Axiomatization of Dynamic Schema Evolution in Objectbases*

Randal J. Peters and M. Tamer Özsu
Laboratory for Database Systems Research
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2H1
{randal,ozsu}@cs.ualberta.ca

Abstract

The *schema* of a system consists of the constructs that model its entities. *Schema evolution* is the timely change and management of the schema. *Dynamic schema evolution* is the management of schema changes while the system is in operation. We propose a sound and complete axiomatic model for dynamic schema evolution in objectbase management systems (OBMSs) that support subtyping and property inheritance. The model is formal, which distinguishes it from the traditional approach of informally defining a number of invariants and rules to enforce them. By reducing systems to the axiomatic model, their functionality with respect to dynamic schema evolution can be compared within a common framework.

1 Introduction

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, medical imaging, and geo-information systems, to name a few. An important characteristic of these applications is that their schema changes frequently and dynamically. For example, in an engineering design application many components of an overall design may go through several modifications before a final product design is achieved. These kinds of changes require modifications to the way components are modeled (i.e., the schema). The evolutionary characteristic of these applications requires sophisticated dynamic schema evolution policies for managing changes in schema and ensuring the overall consistency of the system.

The *schema* (or *meta-information*) of an objectbase management system (OBMS) is the information that describes the structure and operations applicable to the object instances. *Dynamic schema evolution* is the process of applying changes to the schema in a consistent fashion and propagating these changes to the instance level while the system is in operation.

Typical schema changes in an OBMS include adding and dropping types, adding and dropping sub/supertype relationships between types, and adding and dropping properties of types. A complete list of possible schema changes is given in [1]. A typical schema change can affect many aspects of a system. There are two fundamental problems to consider:

1. **Semantics of change:** This refers to the effects of the schema change on the overall way in which the system organizes information;

2. **Change propagation:** This refers to the method of propagating schema changes to the objects.

For the first problem, the traditional approach is to define a number of invariants that must be satisfied by the schema and then to define rules for maintaining these invariants. The invariants and rules are dependent on the underlying object model and since object models differ, their schema evolution policies differ as well. Furthermore, the lack of a formal semantics make systems difficult to compare. In this paper, we propose a sound and complete axiomatization for dynamic schema evolution in OBMSs that is powerful enough to describe the semantics of change in different systems. These systems can be then reduced to this common model and compared.

For the second problem, the typical solution is to explicitly *coerce* objects to coincide with the new schema definition. *Screening*, *conversion*, and *filtering* are techniques for defining when and how coercion takes place. Due to space restrictions, change propagation is not addressed in this paper.

In order to clarify the expressiveness of the axiomatization, the dynamic schema evolution policies of the TIGUKAT[1] OBMS [5, 7] are presented as an example and reduced to the axiomatic model. TIGUKAT is being developed at the University of Alberta and has a uniform, extensible object model capable of supporting database services within a single underlying framework. In keeping with its modeling capability, the schema evolution policies are defined as extensions to the base system. Thus, this paper has two main contributions:

1. it introduces a sound and complete axiomatic model that can be used to formalize and compare the schema management approaches of OBMSs, and

2. it presents the uniform dynamic schema evolution strategies employed by the TIGUKAT OBMS.

*This research is supported by the Natural Science and Engineering Research Council (NSERC) of Canada under research grant OGP0951.

[1] TIGUKAT (tee-goo-kat) is a term in the language of the Canadian Inuit people meaning "objects." The Canadian Inuits, commonly known as Eskimos, are native peoples of Canada with an ancestry originating in the Arctic regions of the country.

The remainder of the paper is organized as follows. The axiomatization of schema evolution is defined in Section 2 and a sketch of the soundness and completeness proofs are given. An overview of the TIGUKAT object model and the representation of schema evolution in TIGUKAT in terms of the axiomatic model is shown in Section 3. The reduction of Orion and other systems to the axiomatic model is presented in Section 4. A discussion and comparison of the axiomatization of TIGUKAT and Orion is given in Section 5. Section 6 contains the concluding remarks and a discussion of future work.

2 Axiomatization of Schema Changes

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

Subtyping is a facility that allows types to be built incrementally from other types. A *subtype relationship* can be represented as a directed arrow from a subtype (the tail) to its supertype (the head). A subtype inherits all properties of its supertype(s). When a subtype has multiple supertypes, it is known as *multiple subtyping* and results in a *directed acyclic graph* or *lattice* of subtype relationships.

Typical schema changes include adding and dropping types, adding and dropping subtype relationships, and adding and dropping type properties. How these affect 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 notation for the axiomatic model is shown in Table 1. The terms denote various arrangements of types and properties that can be represented in any object model. We now explain each of these terms and use the simple type lattice in Figure 1 as an example to clarify their semantics.

Term	Description
$\mathcal{T}$	The lattice of all types in the system.
$s, t, \top, \bot$	Type elements of $\mathcal{T}$.
$P(t)$	Immediate supertypes of type t.
$P_e(t)$	Essential supertypes of type t.
$PL(t)$	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.
$\alpha_x(f, \mathcal{T}')$	Apply-all operation.

Table 1: Notation for axiomatization.

The set of types $\mathcal{T}$ represents all the types in the system on which dynamic schema evolution operations are performed. The set consisting of all types shown in Figure 1 forms $\mathcal{T}$ in this example.

