An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems

RANDAL J. PETERS
University of Manitoba
and
M. TAMER ÖZSU
University of Alberta

The *schema* of a database system consists of the constructs that model the entities of data. *Schema evolution* is the timely change of the schema and the consistent management of these changes. *Dynamic schema evolution* (DSE) is the management of schema changes while a database management system is in operation. DSE is a necessary facility of objectbase systems (OBSs) because of the volatile application domains that OBSs support. We propose a sound and complete axiomatic model for DSE in OBSs that supports the fundamental concepts of object-oriented computing such as subtyping and property inheritance. The model can infer all schema relationships from two identified input sets associated with each type called the *essential supertypes* and *essential properties*. These sets are typically specified by schema designers, but can be automatically supplied within an OBS. The inference mechanism performed by the model has a proven termination.

The axiomatic model is a formal treatment of DSE in OBSs, which distinguishes it from other approaches that informally define a number of schema invariants and the rules that enforce them. An informal approach leads to multiple DSE mechanisms because of the differences in object models and the choices made by system designers. The lack of a common object model makes comparison of OBSs more difficult. The axiomatic model provides a solution for DSE in OBSs by serving as a common, formal underlying foundation for describing DSE of existing systems, which makes comparison of these systems much easier. A design space for OBSs based on the inclusion/exclusion of axioms is developed and can be used to classify, compare, and differentiate the features of OBSs. To test the expressibility of the model, the DSE of several OBSs are reduced to the axiomatic model and compared.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*Schema and subschema*

General Terms: Algorithms, Design, Management, Theory

Additional Key Words and Phrases: Dynamic schema evolution, object database management systems

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 the schema changes frequently and dynamically. For example, in an engineering design application many components of an overall design go through several modifications before a final product design is produced. These kinds of changes require modifications to the way in which data components are

This research is supported by the Natural Science and Engineering Research Council (NSERC) of Canada under operating grants OGP0173259 and OGP0951. This work was performed as part of the first author's Ph.D. dissertation at the University of Alberta.

Authors' Addresses: Randal J. Peters, Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada R3T 2N2, Email: randal@cs.umanitoba.ca; M. Tamer Özsu, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1, Email: ozsu@cs.ualberta.ca

modeled (i.e., changes to the schema). Furthermore, the evolutionary characteristic of the applications require sophisticated mechanisms for managing changes in schema and ensuring the overall consistency of the system. These mechanisms constitute the *dynamic schema evolution* component of database management systems.

The *schema* (or *meta-information*) of an objectbase system (OBS) is the information that describes the structure and operations of object instances stored in an objectbase and managed by an objectbase management system (OBMS). For example, the *types* (or classes) of an object model, together with their *properties* (e.g., attributes, methods, behaviors) form the schema of an OBS. *Dynamic schema evolution* (DSE) is the process of applying changes to the schema in a consistent fashion and propagating these changes to the object instances while the OBS is in operation. The majority of published approaches to DSE informally define a number of invariants and specify a set of rules for maintaining them. Invariants describe constraints on the schema and relationships between schema elements, while the rules define procedures that enforce the invariants. Orion [Banerjee et al. 1987], for example, defines five invariants and twelve rules. Other systems differ in the number and semantics of the invariants and rules they define. With the lack of a formal basis, the management of DSE becomes *ad hoc* and comparing different approaches can be difficult.

In this paper, we propose a sound and complete axiomatic model for DSE in OBSs. The main benefit of the model is the formalization of DSE characteristics into a well-defined set of axioms. The axioms automatically maintain complex schema relationships and properties from two input sets associated with each type in a schema. The elements of these sets can be provided by the user, schema designer, system, or a combination of sources. One set is called the *essential supertypes* and contains the types that must be maintained as supertypes of a type for as long as it is consistently possible. The other set is called the *essential properties* and contains the properties that must be maintained in the type for as long as it is consistently possible. The correct properties and relationships within the schema are automatically derived by the axiomatic model using the essential supertypes and essential properties as a basis. The derivations performed by the axiomatic model have a proven soundness, completeness, and termination. The inclusion/exclusion of axioms in the model leads to a design space that categorizes OBSs into *object-based*, *type-based*, and *object-oriented* systems. The last category is further refined into several distinct subcategories that vary in functionality and expressiveness. To illustrate the power and practical usefulness of the model, the DSE operations of several existing OBSs are reduced to the axiomatic model and compared within this common framework.

In recent years, researchers have addressed the problem of defining DSE policies for OBSs. These studies approach the issue from the perspective of individual systems. The axiomatic model is unique in this respect in that it captures and formalizes the salient features of DSE in OBSs and can be adopted as a common underlying foundation of individual systems.

In order to clarify and illustrate the expressiveness of the axiomatic model, the DSE policies of the Tigukat[1] OBS [Özsu et al. 1995; Peters 1994] are presented as an example. DSE in Tigukat is reduced to the axiomatic model and compared to the axiomatization of Orion. Tigukat is being developed at the Laboratory for Database Systems Research

[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.

of the University of Alberta. There are complementary research and prototyping efforts addressing dynamic schema evolution and view management at the Advanced Database Systems Laboratory of the University of Manitoba. An identifying characteristic of Tigukat is its uniform, extensible object model that is capable of supporting database services within a single underlying framework. In keeping with Tigukat's modeling capability, the DSE policies are defined as uniform extensions to the base system. The schema changes advocated in Tigukat are similar to those of Orion, but vary to deal with complete model uniformity, which is not fully supported in Orion.

The main contributions of the paper are as follows:

(1) It introduces an axiomatic model for DSE in OBSs with the following benefits:
   (a) the model is a formal specification of DSE in OBSs,
   (b) the model has a proven soundness, completeness, and termination for the derivations performed,
   (c) the model is powerful enough to express DSE of existing OBSs, and
   (d) the model can be used in practice by serving as a common foundation for characterizing and comparing DSE of various systems.

(2) It develops a design space for OBSs based on the inclusion/exclusion of axioms.

(3) It presents the uniform, extensible DSE policies of the Tigukat OBS.

(4) It formalizes and compares the DSE policies of Tigukat and Orion through their reduction to the axiomatic model.

