, \ldots, r_m\}$ as the initial subtypes of t, and $\{b_1, \ldots, b_j\}$ as the essential properties of t. Subtypes $\{r_1, \ldots, r_m\}$ can be incorporated by adding a new subtype relationship from each r_i to the new type t. This change can affect the objects in $\{r_1, \ldots, r_m\}$ and their subtypes because they may now inherit some properties in $\{b_1, \ldots, b_j\}$. Furthermore, the new type t may transitively introduce new subtyping relationships between the types $\{r_1, \ldots, r_m\}$ and the types $\{s_1, \ldots, s_n\}$. This may result in some of the properties defined on

$\{s_1, \ldots, s_n\}$ being inherited by $\{r_1, \ldots, r_m\}$, thus affecting the objects in their extents.

Example: Suppose a new type T_worker is added with essential supertypes {T_object, T_taxSource}, initial subtype {T_employee}, and essential properties {GovID, WorkersCompID}. Due to the axiom of rootedness and axiom of pointedness in Table 3, the system can automatically include T_object as an essential supertype and T_null as an initial subtype. The direct supertype links in the lattice will change as shown in Figure 3 so that there is a direct link from T_employee to T_worker and a direct link from T_worker to T_taxSource. The direct link from T_employee to T_taxSource is lost. The new property WorkersCompID (representing a worker's compensation identifier) is inherited by T_employee and T_teachingAssistant and so the objects of these two types are affected by the schema change.

The following example shows how a newly added type can affect the initial subtypes of the new type by transitively inheriting behaviors of the essential supertypes.

Example: Consider the generic type lattice in Figure 4(a) and the schema operation $AT\,(t, \{s_1\}, \{r_1, r_2\}, \{p_z\})$ that adds type t with essential supertype $\{s_1\}$, initial subtypes $\{r_1, r_2\}$, and essential property {p_z}. The essential properties of each type in the figure are listed below the type. Clearly, type r_1 is affected by the addition because it will inherit property p_z from the new type t. It appears that r_2 should not be affected because it already defines property p_z. However, r_2 is affected because it transitively inherits property p_x from s_1 through t. Note that r_1 also transitively inherits p_x. The updated lattice is shown in Figure 4(b).

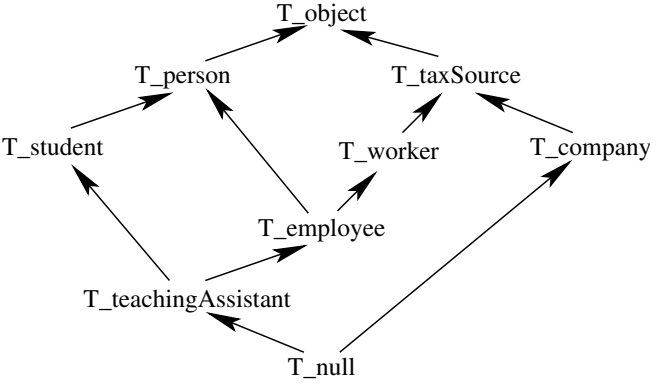

Fig. 3. Revised type lattice with type T_worker added

Drop Type (DT): Drops the type t from the lattice. This axiom is a simplified version of the MT-DSR axiom. It is simpler because there is no need

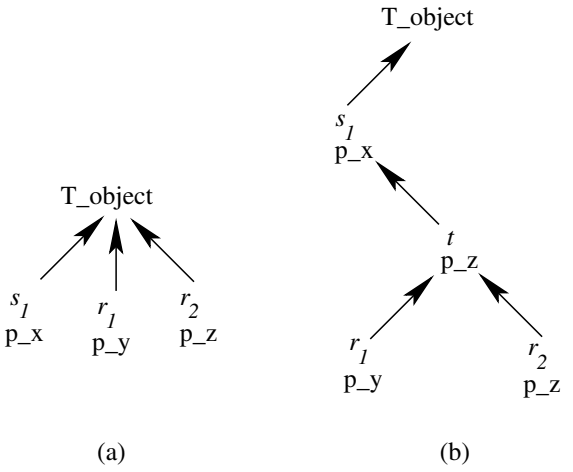

Fig. 4. Generic type lattice illustrating transitive inheritance for Add Type operation

to check for other subtype relationships to t since t will be removed from the lattice.

Example: If T_employee is dropped from the lattice, then the link from T_teachingAssistant to T_taxSource is lost because T_taxSource is not an essential supertype of T_teachingAssistant. This means that T_teachingAssistant will no longer inherit the properties of T_taxSource. Regardless, the objects in the deep extent of T_teachingAssitant are not affected by the schema change because all the properties defined by T_employee and T_taxSource have been defined as essential properties of T_teachingAssistant, or they are inherited from T_person.