The set $\mathcal{T}$ forms a lattice through subtype relationships maintained by the *immediate supertypes*, $P(t)$, for each type t. The immediate supertypes 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, P(T_teachingAssistant) = {T_student, T_employee}[2]. The other supertypes of T_teachingAssistant (i.e., T_person, T_taxSource, and T_object) can be reached through T_student or T_employee.

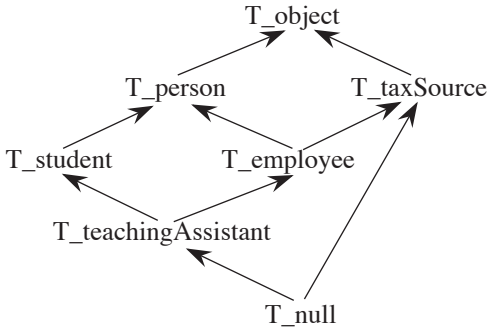

Figure 1: Simple type lattice.

The *essential supertypes*, $P_e(t)$, are those specified by the schema designer as being essential to the construction of type t. The meaning of essential supertypes is that they should be maintained as supertypes of t for as long as possible during schema evolution. 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 s and t or by dropping s entirely. Note that $P(t) \subseteq P_e(t)$, meaning immediate supertypes are essential. In Figure 1, the schema designer may have specified the essential supertypes of T_teachingAssistant as:

$$P_e(\text{T\_teachingAssistant}) = \{\text{T\_student, T\_employee, T\_person, T\_object}\}$$

This means that if, for example, T_student and T_employee are dropped as immediate supertypes of T_teachingAssistant, then T_person would be established as an immediate supertype because it is essential. However, T_taxSource would be lost as a supertype because it was not declared as essential.

The *supertype lattice*, $PL(t)$, of a type t is the set of all types of which t is a subtype, including t itself. For example, PL(T_employee) = {T_employee, T_person, T_taxSource, T_object}.

The *native properties*, $N(t)$, of a type t are those that are not defined in any of the supertypes of t. Note that these properties may be defined by other types not in a subtype relationship with t. 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.

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 is the union of the properties defined on T_person, T_taxSource, and T_object.

[2]The prefix "T_" indicates a type.

The *essential properties*, $N_e(t)$, are those specified by the schema designer as being essential to the construction of type t and consist of all native and possibly some inherited properties $[N(t) \subseteq N_e(t)]$. The meaning of essential properties is that they should be a part of t (either inherited or native) for as long as possible during schema evolution. This may require, for example, the adoption of inherited properties as native if the supertype defining those properties natively is removed. For example, assume there is a "taxBracket" property defined on T_taxSource that is declared as essential in T_employee. This property is inherited by T_employee, but if T_taxSource were deleted, then the "taxBracket" property would be adopted by T_employee as a native property. The *interface*, $I(t)$, of a type t is the union of native and inherited properties of t.

In Table 2 we show how the various arrangements of types and properties in Table 1 can be computed from P_e and N_e, which are specified by the system designer. All schema evolution operations can be handled through these two terms, which make it simple for the schema designer to understand and manage. The axiomatic model takes care of rearranging the schema to conform to these two inputs.

The specification of P_e and N_e can be system or user managed. 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, as default, assume that all supertypes and properties (including inherited properties) are essential in a given type, or that none are essential. The various systems define different semantics for the notions of subtyping, inheritance, and nativeness. This formalization of the concepts leaves it open to interpretation by the individual systems. It is likely that some combination of user and system managed control would be most effective. For example, the system may assume that only the initial supertypes and properties defined on a type are essential. By default, none of the inherited properties are assumed to be essential. A schema designer may evolve the schema by adding and dropping properties, and adding and dropping subtype relationships. These operations are noted in N_e and P_e as the "essential" structure of properties and types. These operations may not be fulfilled in N and P because of other inheritance links that may be present. For example, defining an already inherited property on a type would not include the property in N, but would include it in N_e. Furthermore, adding a subtype relationship between s and t such that s is the supertype of t always includes s in $P_e(t)$, but it is added to $P(t)$ if and only if $s \notin PL(t)$. In this way, $N(t)$ maintains the minimal properties that need to be defined by t and $P(t)$ maintains the minimal supertypes of t. This minimality is important for the efficiency of the system.

We assume the availability of an *apply-all* operation in the axiomatic model. This operation, denoted $\alpha_x(f, \mathcal{T}')$, applies the unary function f to the elements of a set of types $\mathcal{T}' \subseteq \mathcal{T}$. The function f is over the single variable x, denoted as the subscript of the α symbol. Other variables appearing within the parenthesis of the α symbol are substituted with their values and remain constant throughout the apply-all operation.

The semantics of the apply-all operation is to let x range over the elements of $\mathcal{T}'$ and for each binding of x, evaluate f and include the result in the final result set. If $\mathcal{T}'$ is empty, the empty set is returned. In a functional notation, one may think of α as applying the lambda function $\lambda x.f$ to every element of $\mathcal{T}'$ and returning a set containing the results.

Table 2 summarizes the axiomatization of subtyping and property inheritance. Note that the apply-all operations specified in the table return a set of sets and the large union operator preceeding each apply-all performs the extended union over these member sets. We define the extended union of the empty set as the empty set.