The remainder of the paper is organized as follows. The axiomatic model of DSE in OBSs is defined in Section 2, together with the proofs of soundness, completeness and termination. A design space for OBSs based on the inclusion/exclusion of axioms is presented in Section 3. An overview of the Tigukat object model and the definition of DSE in Tigukat in terms of the axiomatic model is presented in Section 4. The reduction of Orion to the axiomatic model is presented in Section 5, along with a comparison of the axiomatization of Tigukat and Orion. Related work is discussed in Section 6 with a particular focus on how systems propagate schema changes to object instances. Finally, Section 7 contains concluding remarks and a discussion of future work.

2. AXIOMATIC MODEL OF DSE

Typical schema changes in an OBS include adding and dropping types, adding and dropping sub/supertype relationships between types, and adding and dropping properties of a type. A fairly extensive classification of widely accepted schema changes is given as part of Orion [Banerjee et al. 1987]. 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 (i.e., the effects on the schema);

(2) **Change propagation:** This refers to the method of propagating the schema change to the underlying objects (i.e., to the existing instances).

For the first problem, the basic approach is to define a number of invariants that must be satisfied by the schema and then to define rules and procedures for maintaining these invariants for each possible schema change. The invariants, and the rules for ensuring their compliance, depend on the underlying object model. Since object models differ, the DSE policies of various systems based on this approach differ as well. Furthermore, the lack of a formal semantics makes systems difficult to compare. The axiomatic model captures the underlying mechanism of DSE and is powerful enough to describe the semantics of change in various systems so that they can be compared.

For the second problem, 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 object representations are lost. *Screening*, *conversion*, and *filtering* are techniques for defining when and how coercion takes place. The scope of this paper is limited to the specification of an axiomatic model for the *semantics of change*. However, the handling of change propagation in various systems is discussed in Section 6.

In this section, we present a formal axiomatic model of DSE in OBSs to deal with the semantics of change problem. This is followed by proofs of the model's termination, soundness, and completeness.

2.1 Axiomatization of Schema Changes

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. We use the term *property* generically as encompassing all of these. 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. We use *type* to denote the construct that defines object properties and *class* to denote the type extent.

Subtyping is a facility of object models that allows types to be built incrementally from other types. We use $\preceq$ to represent a reflexive, transitive, and antisymmetric *subtype relationship* where $t \preceq s$ means 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 in a diagram. 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*) $\mathcal{L} = \langle \mathcal{T}, \leq \rangle$ consists of a set of types $\mathcal{T}$ together with a partial order $\leq$ of the elements of $\mathcal{T}$ based on the subtype relationship ($\preceq$). The term *lattice* (or *semi-lattice*) are commonly used in object-oriented literature to denote a typing structure that supports multiple inheritance. Although 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, 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* with types as vertices and subtype relationships as directed edges.

The notation for the axiomatic model is shown in Table I. 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[2] in Figure 1 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

[2]The prefix "T_ " in Figure 1 indicates a type.

application environments such as those mentioned in the introduction.

Term	Description
$\mathcal{T}$	The set of all types of a system.
$\mathcal{L}$	The type lattice of a 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.
$PL(t)$	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.
$\alpha_x(f, \mathcal{T}')$	Apply-all operation.

Table I. Notation for axiomatic model.

The set of types $\mathcal{T}$ represents all the types in the system on which DSE operations are performed. The set consisting of all types shown in Figure 1 forms $\mathcal{T}$ in this example. A type lattice $\mathcal{L}$ is formed from the set $\mathcal{T}$ and the subtype relationships maintained by the *immediate supertypes*, $P(t)$, for all types $t \in \mathcal{T}$. 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.

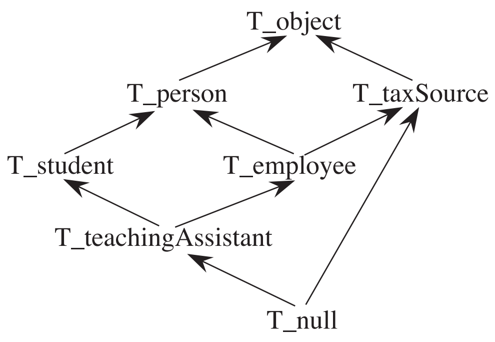

Fig. 1. Simple type lattice.

DEFINITION 1. *Immediate Supertype:* Given two distinct types $s, t \in \mathcal{T}$ where $t \preceq s$, s is an *immediate supertype* of t (equivalently, t is an *immediate subtype* of s) if and only if there does not exist an $x \in \mathcal{T} - \{s, t\}$ such that $t \preceq x$ and $x \preceq s$. □

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.

The *essential supertypes*, $P_e(t)$, are the types identified as being essential to the construction and existence 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 immediate supertypes are essential.

An OBS can impose constraints that force every 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, Tigukat defines a primitive root type "T_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$\}$. Orion also defines a primitive root type "OBJECT" and could operate similarly. In addition to system supplied types, the schema designer can provide elements of $P_e(t)$ that represent the essential inheritance constructs of a particular application domain. 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 1, assume the system provides the root type T_object and assume the schema designer has specified the remaining essential supertypes of T_teachingAssistant as:

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

If 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 is not declared as essential.

The *supertype lattice* $\mathcal{L}_t = \langle PL(t), \leq_t \rangle$ 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 $\leq$ then $x \preceq y$ in $\leq_t$.

DEFINITION 2. *Supertype Lattice Types:* Given type $t \in \mathcal{T}$, the supertype lattice types of t is the set of types $PL(t) \subseteq \mathcal{T}$ defined as $PL(t) = \{s \in \mathcal{T} \mid t \preceq s\}$. □

For example, if we let $t =$ T_employee, then the supertype lattice types of t is given as:

$$PL(\text{T\_employee}) = \{\text{T\_employee}, \text{T\_person}, \text{T\_taxSource}, \text{T\_object}\}.$$

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 be defined by other types not in a subtype relationship with the former type. 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.

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. We consider conflict resolution at two levels: *behavioral* and *functional*.

Behavioral resolution resolves conflicts based on the semantics of properties. With an accepted semantic specification of properties and a notion of semantic equivalence, behavioral resolution is fairly simple to support. For example, the semantics of "name" in T_person and T_taxSource are equivalent and, thus, the semantics of "name" in T_employee should match this semantics. In effect, there is no conflict between "name" at the behavioral level because the semantics of the "name" is equivalent in all places where it appears.