If T_student is now dropped from the lattice, then a direct link from T_teachingAssistant to T_person is established because T_person is an essential supertype of T_teachingAssistant. The objects in the deep extent of T_teachingAssistant are affected because the Grade property defined by T_student is not defined as an essential property of T_teachingAssistant and so it is lost.

The axioms of Table 4 give a precise semantics for determining a sound and complete set of objects affected by schema evolution operations on types. These objects require a coercion mechanism to be applied (e.g., conversion, screening, filtering) so that they correspond to the updated version of the schema. It is clear that the axiomatic model can be used as a formal underlying model of a complete schema evolution mechanism for object-oriented environments. The descriptions and examples following the axiom specifications are not meant to provide a complete explanation. Many subtle qualities can be observed when using the axioms in various examples that are too great in number to present in the limited space allowed for this paper. We have implemented a tool for defining schema and performing schema evolution based solely on essential properties and essential supertypes. The tool currently supports all the axioms for the semantics

of change problem. The tool is currently being updated to incorporate the axioms for change propagation along with user selectable options for determining which coercion mechanism to apply to affected objects.

4 Related Work

Various systems have proposed solutions to the problems of *semantics of change* and *change propagation* for schema evolution in OBMSs. To support semantics of change, the most common approach is to define a number of invariants that must be satisfied by the schema, along with a set of rules and procedures that maintain the invariants with each schema change. To support change propagation, one solution is to explicitly *coerce* objects to coincide with the new definition of the schema. This technique updates the affected objects, changing their representation as dictated by the new schema. Unless a versioning mechanism is used in conjunction with coercion, the old representations of the objects are lost. *Screening*, *conversion*, and *filtering* are techniques that define when and how coercion takes place.

In *screening*, schema changes generate a conversion program that is independently capable of converting objects into the new representation. The coercion is not immediate, but rather is delayed until an instance of the modified schema is accessed. That is, object access is monitored by the system and whenever an outdated object is accessed, the system invokes the conversion program to coerce the object into the newer definition. Conversion programs resulting from multiple independent changes to a type are composed; meaning access to an object may invoke the execution of multiple conversion programs where each one handles a particular change to the schema. Screening causes processing delays during object access because the conversion program may have to be applied. Furthermore, it can be difficult to determine when the system no longer needs to check whether a particular conversion program is required. This can cause overhead during every object access and may increase the amount of supplementary information that the system needs to keep in the form of screening flags.

In *conversion*, each schema change initiates an immediate coercion of all objects affected by the change. This approach causes processing delays during schema modifications, but delays are not incurred during object access. Once conversion is complete, all objects are up to date.

Another solution for handling change consistency of instances is to introduce a new version of the schema with every modification and supplement each schema version with additional definitions that handle the semantic differences between versions. These additional definitions are known as *filters* and the technique is called *filtering*. Error handlers are one example of filters. They can be defined on each version of the schema to trap inconsistent access and produce error and warning messages.

In the filtering approach, changes are never propagated to the instances. Instead, objects become instances of particular versions of the schema. When the schema is changed, the old objects remain with the old version of the schema and

new objects are created as instances of the new schema. The filters define the consistency between the old and new versions of schema and handle the problems associated with properties written according to one version accessing objects of a different version. This approach introduces the overhead of maintaining the separate versions and the filters between them that need to be applied from time to time.

A *hybrid* approach combines two or more of the above methods. For example, a system could use filtering as the underlying mechanism and allow explicit coercion to newer versions of types through screening or conversion. This could be used to reduce the overhead in the number of versions and filters that need to be maintained. Another example is a system that takes an active role by using screening as the default and switching to conversion whenever the system is idle.

The axiomatic model of change propagation is responsible for identifying the sound and complete set of objects affected by a schema change. This set can serve as input to any of the coercion methods described above. Thus, the axiomatic model can act as a "front-end" to systems using these approaches.

Orion [1,5] is the first system to introduce an invariants and rules approach as a structured way of describing schema evolution in OBMSs. Invariants define the consistency of the schema under the constraints of the object model. Rules are introduced to guide the preservation of the invariants when choices in modifying the schema arise. Orion defines five invariants and a set of twelve accompanying rules for maintaining the invariants over schema changes. Orion's taxonomy of changes represents the majority of typical schema modifications allowed in most OBMSs. Change propagation in Orion is handled through screening that coerces out-of-date objects to new schema definitions when the objects are accessed.

Schema evolution in GemStone [9] is similar to Orion in its definition of a number of invariants. The GemStone model is less complex than Orion in that multiple inheritance and explicit deletion of objects are not permitted. As a result, the schema evolution policies in GemStone are simpler and cleaner, but not as powerful as those of Orion. For example, while Orion defines twelve rules to clarify the effects of schema modification, GemStone requires no such rules. Conversion is used in GemStone to propagate changes to the instances. Literature on GemStone mentions the possibility of a hybrid approach that allows both conversion and screening, but it is not clear if such a system has been developed.