Axiom of Closure: Types in $\mathcal{T}$ have supertypes in $\mathcal{T}$, giving closure to $\mathcal{T}$.

Axiom of Acyclicity: There are no cycles in the type lattice formed by $\mathcal{T}$. This axiom disallows any element of $\mathcal{T}$ from appearing in the supertype lattice of any of its supertypes, which would form cycles.

Axiom of Rootedness: There is a single type $\top$ in $\mathcal{T}$ that is the supertype of all types in $\mathcal{T}$. The type $\top$ is called the *root* or *least defined type* of $\mathcal{T}$. This axiom can be relaxed in which case the type lattice has many roots and is known as a *forest*.

Axiom of Pointedness: There is a single type $\bot$ in $\mathcal{T}$ that is the subtype of all types in $\mathcal{T}$. The type $\bot$ is called the *base* or *most defined type* of $\mathcal{T}$. The lattice is said to be *pointed at* $\bot$. This axiom can be relaxed in which case the lattice has many leaves.

Axiom of Supertypes: The set of immediate supertypes of a type t is exactly the subset of the essential supertypes that cannot be reached indirectly through some other type. This axiom provides a way of automatically instantiating the immediate supertypes of a given type based on the essential supertypes of that type.

Axiom of Supertype Lattice: The supertype lattice of a type t includes t itself and recursively the types in the supertype lattices of its immediate supertypes. This axiom provides a way of automatically instantiating the supertype lattice of a given type.

Axiom of Interface: The interface of a type consists of the union of the native and inherited properties of that type. This axiom provides a way of automatically instantiating the interface of a given type.

Axiom of Nativeness: The native properties of a type is the subset of the essential properties that are not inherited. This axiom provides a way of automatically instantiating the native properties of a given type.

Axiom of Inheritance: The inherited properties of a type is the union of the interfaces of its immediate supertypes. This axiom provides a way of automatically instantiating the inherited properties of a given type.

There are several simplifications that can be made to the axioms in order to reduce the amount of mutual recursion among them. Furthermore, several optimizations can be made to the way in which the axioms generate their results. Space restrictions do not allow us to present these simplifications and optimizations in this paper.

Axiom of Closure	$\forall t \in \mathcal{T}, P_e(t) \subseteq \mathcal{T}$	(1)
Axiom of Acyclicity	$\forall t \in \mathcal{T}, t \notin \bigcup \alpha_x(PL(x), P(t))$	(2)
Axiom of Rootedness	$\exists \top \in \mathcal{T}, \forall t \in \mathcal{T} \mid \top \in PL(t) \wedge P_e(\top) = \{\}$	(3)
Axiom of Pointedness	$\exists \bot \in \mathcal{T}, \forall t \in \mathcal{T} \mid t \in PL(\bot)$	(4)
Axiom of Supertypes	$\forall t \in \mathcal{T}, P(t) = P_e(t) - \bigcup \alpha_x(PL(x) \cap P_e(t) - \{x\}, P_e(t))$	(5)
Axiom of Supertype Lattice	$\forall t \in \mathcal{T}, PL(t) = \bigcup \alpha_x(PL(x), P(t)) \cup \{t\}$	(6)
Axiom of Interface	$\forall t \in \mathcal{T}, I(t) = N(t) \cup H(t)$	(7)
Axiom of Nativeness	$\forall t \in \mathcal{T}, N(t) = N_e(t) - H(t)$	(8)
Axiom of Inheritance	$\forall t \in \mathcal{T}, H(t) = \bigcup \alpha_x(I(x), P(t))$	(9)

Table 2: Axiomatization of subtyping and behavioral inheritance.

To illustrate the expressiveness of the axioms, consider again the simple type lattice in Figure 1. Axioms 1 and 2 are satisfied by the lattice.

Axiom 3 holds when $\top$ = T_object and Axiom 4 holds when $\bot$ = T_null. Assume the essential supertypes of T_teachingAssistant are defined as follows:

$$P_e(\text{T\_teachingAssistant}) = \{\text{T\_student}, \text{T\_person}, \text{T\_employee}, \text{T\_object}\}$$

That is, it is essential that a teaching assistant is a student, person, employee, and object, but not essential that it is a tax source. Note that teaching assistants are tax sources by inheritance through T_employee. The meaning of this separation is that if teaching assistants cease to be employees, by removing the subtype relationship, then they automatically cease to be taxable sources. Axiom 5 instantiates the immediate supertypes of T_teachingAssistant as {T_student, T_employee}. Now, if T_student is dropped from P_e(T_teachingAssistant), then the new instantiation of the immediate supertypes would only include T_employee. The properties inherited from T_student are lost in T_teachingAssistant, except for those declared in N_e(T_teachingAssistant). Moreover, if T_employee is dropped as an essential supertype, then Axiom 5 instantiates {T_person} as the only immediate supertype of T_teachingAssistant. The properties of T_employee and T_taxSource are lost in T_teachingAssistant (except for the essential properties).

The axioms provide a consistent and automatic mechanism for re-computing the entire type lattice structure after a change is made to either the essential supertypes P_e or the essential properties N_e of a type. Changes to these two components are fundamental to schema evolution and the axiomatic model can handle variations of the other type and property arrangements depending on the defaults required. This can in turn be used to describe and compare schema evolution in any object model that supports subtyping and property inheritance.