Currently, the most widely used notion of property semantics are *signatures*. In general, a *signature* consists of a name, a list of argument types, and a result type. The name is used to apply the property to an object (e.g., sending a message to the object) with a list of arguments. The result corresponds to the result type given in the signature. Approaches such as co-variance and contra-variance can be used to identify semantic equivalence. Behavioral resolution relies on an acceptable semantic specification for properties, but once defined, it is fairly simple to support. Signatures provide a first step in semantic specification, but in general are inadequate for most applications. A reasonably sufficient semantics for type properties is an open research problem.

Functional resolution resolves conflicts based on the implementations of property semantics. The semantics of a property must be implemented at each type that defines it. Implementation can be in the form of stored attributes or computed methods. A single semantic property may be implemented differently in the types where it is defined. For example, "name" may be implemented as a length encoded string in T_person and a null terminated string in T_taxSource. Furthermore, an "age" property in T_person and T_taxSource may be implemented as a stored attribute in one type and a computed method in the other. Although "name" and "age" are semantically equivalent in the two types, they are implemented differently. In this case, the implementation of "name" and "age" in T_employee is not clear. The typical choices are to select the implementation in T_person, the implementation in T_taxSource, or to completely redefine the implementation. The focus of this paper is on DSE at the semantic (or behavioral) level rather than the implementation (or functional) level and, thus, we do not discuss functional conflict resolution techniques. For our purposes, we assume that when a functional conflict occurs the schema designer is asked to resolve it by choosing an implementation from the list of conflicting ones or redefining the implementation entirely.

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. In contrast, the native properties of T_employee are those defined on T_employee, but not defined on any of T_person, T_taxSource, or T_object.

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 $N_e(t)$ specification for each type t. 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 in both the definition essential properties, $N_e(t)$, and essential supertypes, $P_e(t)$.

Schema evolution and essential property specifications may require inherited properties of a type to be adopted as native properties 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 is deleted, then the "taxBracket" property would be adopted by T_employee as a native property. The derivations by the axioms automatically make these adjustments in the type lattice.

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.

Table II depicts the axioms of DSE using the various types and properties in Table I. The derivation of the various sets in the axioms are based on the $P_e(t)$ and $N_e(t)$ terms. All DSE operations can be handled through these two terms, which eases the burden on the schema designer and makes the system more manageable. However, 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 $N_e(t)$. Changes to these two components are fundamental to the evolution of the schema. 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 DSE in OBSs that support subtyping and property inheritance.

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, or that only the immediate ones are essential. Current systems vary in the semantics defined for the notions of subtyping, inheritance, and nativeness. Our formalization of these concepts 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.

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 would be 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 essential properties and types. The 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 $t \preceq s$ between types s and t always includes s in $P_e(t)$, but s is added to $P(t)$ if and only if s is not already a supertype of t (i.e., $s \notin PL(t)$). In this way, $N(t)$ maintains the minimum properties that must be defined in t and $P(t)$ maintains the minimum supertypes of t. This minimality is beneficial 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 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 execution and they remain constant throughout the apply-all operation.

The semantics of apply-all will let x range over the elements of $\mathcal{T}'$ and for each type bound to x, f is evaluated and the answer is included in the final result set. If $\mathcal{T}'$ is empty, the empty set is returned. In functional notation, the α operation applies the lambda function $\lambda x.f$ to every element of $\mathcal{T}'$ and returns a set containing the results.

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

Table II. Axiomatization of subtyping and property inheritance for DSE in OBSs.

Table II summarizes the axiomatization of subtyping and property inheritance for DSE in OBSs. Each α apply-all operation in the table returns a set of sets as its result. The union operator immediately preceding each α operator performs an *extended union* over the members of the result set, which has the effect of unnesting the result into a single level set of types. We define the extended union of the empty set as the empty set. Each axiom in Table II is discussed below:

Axiom of Closure. All types in $\mathcal{T}$ have supertypes in $\mathcal{T}$, giving closure to $\mathcal{T}$. Note that the root has no essential supertypes, but this empty set is a proper subset of $\mathcal{T}$.

Axiom of Acyclicity. There are no cycles in the type lattice formed from $\mathcal{T}$ and its partial order. This axiom disallows any element of $\mathcal{T}$ from appearing in the supertype lattice types of any of its immediate 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 transitively through some other type. This axiom provides a means to automatically instantiate the immediate supertypes of a 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 all types in the supertype lattices of its immediate supertypes. This axiom provides a means to automatically instantiate the supertype lattice types of a 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 means to automatically instantiate the interface of a type.

Axiom of Nativeness. The native properties of a type are the subset of the essential properties that are not inherited. This axiom provides a means to automatically instantiate the native properties of a type based on the essential properties of that type.

Axiom of Inheritance. The inherited properties of a type is the union of the interfaces of its immediate supertypes. This axiom provides a means to automatically instantiate the inherited properties of a type.

There are several simplifications that can be made to the axioms in order to reduce the amount of mutual recursion among them. Recall that $P(t) \subseteq P_e(t)$, which can be easily verified by observing the set difference operation in the Axiom of Supertypes. This means that $P_e(t)$ contains at least the information of $P(t)$. We also point out that all types in $P_e(t) - P(t)$ are supertypes of at least one type in $P(t)$. That is, they are reachable from t, transitively, through a type in $P(t)$. The significance of this point is that any properties defined on types in $P_e(t) - P(t)$ are inherited by at least one type in $P(t)$. This means that $P_e(t)$ does not contain any more property information than $P(t)$. We can, therefore, declare that $P_e(t)$ can be safely substituted for $P(t)$ wherever supertype lattices and property inheritance is concerned. Thus, Axioms 2, 6, and 9 can be expressed as follows:

2*.	Axiom of Acyclicity*	$\forall t \in \mathcal{T}, t \notin \bigcup \alpha_x(PL(x), P_e(t))$
6*.	Axiom of Supertype Lattice*	$\forall t \in \mathcal{T}, PL(t) = \bigcup \alpha_x(PL(x), P_e(t)) \cup \{t\}$
9*.	Axiom of Inheritance*	$\forall t \in \mathcal{T}, H(t) = \bigcup \alpha_x(I(x), P_e(t))$

Furthermore, by substituting the equation for $N(t)$ in the formula for $I(t)$ in Axiom 7 and then reducing, a simpler equation based on $N_e(t)$ rather than $N(t)$ can be derived for $I(t)$. The derivation is as follows:

7*. Axiom of Interface*
$$\begin{aligned} I(t) &= N(t) \cup H(t) \\ &= (N_e(t) - H(t)) \cup H(t) \\ &= N_e(t) \cup H(t) \end{aligned}$$

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, 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 would be lost in T_teachingAssistant, except for those declared in N_e(T_teachingAssistant). Moreover, if T_employee is now 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 again for the declared essential properties).