Skarra and Zdonik [14,15] define a framework for versioning types in the Encore object model as a support mechanism for evolving type definitions. Their work is focused on dealing with change propagation. The schema evolution operations of Encore are similar to Orion. The authors introduce a generic type as a collection of individual versions of that type. This is known as the *version set* of the type. Every change to a type results in the generation of a new version of that type. Since a change to a type can also affect its subtypes, new versions of the subtypes may also be generated. By default, objects are bound to a specific type version and must be explicitly coerced to a newer version. Since objects are bound to a specific type version, a problem of missing information can arise if programs (i.e., methods) written according to one type version are applied

to objects of a different version. For example, if a property is dropped from a type, programs written according to an older type version may no longer work on objects created with the newer version because the newer object is missing some information (i.e., the dropped property). Similarly, if a property is added to a type, programs written with the newer type version in mind may not work on older objects because of missing information. For this reason, type versions include additional definitions, called *handlers*, which act as filters for managing the semantic differences between versions. This approach is the first to address the issue of maintaining consistency between versions of types.

Nguyen and Rieu [7] discuss schema evolution in the Sherpa model and compare their work to Encore, Gemstone, Orion, and one of their earlier models for CAD systems called Cadb. The emphasis of this work is to provide equal support for semantics of change and change propagation. The schema changes allowed in Sherpa follow those of Orion. Schema changes are propagated to instances through conversion or screening, which is selected by the user. However, only the conversion approach is discussed. Change propagation is assisted by the notion of *relevant classes*. A *relevant class* is a semantically consistent partial definition of a *complete class* and is bound to the class. A relevant class is similar to a type version in [14] and a complete class resembles a version set. The purpose of relevant classes is to evaluate the side effects of propagating schema changes to the instances and to guide this propagation.

In OTGen [6] the focus shifts from dynamic schema evolution to database reorganization. The invariants and rules approach is used, and the typical schema changes are allowed. The invariants are used to define default transformations for each schema change. Schema changes produce a transformation table that describes how to modify affected instances. Multiple schema changes are usually grouped and released as a package called a *transformer*. Screening is used to apply the transformer and propagate changes to the instances. Multiple releases are composed and, thus, access to an older object can invoke multiple transformers to bring the object up to date. One result of the database reorganization approach is that multiple changes are packaged into a single release and this is expected to reduce the number of screening operations that need to be invoked for each object access. Another result is that transformers are represented as tables that are initialized by OTGen.

The Tigukat OBMS [8] incorporates a uniform object model and schema evolution is handled as type and property extensions to the base model. As discussed in Section 2, a sound and complete axiomatic model for the semantics of change problem has been developed in the context of Tigukat. This axiomatic model is used to compare the schema evolution operations of Tigukat, Orion, GemStone, O2, and others [2,12]. Change propagation in Tigukat is handled by a filtering approach that uses behavior histories [10], which are based on the temporal aspects of the object model [3,4]. When a change is made to the schema, the change is not automatically propagated to the instances. Instead, the old version of the schema is maintained and the change is recorded in the proper temporal histories. Existing objects continue to maintain the characteristics of

the older schema while newly created objects correspond to the semantics of the newer schema. Coercion of older objects to newer versions of the schema is optional in Tigukat. Since different versions of types are maintained through temporal histories, the schema information of older objects is available and can be used to continue processing these objects in a historical manner. If coercion is desired, the entire object does not need to be updated at once. Objects can be coerced to a newer version of the schema one property at a time. This means that some properties of an object may work with newer versions, while others may work with older ones. This is in contrast to other models where an object is converted in its entirety to a newer schema version, thereby losing the old information of the object.

5 Conclusion

The formal axiomatic model of schema evolution is a powerful mechanism for reasoning about objectbase management systems. The model consists of two major components: namely, (i) *semantics of change*, and (ii) *change propagation*. Our previous work [12] deals with the first component. This paper extends that work to change propagation. The first step in performing change propagation is to identify the affected objects. The subsequent steps carry out the changes to the objects by coercing them into structures that correspond to the evolved types. The axiomatic model extensions described in this paper give a precise semantics for identifying the objects affected by a schema change. The extended model determines the affected objects using existing information from the type system without making any changes to the type system itself. This could be used in conjunction with a graphical display to show the implications of a schema change and allow the designer to easily cancel the change. We show that this model can be used as a "front-end" component to various coercion methods. Further, the model ties in with the axiomatic model for the semantics of change problem [12] to form a complete axiomatic model of schema evolution in OBMSs. The model is easy to work with because the designer only needs to specify and maintain two sets for each type: the *essential supertypes* (P_e) and *essential properties* (N_e). Subsequently, the axioms provide an automated means of managing the *semantics of change* and *change propagation* based on modifications to these two sets. The utility of this model is further demonstrated by its inherent automation. Application of the axioms is all that is required to determine which objects are affected. Other researchers have demonstrated the utility of the axiomatic model by extending it into areas such as real-time systems Zhou *et al.* [16], but their research does not addressed the issue of change propagation. Finally, the axiomatic model is proven sound and complete by the work in this paper together with previous work [12].