Due to space restrictions, only a sketch of the proofs for soundness and completeness are presented. The full proofs will appear in a subsequent paper.

Theorem 2.1 The schema evolution axioms are sound.
Proof: Assumes $P_e(t)$ and $N_e(t)$ are sound and then shows through subset inclusion and induction on maximal path lengths to root type T_object that only sound sets are produced for $P(t), PL(t), I(t), N(t)$ and $H(t)$.

Theorem 2.2 The schema evolution axioms are complete.
Proof: Assumes $P_e(t)$ and $N_e(t)$ are complete and then shows through induction on maximal path lengths to root type T_object that only complete sets are produced for $P(t), PL(t), I(t), N(t)$ and $H(t)$.

3 Schema Evolution in TIGUKAT

In this section, we give a brief overview of the TIGUKAT object model, define the dynamic schema evolution policies of TIGUKAT, and indicate how these policies can be described using the axiomatic model presented in Section 2. We focus on the definition of the semantics of schema changes and exclude change propagation from the discussion. See [7] for a discussion of change propagation in TIGUKAT that uses the temporality of the model [2].

3.1 Object Model Overview

The TIGUKAT object model [5, 7] is purely *behavioral* with a *uniform* object semantics. The model is *behavioral* in that all access and manipulation of objects is based on the application of behaviors to objects. Behaviors in TIGUKAT correspond to the generic concept of properties discussed in Section 2. The model is *uniform* in that every component of information, including its semantics, is modeled as a *first-class object* with well-defined behavior.

The primitive type system of TIGUKAT is shown in Figure 2. Types define behaviors that are applicable to their instances. The type T_object is the root of the type system and T_null is the base. We concentrate on the shaded types in the figure, which are used to model schema, and describe their role in supporting schema evolution in terms of the axiomatic model. For a complete model definition, including primitive behaviors, see [7].

Primitive objects of TIGUKAT include: *atomic entities* (reals, integers, strings, etc.); *types* for defining common features of objects; *behaviors* for specifying the semantics

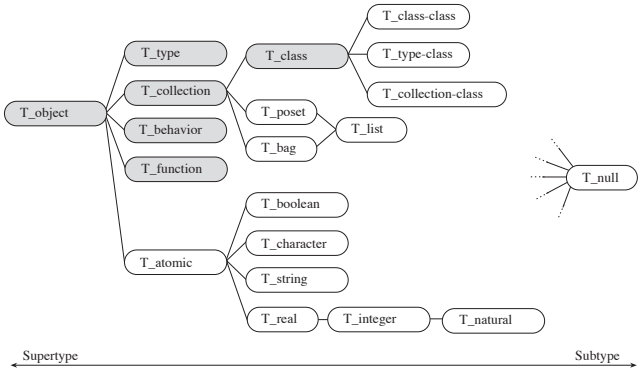

Figure 2: Primitive type system of TIGUKAT.

of object properties; *functions* for specifying implementations of behaviors; *classes* for automatic classification of objects based on their type[3]; and *collections* for general heterogeneous groupings of objects. In this paper, a reference prefixed by "T_" refers to a type, "**C_**" a class, and "*B_*" a behavior. For example, T_person refers to a type, **C_person** its class, and *B_age* one of its behaviors. A reference such as David, without a prefix, denotes some other application specific reference.

Objects consist of a unique identity and an encapsulated state. Access and manipulation of objects occurs exclusively through the application of behaviors. We clearly separate the definition of a behavior from its possible implementations (functions/methods). This supports *overloading* and *late binding* of implementations to behaviors.

One component of every behavior is its semantics. We use signatures as a partial semantics of behaviors. A signature includes a name used to apply the behavior, a list of argument types, and a result type. The object to which a behavior is applied is called the *receiver*. We use the dot notation $o.b$ to denote behavior b applied to object o.

A *type* defines behaviors and encapsulates behavior implementations and state representation for objects created using that type. TIGUKAT supports *multiple subtyping*, so the type structure forms a lattice. The lattice is rooted by the type T_object and *pointed* at the base type T_null. The type T_null defines objects that can be assigned to behaviors when no other result is known (e.g., null, undefined, etc.). The set of behaviors defined by a type specify the *interface* for the objects of that type. A type interface is separated into *inherited* and *native* behaviors that correspond to H and N in the axiomatic model.

Types represent the foundation of schema in most object models, including TIGUKAT. The fundamental schema evolution operations on types are to add and drop types, add and drop subtype relationships between types, and add and drop behaviors (i.e., properties) of types. In order to illustrate the principles of the axiomatization in Section 2, we define the modeling of types in TIGUKAT. The uniformity of TIGUKAT dictates that types are modeled as objects. The primitive type T_type defines the behaviors of types. The behaviors related to schema evolution include *B_supertypes*, *B_super-lattice*, *B_interface*, *B_native*, and *B_inherited*. The *B_supertypes* behavior returns the immediate supertypes of receiver type. The *B_super-lattice* behavior returns a partially ordered collection of types representing the supertype lattice pointed at the receiver type and rooted at T_object. Behaviors *B_interface*, *B_native*, and *B_inherited* represent the collections of behaviors related to the interface components of types.

A *class* ties together the notions of *type* and *object instances*. A *class* is a supplemental, but distinct, construct responsible for managing all instances of a particular type (i.e., the type extent). In this way, the model clearly separates types from from their extents.