The axiomatic model scales to more complex type lattices such as those required by geo-information systems, CAD/CAM, multimedia systems, and so on. This is supported by the ability to view each type in isolation and only decide on its essential supertypes and properties, while the model provides the automatic derivation of the complete subtype and property relationships.

2.2 Termination, Soundness, and Completeness

In this section, we prove the *termination*, *soundness*, and *completeness* of the axioms. Termination guarantees that the axioms eventually complete all derivations performed. Termination is directly related to the structure of the type lattice and the specifications of essential supertypes and essential properties. The Axiom of Acyclicity guarantees that there are no cycles in the type lattice – the remaining axioms traverse the lattice from any given type towards the root where they eventually terminate and "bottom out" on the essential supertype and property sets. Soundness guarantees that *only valid* schema objects, properties, and relationships are derived by the axioms. Soundness is important to ensure that no erroneous results are produced by the axioms. On the other hand, completeness guarantees that *all valid* schema objects, properties, and relationships are derived. Completeness is important to ensure that nothing is missed by the axioms.

We define $\overrightarrow{\mu\nu}$ as the *maximal path length* (MPL) in the type lattice from type μ to type ν. The maximal path length is the longest possible path of immediate supertype links from μ to ν. For example, in Figure 1 if we let μ = T_null and ν = T_object, then $\overrightarrow{\mu\nu} = 4$. Furthermore, we define $\mathcal{T}^n \subseteq \mathcal{T}$ as $\mathcal{T}^n = \{\mu \in \mathcal{T} \mid \overrightarrow{\mu\top} \leq n\}$. That is, $\mathcal{T}^n$ is the complete set of types whose MPL to the root type is less than or equal to n.

A dependency graph of the derivations performed by the simplified set of axioms is shown in Figure 2. For reference, axiom numbers are shown in square brackets. The calculation of an axiom at the tail of a directed edge uses (or depends on) the axiom at the head of that edge. A dotted edge indicates that the calculation of the axiom at the tail uses the axiom at the head on supertypes of the type t. The semantics of the dotted edges are that they traverse the type lattice towards the root. The significance of this graph is that it illustrates two recursive calculations (denoted by cycles in the graph) of the axioms. The one cycle is between $I(t)$ and $H(t)$, and the second cycle is on $PL(t)$. These must be carefully considered when proving the termination of the axioms.

$$\begin{tikzcd}
 & & Ne(t) & Pe(t) & \\
I(t)_{[7*]} \arrow[urr] \arrow[drr] & N(t)_{[8]} \arrow[ur] \arrow[dr] & & & \\
 & & H(t)_{[9*]} \arrow[uur] \arrow[ull, dashed, bend left] & P(t)_{[5]} \arrow[uu] \arrow[r, dashed] & PL(t)_{[6*]} \arrow[uul] \arrow[loop right, dashed]
\end{tikzcd}$$

Fig. 2. Axiom dependency graph.

Intuitively, the termination of the axioms is mostly straightforward from the dependency graph. The cycles in the graph are the only factors that represent the potential for non-termination. If we prove the termination of each cycle, then we prove the termination of the axioms. In the $[I(t), H(t)]$ cycle there is a dotted edge from $H(t)$ to $I(t)$, which represents a "shift up" in the type lattice. That is, $I(t)$ uses $H(t)$ in its calculation and $H(t)$ uses $I(x)$ where the x's are all supertypes of t. Subsequently, the calculation of $I(x)$ uses $H(x)$, which uses $I(y)$ where the y's are all supertypes of x. Following this pattern leads to the root type $\top$, which has no supertypes and terminates the calculation.

Termination of the $PL(t)$ cycle is proven similarly. The calculation of $PL(t)$ uses $PL(x)$ where the x's are all supertypes of t, and so on. This again leads to the root type which has no supertypes and terminates the recursion. There are several opportunities for optimization in an actual implementation. For example, the inherited properties could be stored in each type to avoid recursion. This section is only concerned with theoretical proofs of the axiomatic model and does not imply an implementation strategy. The formal proofs of termination, soundness, and completeness follow.

THEOREM 1. The schema evolution axioms are terminating.

Proof: Assume $P_e(t)$ and $N_e(t)$ are terminating by being fixed sized sets. Since these are typically supplied by the schema designer and through system constraints, it is reasonable to assume these are of fixed size.

Axioms 1, 2*, 3, and 4 assert a condition on the types $\mathcal{T}$. If the components they rely on are terminating, then they are terminating as well. Axiom 1 is terminating because $P_e(t)$ is a fixed set. We now turn attention to proving the termination of Axioms 5, 6*, 7*, 8, and 9* on which Axioms 2* through 4 depend. Termination of Axioms 5, 6*, 7*, 8, and 9* is proved by induction on MPLs in the type lattice.

Basis. Root type. We show that Axioms 5, 6*, 7*, 8, and 9* are terminating when $t = \top$, the root type.

Axiom 5. Since $P_e(\top) = \{\ \}$, the α operation terminates with $\{\ \}$ and, thus, $P(\top)$ terminates.

Axiom 6.* Since $P_e(\top) = \{\ \}$, the α operation terminates with $\{\ \}$ and, thus, $PL(\top)$ terminates.

Axiom 9.* Since $P_e(\top) = \{\ \}$, the α operation terminates with $\{\ \}$ and, thus, $H(\top)$ terminates.

Axiom 8. Since $H(\top)$ terminates and $N_e(\top)$ is fixed, $N(\top)$ terminates.

Axiom 7.* Since $H(\top)$ terminates and $N_e(\top)$ is fixed, $I(\top)$ terminates.