This research can be extended in several directions. Our current schema management tool implements the *semantics of change* system [12]. Clearly, the change propagation described in this paper can be added to the schema management tool to enhance its functionality. Another interesting research direc-

tion is to determine how these techniques can be applied to a federated system that requires integration. For example, adding a "new" database into a federated schema could be seen as little more than a change to the existing integrated schema. Successful application of these techniques would make substantial progress in automating the process of system integration. Another research direction to consider is extending the model to manage security within complex systems such as OBMSs. An axiomatic representation of role based security models may assist in the architecture and management of changing security permissions in these systems.

Another problem to investigate is the extension into object view models and the management of object view schemas. Evolution of view schemas is more challenging because different perspectives are placed on objects according to the groups of users accessing them. In addition, closure constraints must be taken into consideration when developing view schemas. Closure refers to the condition where including a certain type (or property) in a view, requires other types (and/or properties) be added to ensure the new ones are meaningful. For example, if you create a view that has a T_car type with a Manufacturer property, then the T_company type should be included in some form so that the Manufacturer can be related to it. One open question asks what portions of T_company should be included to be semantically meaningful without compromising security or scope considerations. Another question asks to what level should the inclusion be carried out. That is, if T_company is included with some properties then what other types need to be included for them to be meaningful?

References

1. J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in Proc. ACM SIGMOD Int'l Conf. on Management of Data, 1987, pp. 311-322.
2. A. A. D'Silva, "Dynamic Evolution of Integrated Schemas for Federated Objectbase Systems." University of Manitoba, Computer Science, M.Sc. 1998.
3. I. Goralwalla, D. Szafron, M. T. Özsu, and R. J. Peters, "Managing Schema Evolution using a Temporal Object Model," in Proc. Int'l Conf. on Conceptual Modeling, 1997.
4. I. Goralwalla, D. Szafron, M. T. Özsu, and R. J. Peters, "Approach to Managing Schema Evolution in Object Database Systems," Data & Knowledge Engineering, 1998. In press - to appear November 1998.
5. W. Kim and H.-T. Chou, "Versions of Schema for Object-Oriented Databases," in Proc. Int'l Conf. on Very Large Databases, 1988, pp. 148-159.
6. B. S. Lerner and A. N. Habermann, "Beyond Schema Evolution to Database Reorganization," in Proc. Int'l Conf. on Object-Oriented Programming, Languages, Applications, 1990, pp. 67-76.
7. G. T. Nguyen and D. Rieu, "Expert Database Support for Consistent Dynamic Objects," in Proc. Int'l Conf. on Very Large Databases, 1987, pp. 493-500.
8. M. T. Özsu, R. Peters, D. Szafron, B. Irani, A. Lipka, and A. Muñoz, "TIGUKAT: A Uniform Behavioral Objectbase Management System," The VLDB Journal, 4(3):445-492, 1995. Special issue on persistent object systems.

9. D. J. Penney and J. Stein, "Class Modification in the GemStone Object-Oriented DBMS," in Proc. Int'l Conf. on Object-Oriented Programming, Languages, Applications, 1987, pp. 111-117.
10. R. J. Peters, "TIGUKAT: A Uniform Behavioral Objectbase Management System." University of Alberta, Computing Science, Ph.D. 1994. Available as University of Alberta Technical Report TR94-06.
11. R. J. Peters and M. T. Özsu, "Axiomatization of Dynamic Schema Evolution in Objectbases," in Int'l Conf. on Data Engineering. Taiwan, 1995, pp. 156-164.
12. R. J. Peters and M. T. Özsu, "An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems," ACM Transactions on Database Systems, 22(1):75-114, 1997.
13. G. Shaw and S. Zdonik, "A Query Algebra for Object-Oriented Databases," in Proc. Int'l Conf. on Data Engineering, 1990, pp. 154-162.
14. A. H. Skarra and S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database," in Proc. Int'l Conf. on Object-Oriented Programming, Languages, Applications, 1986, pp. 483-495.
15. A. H. Skarra and S. B. Zdonik, "Type Evolution in an Object-Oriented Database," in Research Directions in Object-Oriented Programming: MIT Press, 1987, pp. 393-415.
16. L. Zhou, E. A. Rundensteiner, and K. G. Shin, "Schema Evolution of an Object-Oriented Real-Time Database System for Manufacturing Automation," IEEE Trans. on Knowledge and Data Eng., 9(6):956-977, 1997.