Collections are defined as heterogeneous grouping constructs as opposed to *classes*, which are homogeneous up to inclusion polymorphism. Object creation occurs only through classes; thus they are extents of types and are managed automatically by the system. Collections are managed explicitly by the user.

The types T_class-class, T_type-class, and T_collection-class are part of the extended *meta* type system. Their placement within the type lattice directly supports the uniformity of the model and brings the definition of the meta-model within the model itself. For a discussion on the architecture of the meta-model and the features it provides (e.g., class behaviors, reflective queries), see [8].

3.2 Definition of Schema

There are different kinds of objects modeled by TIGUKAT, some of which are classified as schema objects. All objects managed by TIGUKAT fit in the category of *type, class, behavior, function, collection* or *other*. These categories are used to distinguish the "schema" of the model and the changes that affect it. First, the definition of what constitutes schema objects is presented. This is followed by the definition of the "schema."

Definition 3.1 *Schema Objects:* The following defines the primitive schema objects of the model:

- Types are schema objects. The class **C_type** forms the set of *type schema objects* denoted as TSO. This is equivalent to $\mathcal{T}$ in the axiomatic model.

- For all types $t \in TSO$, the extended union over interfaces of these types (i.e., $\bigcup t.B\_interface$) forms the set of *behavior schema objects* denoted as BSO. Only those behaviors defined in the interface of some type are considered to be behavior schema object, which means $BSO \subseteq$ **C_behavior**. BSO represents all properties in the axiomatic model, which is equivalent to $I(\perp)$.

- For all behaviors $b \in BSO$ and for all types $t \in TSO$, the extended union over the implementations of these behaviors (i.e., $\bigcup b.B\_implementation(t)$) forms the set of *function schema objects* denoted as FSO. Only those functions defined as the implementation of some behavior for some type are considered to be function schema objects, which means $FSO \subseteq$ **C_function**. Function objects denote the implementations of properties in the axiomatic model, which can be represented as attributes, methods, and so on. The axiomatic model is high-level and does not directly deal

[3] Types and their extents are separate constructs in TIGUKAT.

with implementations or the conflict resolution strategies associated with them. Conflict resolution of properties is at a semantic level in which the semantics of a property is unique and, therefore, simple set operations can be used to resolve conflicts.

- The class **C_collection** forms the set of *collection schema objects* denoted as LSO. Collections represent a flat space of heterogeneous, user-defined and managed object groupings. The axiomatic model does not restrict the management of collections.

- The class **C_class** forms the set of *class schema objects* denoted as CSO. Note that $CSO \subseteq LSO$. Classes in TIGUKAT mirror types in that a class is responsible for managing the extent of a type. Thus, the subset inclusion structure of classes is represented by the subtyping lattice in the axiomatic model. □

Definition 3.2 *Schema:* The *schema* of a TIGUKAT objectbase is the union of all schema object sets:

$$schema \equiv TSO \cup BSO \cup FSO \cup LSO \cup CSO$$

Note that CSO is included for completeness. □

There are three basic operations that can be performed on objects: *add, drop*, and *modify*. Table 3 shows the combinations between the object categories and the kinds of operations that can be performed. The **bold** entries represent combinations that imply schema evolution modifications, while the *emphasized* entries denote changes that are not considered to be part of the schema evolution.

3.3 Semantics of Change

In this section the modifications that affect the schema are described. The basic operations affecting the schema include adding behaviors to a type definition, dropping behaviors from a type definition, changing the implementation of a behavior in a type, and adding and dropping classes. The other schema changes, namely, adding and dropping types, adding and dropping subtype relationships, dropping behaviors and dropping functions are defined in terms of the basic operations.

The **MT-AB (Modify Type - Add Behavior)** operation adds a behavior as an essential component of a type and the behavior then becomes part of BSO. To add behavior b to type t, b is added to $N_e(t)$ and $N(t), H(t), I(t)$ are recomputed. The results are reflected in $t.B\_native, t.B\_inherited, t.B\_interface$, respectively.

The **MT-DB (Modify Type - Drop Behavior)** operation drops a behavior as an essential component of a type, which could possibly remove it from BSO. To drop behavior b from type t, b is removed from $N_e(t)$ and $N(t), H(t), I(t)$ are recomputed. The results are reflected in $t.B\_native, t.B\_inherited, t.B\_interface$, respectively. Note that this may not actually remove b from the interface of t because b may be inherited from one or more supertypes of t. However, if eventually the links to all supertypes defining b are removed, then b will no longer be part of t.

The **MT-ASR (Modify Type - Add Subtype Relationship)** operation adds a type as an essential supertype of another type, which effectively adds a subtype relationship between the two types. To add type s as a supertype of type t, s is added to $P_e(t)$ and all computations depending on P_e are recomputed. The results of any lattice change are reflected in $t.B\_supertypes$. Due to the axiom of acyclicity, the addition of a type as a supertype of another type is rejected if it introduces a cycle into the lattice.

The **MT-DSR (Modify Type - Drop Subtype Relationship)** operation drops a type as an essential supertype of another type, which effectively drops a subtype relationship between the two types. To drop type s as a supertype of type t, s is removed from $P_e(t)$ and all computations depending on P_e are recomputed. The results of any lattice change are reflected in $t.B\_supertypes$. Due to the axiom of rootedness, which TIGUKAT obeys, a subtype relationship to T_object cannot be dropped.