Induction. Assume Axioms 5, 6*, 7*, 8, and 9* are terminating $\forall t \in \mathcal{T}^n$. Choose $\nu \in \mathcal{T}^{n+1}$ such that $\overrightarrow{\nu\top} = n + 1$. We now show that Axioms 5, 6*, 7*, 8, and 9* are terminating for ν.

Axiom 5. $P_e(\nu)$ is a finite set of types and $PL(x)$ is terminating $\forall x \in P_e(\nu)$ because $x \in \mathcal{T}^n$. Therefore, the α operation is terminating and, thus, $P(\nu)$ terminates.

Axiom 6.* Again, since $P_e(\nu)$ is finite and $PL(x)$ is terminating $\forall x \in P_e(\nu)$, the α operation is terminating and, thus, $PL(\nu)$ terminates.

Axiom 9.* Similarly, since $P_e(\nu)$ is finite and $I(x)$ is terminating $\forall x \in P_e(\nu)$ because $x \in \mathcal{T}^n$, the α operation is terminating and, thus, $H(\nu)$ terminates.

Axiom 8. Since $H(\nu)$ terminates and $N_e(\nu)$ is fixed, $N(\nu)$ terminates.

Axiom 7.* Since $H(\nu)$ terminates and $N_e(\nu)$ is fixed, $I(\nu)$ terminates.

The proof follows by induction.

QED. Since Axioms 2* through 4 rely solely on terminating Axioms 5 through 9* and the fixed set $P_e(t)$, they are terminating. □

THEOREM 2. The schema evolution axioms are sound.
Proof: Assume $P_e(t)$ and $N_e(t)$ are sound. Since these are typically supplied by the schema designer and through system constraints, it is important that these be vacuously sound.

Axioms 1, 2*, 3, and 4 assert a condition on the types $\mathcal{T}$. If the components they rely on are sound, then they are sound as well. Axiom 1 is sound because $P_e(t)$ is a sound set for each type $t \in \mathcal{T}$. We now turn attention to proving the soundness of Axioms 5, 6*, 7*, 8, and 9* on which Axioms 2* through 4 depend.

Axiom 5. For $P(t)$ to be unsound we must have a type in $P(t)$ that is not an immediate supertype of t, but instead is transitively linked to t through some other type. Formally, we must have a type $s \in P(t)$ such that there exists a distinct type r such that $t \preceq r \preceq s$. Observe that in order to satisfy this transitive relationship either $r \in P_e(t)$ or there exists a type $v \in P_e(t)$ such that $t \preceq v \preceq r$. Choose r in such a way that $r \in P_e(t)$. Due to the set difference operation, in order for s to have any chance in being included as an element of $P(t)$ we must have $s \in P_e(t)$.

Now, when the x variable in the α operation is bound to r we have $PL(r) \cap P_e(t) - \{r\}$ in the right hand side of Axiom 5. Since $r \preceq s$ we must have $s \in PL(r)$ and because $s \in P_e(t)$ the result of this equation and, hence, the extended union over all other α opertions, is a set containing s. However, the set difference with minuend $P_e(t)$ removes s from the result and we have $s \notin P(t)$. We have shown that the unsound type s cannot exist in $P(t)$. Thus, for all types $s \in P(t)$ we must have $s \in P_e(t)$ such that there does not exist a distinct type r such that $t \preceq r \preceq s$. In other words, every type $s \in P(t)$ is an immediate supertype of t. This satisfies the definition of immediate supertypes and $P(t)$ is sound.

Axiom 8. For $N(t)$ to be unsound we must have a property in $N(t)$ that is not defined natively, but instead is inherited. Formally, we must have a property $p \in N(t)$ such that $p \in H(t)$ as well. Due to the set difference operation, in order for p to have any chance in being included as an element of $N(t)$ we must have $p \in N_e(t)$. However, if we have both $p \in N_e(t)$ and $p \in H(t)$ then we must have $p \notin N(t)$. We have shown that the unsound property p cannot exist in $N(t)$. Thus, for all properties $p \in N(t)$ we must have $p \in N_e(t)$ and $p \notin H(t)$. That is, all native properties are essential and not inherited. This satisfies the definition of nativeness and $N(t)$ is sound.

*Axiom 9**. The proof that the Axiom of Inheritance is sound is based on induction of MPLs.

Basis. *Root type*. We show that $H(\top) = \bigcup \alpha_x(I(x), P_e(\top))$ is sound. Since $P_e(\top) = \{\ \}$, the α operation results in the empty set and, thus, $H(\top) = \{\ \}$, which is sound.

Induction. Assume $H(t)$ is sound $\forall t \in \mathcal{T}^n$. Choose $\nu \in \mathcal{T}^{n+1}$ such that $\overrightarrow{\nu\top} = n+1$. We now show that $H(\nu)$ is sound. Note that $P_e(\nu)$ is sound and $\forall x \in P_e(\nu)$, $N(x)$ is sound and $H(x)$ is sound because $x \in \mathcal{T}^n$. Each $I(x)$ set is sound because it is the union of two sound sets, namely $N(x)$ and $H(x)$, and these are the only contributing factors to $I(x)$. Now, since $H(\nu)$ is the extended union over the sound $I(x)$ sets, it is sound. The proof follows by induction.

*Axiom 7**. The set $I(t)$ is sound because it is the union of two sound sets, namely $N_e(t)$ and $H(t)$, and these are the only contributing factors to $I(t)$.

*Axiom 6**. The proof that the Axiom of Supertype Lattice is sound is based on induction of MPLs.

Basis. Root type. We show that $PL(\top) = \bigcup \alpha_x(PL(x), P_e(\top)) \cup \{\top\}$ is sound. Since $P_e(\top) = \{\ \}$, the α operation results in the empty set and, thus, $PL(\top) = \{\top\}$, which is sound.

Induction. Assume $PL(t)$ is sound $\forall t \in \mathcal{T}^n$. Choose $\nu \in \mathcal{T}^{n+1}$ such that $\overrightarrow{\nu\top} = n + 1$. We now show that $PL(\nu)$ is sound. Note that $P_e(\nu)$ is sound and $P_e(\nu) \subseteq \mathcal{T}^n$ by definition of subtyping and MPLs. Therefore, $\forall x \in P_e(\nu)$, $PL(x)$ is sound because $x \in \mathcal{T}^n$. The extended union over the sound $PL(x)$ sets results in a sound set and the union of this result with $\{\nu\}$ produces a sound result for $PL(\nu)$. The proof follows by induction.

QED. Since Axioms 2* through 4 rely solely on sound Axioms 5 through 9* and the sound set $P_e(t)$, they are sound. □

THEOREM 3. The schema evolution axioms are complete.

Proof: Assume $P_e(t)$ and $N_e(t)$ are complete. Since these are typically supplied by the schema designer and through system constraints, it is important that these be vacuously complete.

Axioms 1, 2*, 3, and 4 assert a condition on the types $\mathcal{T}$. If the components they rely on are complete, then they are complete as well. Axiom 1 is complete because $P_e(t)$ is a complete set for each type $t \in \mathcal{T}$. We now turn our attention to proving the completeness of Axioms 5, 6*, 7*, 8, and 9* on which Axioms 2* through 4 depend.

*Axioms 5 and 6**. The proof of completeness for axioms $P(t)$ and $PL(t)$ are done in parallel and are based on induction of MPLs.

Basis. Root type. We show that $P(\top) = P_e(\top) - \bigcup \alpha_x(PL(x) \cap P_e(\top) - \{x\}, P_e(\top))$ and $PL(\top) = \bigcup \alpha_x(PL(x), P_e(\top)) \cup \{\top\}$ are complete. Since $P_e(\top) = \{\ \}$, then $P(\top) = \{\ \}$, which is complete. Since $P_e(\top) = \{\ \}$, the α operation in $PL(\top)$ results in the empty set and, therefore, $PL(\top) = \{\top\}$, which is complete. These satisfy the definitions of immediate supertypes and supertype lattice.

Induction. Assume $P(t)$ and $PL(t)$ are complete $\forall t \in \mathcal{T}^n$. Choose $\nu \in \mathcal{T}^{n+1}$ such that $\overrightarrow{\nu\top} = n + 1$. We now show that $P(\nu)$ and $PL(\nu)$ are complete.

$P(\nu)$ is complete because $P_e(\nu)$ is complete and the subtrahend removes all the non-immediate supertypes of ν resulting in a complete set of direct supertypes $P(\nu)$.

To prove $PL(\nu)$ is complete, note that $P_e(\nu)$ is complete and $\forall x \in P_e(\nu), PL(x)$ is complete because $x \in \mathcal{T}^n$. Now, the extended union over the complete $PL(x)$ sets together with the union of $\{\nu\}$ results in a complete set of supertype lattice types $PL(\nu)$. The proof follows by induction.

Axioms 7 and 9**. Completeness of $I(t)$ and $H(t)$ are proved in parallel based on induction of MPLs.

Basis. Root type. We show that $H(\top)$ and $I(\top)$ are complete.

To show that $H(\top) = \bigcup \alpha_x(I(x), P_e(\top))$ is complete note that $P_e(\top) = \{\ \}$ resulting in the empty set for the α operation. Thus, $H(\top) = \{\ \}$, which is complete. Furthermore, $I(\top) = N_e(\top) \cup H(\top)$ is complete because it is the union of two complete sets $N_e(\top)$ and $H(\top)$. These satisfy the definitions of inherited properties and type interface.

Induction. Assume $H(t)$ and $I(t)$ are complete $\forall t \in \mathcal{T}^n$. Choose $\nu \in \mathcal{T}^{n+1}$ such that $\overrightarrow{\nu\top} = n + 1$. We now show that $H(\nu)$ and $I(\nu)$ are complete.

To prove $H(\nu)$ is complete, note that $P_e(\nu)$ is complete and $I(x)$ is complete $\forall x \in P_e(\nu)$ because $x \in \mathcal{T}^n$. The extended union over the complete $I(x)$ sets results in the complete set $H(\nu)$.

$I(\nu)$ is complete because it is the union of two complete sets, namely $N_e(\nu)$ and $H(\nu)$, and these are the only contributing factors to $I(\nu)$. The proof follows by induction.

Axiom 8. Assume $N(t)$ is not complete. This means there exists a native property p defined on t such that $p \notin N(t)$. Since p is defined on t it is supported in the interface of t and we must have $p \in I(t)$. Moreover, since p is a native property it is not inherited from a supertype of t and we must have $p \notin H(t)$. In order for $p \in I(t)$ and $p \notin H(t)$ to be satisfied, our initial assumption must be incorrect and we must have $p \in N(t)$ making $N(t)$ complete.

QED. Since Axioms 2* through 4 rely solely on complete Axioms 5 through 9* and the complete set $P_e(t)$, they are complete. □

3. AXIOMATIC DESIGN SPACE FOR DSE

Wegner [Wegner 1987] describes a design space of object language paradigms with respect to interesting subsets of features such as *objects*, *classes*, and *inheritance*. Following a similar process, we identify interesting subsets of the DSE axioms, relate them to the object-based paradigms given by Wegner, and examine OBS issues in each of the various classifications. This offers a design space for DSE in OBSs based on the axiomatic model and indicates which axioms contribute to the various kinds of expressibility in an OBS.

According to Wegner, *objects* are autonomous entities that consist of a set of "operations" and a "state". Objects respond to operations applied to them and the operation on an object may update the state and return results dependent on the object state. The paradigm of *object-based language* is identified as a language that supports the notion of objects.

A *class* (or *type*) classifies objects according to their common operations. A class acts as a template from which objects can be created. Objects created from the same class have common operations. The paradigm of *class-based language* is identified as an object-based language where every object is supported by a class.

Inheritance serves to categorize classes/types by their shared operations. A class/type may inherit operations of superclasses/supertypes and in turn may have its operations inherited by subclasses/subtypes. Single or multiple inheritance may be supported. The paradigm of *object-oriented language* is identified as a class-based language where a hierarchy of classes/types may be incrementally defined by an inheritance mechanism.

The collection of object-oriented languages are a proper subset of class-based languages which are a proper subset of object-based languages. The three paradigms are summarized in Table III.

Paradigm	Supports
object-based language	objects
class-based language	objects + classes (or types)
object-oriented language	objects + classes (or types) + inheritance

Table III. Summary of object language paradigms.

Object-based languages support the functionality of objects, but object management must be handled outside the language. Class-based languages provide some degree of object management by collecting shared operations in a common place, but do not provide a mechanism to support the management of classes. Object-oriented languages are known as "wide-spectrum languages" because they allow both objects and classes to be managed within the language, thereby providing a uniform mechanism for both high-level design of class hierarchies and low-level implementation of objects.