The **AT (Add Type)** operation creates a new type, adds it to TSO, and integrates it with the existing lattice. Type creation is supported through regular *subtyping*, which is an operation provided by the primitive model. A *B_new* behavior is defined as part of the meta-system that accepts a collection of supertypes and a collection of behaviors as arguments. The result of creating a new type t as the subtype of types $s_1, \ldots, s_n$ with essential behaviors $b_1, \ldots, b_m$ adds $s_1, \ldots, s_n$ to $P_e(t), b_1, \ldots, b_m$ to $N_e(t)$, and the axioms are recomputed. If no supertypes are specified, T_object is assumed. Due to the axiom of pointedness, which TIGUKAT obeys, the new type t is added to P_e(T_null) because all types are essential supertypes of this base type.

The **DT (Drop Type)** operation drops a given type and removes it from TSO. To drop a type t, the type is removed from **C_type** and from the P_e of all subtypes of t. The axiomatic model does not specifically define a subtypes property, but this can be defined as the inverse operation of the supertypes property. TIGUKAT does define a *B_subtypes* behavior for types, so finding all subtypes of a dropped type is trivial.

In TIGUKAT, there is the restriction that the primitive types of the model (i.e., those in the primitive type system of Figure 2) cannot be dropped. When a type is dropped, the type's associated class and extent are dropped as well. With the use of object migration techniques, the instances can be ported to some other type prior to being dropped in order to preserve their existence. Object migration is outside the scope of this paper.

The **AC (Add Class)** operation creates a class, adds it to CSO, and uniquely associates it with a particular type to manage its extent. The creation of a class allows instances of its associated type to be created.

The **DC (Drop Class)** operation drops the associated class of a type and removes it from CSO. The extent managed by a dropped class is also dropped. As mentioned above, object migration techniques can be used to migrate objects to another class in order to preserve them.

Since explicitly dropping behaviors from a type definition (operation MT-DB) is a schema change, dropping a behavior in its entirety is also a schema change because the behavior may be defined on one or more types.

The **DB (Drop Behavior)** operation drops a given behavior and removes it from BSO. A dropped behavior is dropped from all types that define the behavior as essential. The semantics of this operation follows dropping behaviors from types (operation MT-DB) defined above.

The **MB-CA (Modify Behavior - Change Association)** operation changes the implementation of a behavior by as-

	Operation		
Objects	Add (A)	Drop (D)	Modify (M)
Type (T)	**subtyping**	**type deletion**	**add behavior**(AB) **drop behavior**(DB) **add subtype relationship**(ASR) **drop subtype relationship**(DSR)
Class (C)	**class creation**	**class deletion**	*extent change*
Behavior (B)	*behavior definition*	**behavior deletion**	**change association**(CA)
Function (F)	*function definition*	**function deletion**	*implementation change*
Collection (L)	**collection creation**	**collection deletion**	*extent change*
Other (O)	*instance creation*	*instance deletion*	*instance update*

Table 3: Classification of schema changes.

sociating it with a different function, which could also affect the function's membership in FSO. This is an implementation based schema change that is outside the scope of the high-level axiomatic model. We have defined a complete set of possible implementation changes and conflict resolution procedures for these changes. Details are given in [7]. Conflict resolution of high-level behaviors is based on their semantics, which is a unique description of the behavior and, thus, set operations can be used.

Since changing the association of a function with a behavior is considered a schema change, dropping a function in its entirety is a schema change because the function may be the implementation of a behavior in some type.

The **DF (Drop Function)** operation drops a given function and removes it from FSO. The operation is rejected if the function is associated as the implementation of a behavior in a type that has an associated class.

Collections are heterogeneous, user-defined and managed object groupings and the axiomatic model does not interfere with this flexibility. They are included here for completeness.

The **DL (Drop Collection)** operation drops a given collection and removes it from LSO. Unlike classes, dropping a collection does not drop its members.

The **AL (Add Collection)** operation adds a new empty collection to LSO. Collection addition is collection creation as defined by the primitive model.

The remaining entries in Table 3 represent changes that are not considered part of the schema evolution problem. Creating, dropping, and updating object instances (operations AO, DO, and MO) other than the ones discussed above clearly are operations concerned with the real-world concepts modeled in the objectbase and, therefore, do not have an affect on the schema. Defining a new behavior (operation AB) does not affect the schema because behaviors don't become part of the schema until after they are added as essential behaviors of some type. Defining a new function (operation AF) does not affect the schema because functions don't become part of the schema until after they are associated as the implementation of a behavior defined on some type. Modifying a function (operation MF) does not affect the semantics of the behaviors it may be associated with and, therefore, this operation does not affect the schema. Collections are groupings of objects that are defined and maintained by the user. Modifying a collection involves changing the membership of its extent and changing its membership type. These are operations related to the contents of the collection and, therefore, are not part of the schema evolution problem.

4 Related Work

In recent years, several researchers have addressed the problem of defining schema evolution policies for OBMSs. All of these studies address the issue from the perspective of individual systems. The axiomatic model provided in this paper is unique in this respect. Some systems are described below in relation to the concepts introduced in this paper.