Following this taxonomy, we relate the inclusion/exclusion of DSE axioms in an OBS to object-based, type-based, or object-oriented paradigms. An OBS that supports the definition of properties for each object on an individual basis fits into the object-based paradigm. Object-based OBSs do not support any of the axioms and do not even support the definitions of $N_e(t)$ and $P_e(t)$ because there is no notion of type. A new definition based on objects instead of types must be introduced for object-based OBS to denote native properties. For example, we could introduce a set, $N(o)$, for each object o that defines the native properties of that object. Each object maintains its own independent set of native properties, which also serves as the complete interface of the object. If different objects need to support common properties, then the properties must be duplicated across the objects. Native properties on a per object basis is the only notion supported by object-based OBSs. Changes to the native properties of objects is the responsibility of the object designer, with the exception that the system could propagate a change to the corresponding object. DSE in object-based OBSs is simple to manage because any schema change to any object o is isolated to the $N(o)$ set of o, meaning no other objects need be considered. As a result, these systems lack in semantic richness because there are no inherent semantic relationships between objects.

An OBS that supports the notion of $N_e(t)$ fits into the type-based paradigm because the common properties of all objects of a type t are recorded by $N_e(t)$. This paradigm identifies the need and reason for introducing $N_e(t)$ into the axiomatic model. None of the axioms are supported for DSE in type-based OBSs because all the axioms rely on inheritance in one form or another. The management of semantics of change is simple in type-based OBSs because any schema change to any type t is isolated to the $N(t)$ set of t, meaning no other types need be considered. However, the result of a schema change may have to be propagated to the objects of the type and, thus, many of the issues concerning change propagation such as "when to propagate the change" (e.g., screening, conversion) need to be addressed in type-based OBSs.

A type-based OBS that supports $P_e(t)$ belongs to the object-oriented paradigm. Since the object-oriented paradigm supports inheritance, the Axiom of Inheritance* (Axiom 9*) is supported by object-oriented OBSs. As a consequence, the Axiom of Interface* (Axiom 7*) must also be supported so that types can form a complete interface based on native and inherited properties. This represents the minimum requirements for an OBS to fit into the object-oriented paradigm.

Table IV illustrates various classifications of OBS paradigms and the features that each supports. Object-oriented OBSs are further classified into more refined subsets that support additional axioms. These subclassifications are discussed below and can be used to compare and differentiate OBSs from one another.

OBSs are typically not concerned with information that is not modeled and stored in the objectbase (i.e., information outside the universe of discourse). As a result, many OBSs support a notion of closure that limits management to the information currently in

Paradigm	Supports
object-based OBS	objects, $N(o)$
type-based OBS	objects + types, $N_e(t)$
object-oriented OBS	objects + types + inheritance, $N_e(t), P_e(t)$
Minimal	Axiom of Inheritance*, Axiom of Interface*
Closed	Minimal + Axiom of Closure
Native	Minimal + Axiom of Nativeness
Lattice	Minimal + Axiom of Supertype Lattice*
Direct supertyped	Lattice + Axiom of Supertypes
Acyclic	Lattice + Axiom of Acyclicity*
Rooted	Lattice + Axiom of Rootedness
Pointed	Lattice + Axiom of Pointedness

Table IV. Summary of OBS design space based on the axiomatic model.

the objectbase. OBSs that support the Axiom of Closure are classified as *closed* systems. Closure is supported by many OBSs and because of its widespread use it could be included as part of the minimal requirements for object-oriented OBSs. However, we leave this as an open design choice and allow the Axiom of Closure to serve as a differentiating factor when comparing systems.

Systems that support *native* properties have a clean separation between properties defined by each type and those inherited by the type. This can be used to improve efficiency in searching for implementations of properties during dynamic dispatching and for reducing the number of implementations that need to be considered when resolving conflicts at the implementation level (i.e., during functional resolution).

The support of a *lattice* of supertypes based on each type is a simple extension to the support of an overall type lattice in object-oriented OBSs. It simply involves maintaining subsets of the type system along with subsets of the partial ordering based on the subtype relationships. Supertype lattice support can be useful for interoperability of systems when portions of the schema need to be integrated with other schemas or exported to participate in a federation of OBSs. It can also serve to help determine the closure of views in OBSs with view management support.

Supertype lattices lead to additional classifications of object-oriented OBSs. Systems that support direct *supertypes* give a type lattice where each type maintains only the minimum number of supertype links. This can simplify dynamic dispatching in some OBSs by reducing the number of supertypes that need to be inspected for inherited properties that inherit implementations from the supertypes as well. For example, GemStone performs Smalltalk-like dynamic dispatching that inspects supertypes of a type when an implementation for a property is not found at that type. Direct supertypes minimize the number of types that need to be inspected. They can also simplify functional resolution because examining only direct supertypes guarantees that the minimal number of types that could contribute to a conflict are being considered. With respect to graphical type browsers, displaying only direct supertype links offers the complete type lattice in the most concise form.

Like closure, an *acyclic* type system is a common feature of OBSs. The obvious advantages lie in the reduced overall complexity of the type system and the polynomial computational complexity of graph based algorithms for searching through the type system.

A *rooted* type system offers a reasonable environment to support uniformity where everything in the system is an object and shares the common interface of the root type. If the root type describes the properties of objects in general, then this description becomes part of the type system and an external notion of object is not required. Since all other types must inherit the properties of the root, they must inherit and support the general notion of objects. If the system itself is defined in terms of types that inherit from the root type, then everything in the system, including the schema, can be modeled as objects.

A *pointed* type system is fairly easy to support and offers some advantages in OBSs. Any type system can be made pointed by *lifting* with a bottom type $\perp$. The bottom type need not be physically stored, but instead can be managed under the semantic notion that $\perp$ is the subtype of all other types and supports the properties of all types. A bottom type can be useful for defining primitive **null** and **error** objects that can be returned as the result of any operation on any object, and subsequently can have any operation applied to them. These objects can be safely used to initialize properties of other objects and to return error conditions on objects as the result of operations. They can subsequently have any property applied to them without encountering the *don't understand message* scenario in Smalltalk or type-checking errors as in C++.