The Orion [1] model is the first system to introduce the invariants and rules approach as a structured way of describing schema evolution in OBMSs. Orion defines a complete set of invariants and a set of twelve accompanying rules for maintaining the invariants over schema changes. The allowed schema changes are classified into several categories, each of which affects different parts of the schema. These changes represent the typical schema modifications allowed in most systems. The changes supported in TIGUKAT are similar to those of Orion, but vary to deal with uniformity, which is not part of Orion. For example, stored properties and computed methods are separate concepts in Orion and need to be handled separately, while in TIGUKAT they are treated uniformly as behaviors and, therefore, a single mechanism suffices for both.

Orion defines eight fundamental operations that are declared as being inclusive of all "interesting" schema changes. The soundness and completeness of these operations are proven. We show how the semantics of the eight operations are represented in the axiomatic model. The Orion terminology of *class*, *subclass*, and *superclass* is used instead of type, subtype, and supertype.

In mapping the Orion class structure to the axiomatic model, P_e represents the superclasses of an Orion class. There is no notion of the minimal superclasses, P, in Orion. The superclasses in Orion are ordered for conflict resolution purposes. The P_e set can easily be ordered for this purpose. There is also no explicit superclass lattice in Orion, but it is implied by the superclass relationships.

In mapping properties, N_e represents the defined or redefined properties of an Orion class. There is no notion of the minimal native, N, or inherited, I, properties in Orion. Inherited properties of a class C in Orion is equivalent to $I(C) - N_e(C)$ in the axiomatic model. The interface, I, of a class has the same meaning in Orion and the axiomatic model. Properties in Orion have names and domains, which are used in conflict resolution. The axiomatic model assumes that properties have a given semantics. Names and

domains can be part of the semantics, which in turn can be used for conflict resolution.

The Axiom of Closure is not explicitly stated in Orion, but is implied by the connected nature of the class structure. The Axiom of Acyclicity, on the other hand, is strictly enforced. Furthermore, the Axiom of Rootedness is obeyed with $\top$ = OBJECT and the Axiom of Pointedness is relaxed since there is no single class as a base.

The eight fundamental operations of Orion and their semantics in terms of the axiomatic model are as follows:

OP1: Add a new property v to a class C: Add v to $N_e(C)$. Perform Orion conflict resolution as necessary. The same operation is performed whether v is an attribute or a method.

OP2: Drop an existing property v from a class C: Drop v from $N_e(C)$. Perform conflict resolution as necessary. The same operation is performed whether v is an attribute or a method.

OP3: Add an edge to make class S a superclass of class C: Add S to the end of ordered $P_e(C)$. Perform conflict resolution as necessary. If the Axiom of Acyclicity is violated, the operation is rejected.

OP4: Drop an edge to remove class S as a superclass of class C: Remove S from $P_e(C)$ unless S is the last superclass of C in which case C is linked to the superclasses of S (i.e., $P_e(S)$). If S is the last superclass of C and S is OBJECT, the operation is rejected. The following algorithm illustrates the procedure:

if $P_e(C) = \{S\}$ **then** // Last superclass of C?
  if S = OBJECT **then** REJECT operation
  else $P_e(C) = P_e(S)$ // Link C to superclasses
else remove S from $P_e(C)$

OP5: Change the ordering of superclasses of a class C: Simply change the ordering of classes in $P_e(C)$. Perform conflict resolution as necessary.

OP6: Add a new class C as the subclass of a class S: Create C and add S to $P_e(C)$. If S is not specified, then S = OBJECT by default. In Orion, additional superclasses can be added to C using OP3.

OP7: Drop an existing class S: For all subclasses C of S, remove S as a superclass of C using OP4.

OP8: Change the name of a class C: Change every occurrence of C in the P_e's of the various classes to the new name.

Since each of the fundamental operations have an equivalent semantics in the axiomatic model, the soundness and completeness of these operations are preserved. Thus, Orion is reducible to the axiomatic model. The reduction of axiomatic model to Orion is not possible since, for example, Orion does not maintain minimal superclasses or native properties of classes. Thus, the axiomatic model subsumes the schema evolution techniques of Orion. In Section 5, we discuss the similarities and differences of schema evolution in TIGUKAT and Orion in terms of their axiomatizations.

Due to space limitations, only a brief discussion of other schema evolution approaches are presented below. These systems define schema evolution policies similar to Orion and they too are reducible to the axiomatic model.

Schema evolution in GemStone [6] 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. Based on published work [6], the GemStone schema changes can be expressed by the axiomatic model.

Skarra and Zdonik [9] define a framework for versioning types in Encore as a support mechanism for evolving type definitions. This work is focussed on dealing with change propagation rather than semantics of change. Their schema evolution operations are similar to Orion and, thus, representable by the axiomatic model.

Nguyen and Rieu [4] discuss schema evolution in the Sherpa model and compare their work to Encore, GemStone, Orion, and their earlier model 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 and, therefore, can be represented by the axiomatic model.

5 Discussion and Comparison

One advantage of the axiomatization presented in this paper is the precision that it introduces to the specification of dynamic schema evolution strategies of various systems. Another important advantage is the possibility of comparing these approaches based on a common specification. In this section, we provide a few important comparisons between TIGUKAT and Orion based on their reduction to the axiomatic model.

In terms of subtyping and property inheritance, TIGUKAT and the axiomatic model are reducible in both directions while only the reduction from Orion to the axiomatic model is possible. One result of this is that the minimal supertypes and minimal native properties cannot be exploited in Orion, which can be useful for the efficiency of the system. For example, to resolve property naming conflicts in a type, it would only be necessary to iterate through the minimal supertypes of that type because any conflicts would be detectable in these supertypes alone. Another use for minimal supertypes is in displaying the type lattice graphically. A user would only need to see the minimal subtype relationships in order to understand the complete functionality of a type.

In comparing the schema operations of the two systems, we focus on the eight fundamental operations identified in Orion. The operations of adding and dropping properties of types (classes) are virtually identical in both systems. The only difference is that TIGUKAT maintains the minimal native properties while Orion does not.

The operation of adding a subtype relationship between two types (an edge between two classes in Orion) is also similar in both systems. The differences are that Orion maintains an ordered supertype list for conflict resolution and TIGUKAT maintains a minimal supertype set.

The operation of dropping a subtype relationship (edge) between two classes is quite different in the two systems. Dropping a series of edges in Orion can produce a different lattice depending on the order in which the edges are dropped. In TIGUKAT, the ordering is irrelevant and the same lattice is produced no matter the order in which they are dropped. Based on the axiomatization of the two systems, the judgement we make here is that the dropping of subtype relationships in TIGUKAT is much simpler and is uniform as compared to Orion.

The operation of adding a type (class) is similar, except that TIGUKAT obeys the Axiom of Pointedness and so the new type is added as an essential supertype of T_null. The operation of dropping a type (class) is different in the two systems. In both, this operation is based on dropping subtype relationships (edges),but the two systems differ in the handling of this operation.

The operation of changing the ordering of classes was introduced in the axiomatization of Orion to deal with conflict resolution. This is an implementation detail that was abstracted out in the axiomatization of TIGUKAT. Similarly, the operation of changing the name of a class is specific to Orion. Again, TIGUKAT deals more abstractly with the notion of references (which act as names) to objects with unique identity. For example, the act of adding s to $P_e(t)$ does not mean "add the name s to the set $P_e(t)$", instead it means "add a reference to the object identified by s to the set $P_e(t)$". There may be two different references (with different names) that refer to the same object. There is no notion of renaming objects in TIGUKAT because objects are created with a unique, immutable object identity.

From the above comparison, TIGUKAT and Orion are quite similar with respect to lattice and property inheritance schema operations. TIGUKAT is a uniform model and this is reflected in its handling of schema evolution. Additional differences lie in TIGUKAT's separation of types from their extents and behaviors from their implementations (functions). This allows the axiomatization of TIGUKAT to consider types without considering their extents and to consider behaviors without considering their implementations.

6 Conclusions and Future Work

There are two issues in schema evolution: the *semantics of change* issue describes the possible schema changes and the *change propagation* issue describes how schema changes affect object instances. In this paper, we only address the semantics of change issue and do not discuss change propagation. We have informally described change propagation in TIGUKAT [7]. However, a formal axiomatic model for change propagation and its integration with the model proposed here is under development.

An axiomatization of dynamic schema evolution provides a foundation for describing the schema evolution policies of different systems using a single, underlying framework. The sound and complete axiomatic model proposed in this paper has the power to provide this basis to object models supporting subtyping and property inheritance. Furthermore, the common description facility offered by the model provides a means of better comparing the schema evolution facilities of the various systems. To illustrate the power of the axiomatic model, the schema evolution policies of TIGUKAT and Orion were reduced to the model and compared.

One area of great interest is the implementation of schema evolution in TIGUKAT based on the axiomatic model. A prototype implementation of the core object model is complete [3] and its extension with efficient algorithms for schema evolution is currently under development. The completion of this task will provide the necessary empirical evidence of its performance characteristics. Also of interest is a formal complexity analysis of our implementation techniques, which will provide the theoretical evidence of performance.

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. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 311–322, May 1987.

[2] I. Goralwalla and M.T. Özsu. Temporal Extensions to a Uniform Behavioral Object Model. In *Proc. of the 12th Int'l Conf. on Entity–Relationship Approach*, pages 115–127, December 1993.

[3] B. Irani. Implementation Design and Development of the TIGUKAT Object Model. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report TR93-10.

[4] G.T. Nguyen and D. Rieu. Schema Evolution in Object-Oriented Database Systems. *Data & Knowledge Engineering*, 4:43–67, 1989.

[5] M.T. Özsu, R.J. Peters, D. Szafron, B. Irani, A. Lipka, and A. Muñoz. TIGUKAT: A Uniform Behavioral Objectbase Management System. *The VLDB Journal (Special Issue on Persistent Object Systems)*, January 1995. In press.

[6] D.J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proc. OOPSLA Conf.*, pages 111–117, October 1987.

[7] R.J. Peters. *TIGUKAT: A Uniform Behavioral Objectbase Management System*. PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1994. Available as University of Alberta Technical Report TR94-06.

[8] R.J. Peters and M.T. Özsu. Reflection in a Uniform Behavioral Object Model. In *Proc. of the 12th Int'l Conf. on Entity–Relationship Approach*, pages 37–49, December 1993.

[9] A.H. Skarra and S.B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *Proc. OOPSLA Conf.*, pages 483–495, September 1986.