4. EXAMPLE AXIOMATIZATION – DSE 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 2.1. Since the focus of this paper is the effects of changes at the schema level, we limit our discussion to the definition of the semantics of schema changes and exclude change propagation because the latter deals with the effects at the object instance level. See [Peters 1994] for a discussion of change propagation in Tigukat that uses the temporality of the object model [Goralwalla et al. 1995].

4.1 Object Model Overview

The Tigukat object model [Özsu et al. 1995; Peters 1994] 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. Behaviors in the Tigukat model correspond to the generic concept of properties discussed in Section 2.1. The model is *uniform* in that every component of information, including its semantics, is modeled as a *first-class object* with well-defined behavior. This means that in addition to the usual application specific objects (e.g., persons, employees, etc.) the system also manages information about itself (i.e., types, behaviors, classes, etc.) as objects. This results in a self-managed system with reflective capabilities [Peters and Özsu 1993].

The primitive type system of Tigukat is shown in Figure 3. 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 uniformly model the schema, and describe their role in supporting DSE in terms of the axiomatic model. For a complete model definition, including primitive behaviors, see [Peters 1994].

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

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

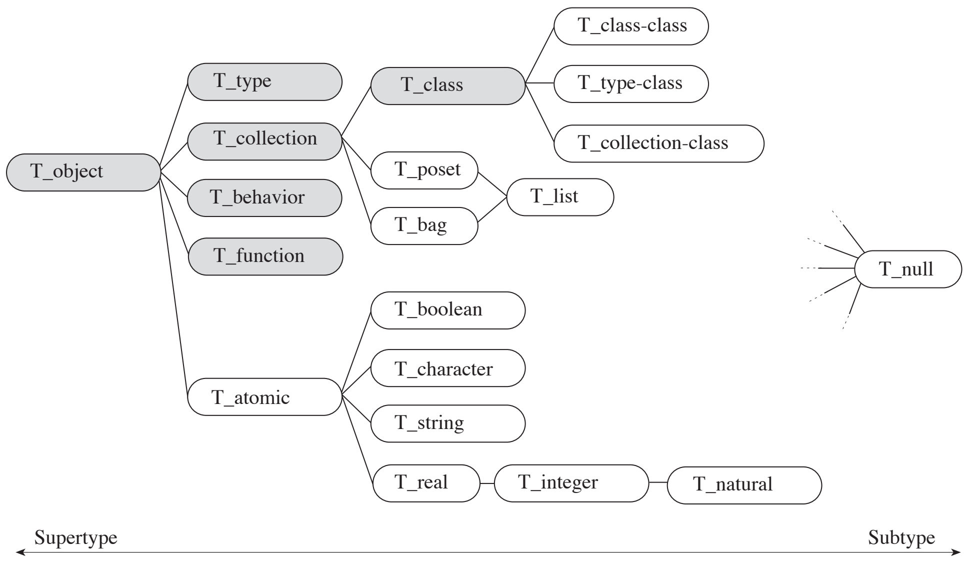

Fig. 3. Primitive type system of the Tigukat object model.

"**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 to denote a partial semantics of behaviors. A complete semantic specification mechanism is currently under development. A signature includes a name that is 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 $o.B\_something$ to denote the application of behavior $B\_something$ to the receiver object o.

A *type* defines behaviors and encapsulates behavior implementations and state representation for objects created using that type as a template. Tigukat supports *multiple inheritance*, so the type structure forms a lattice. The lattice is rooted by the type T_object and *lifted* with the base type T_null to form a *pointed* lattice. 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 denote the *interface* for the objects of that type.

Types are 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 Section 2.1, we define the modeling of types in Tigukat in terms of the axiomatic model.

The uniformity of Tigukat dictates that types are modeled as objects. The primitive type T_type defines the behaviors of type objects. The behaviors of types that are related to schema evolution include *B_supertypes*, *B_super-lattice*, *B_interface*, *B_inherited*, and *B_native*. These correspond exactly to $P(t)$, $PL(t)$, $I(t)$, $H(t)$, and $N(t)$, respectively, in the axiomatic model. The *B_supertypes* behavior returns the immediate supertypes of the receiver type. A type is not an immediate supertype of itself. The *B_super-lattice* behavior returns a partially ordered collection of types representing the supertype lattice, which is pointed at the receiver type and rooted at T_object. In this way, every type is the base type of its supertype lattice. Behaviors *B_interface*, *B_inherited*, and *B_native* represent the collections of behaviors related to the various interface components of types. Furthermore, each type also includes a *B_essSupertypes* and *B_essBehaviors* that maintain the essential supertypes, $P_e(t)$, and essential properties, $N_e(t)$, respectively. This is sufficient to model DSE in Tigukat in terms of the axiomatic model. There are other aspects of Tigukat related to the change propagation aspect of DSE. These are briefly discussed below.

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 their extents. When a schema change occurs to a particular type, the changes must be propagated to the type extent (i.e., to its associated class).

Collections are heterogeneous grouping constructs as opposed to *classes* which are homogeneous up to inclusion polymorphism. Object creation occurs exclusively through classes; thus, classes are extents of types and are managed automatically by the system. Collections on the other hand are managed explicitly by the user. An object cannot exist in Tigukat without its existence in some class. Thus, change propagation on collections is unnecessary because any changes will be performed through the class of the object instead.

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 [Peters and Özsu 1993].

4.2 Definition of Schema

There are various kinds of objects modeled by Tigukat, some of which are classified as *schema objects*. All objects in Tigukat fit into one of the following categories: *type, class, behavior, function, collection* or *other*. These categories are used to distinguish the schema objects from the other objects so that the changes that affect them can be identified as schema evolution operations. A formal definition of schema objects is presented below. This is followed by the definition of what constitutes the "schema" of a Tigukat OBS.

DEFINITION 3. *Schema Objects:* The following general classifications of objects form the *schema objects* of Tigukat:

—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 the interface sets 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 the set of all properties in the axiomatic model, which is represented by $I(\perp)$.

—For all behaviors $b \in BSO$ and for all types $t \in TSO$, the extended union over the implementations of these behaviors (i.e., $\bigcup b.\mathit{B\_implementation}(t)$) 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**. Functions objects denote the implementations of properties in the axiomatic model, which can be represented as attributes, methods, executable code, and so on. The axiomatic model is high-level and does not directly deal 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 impose any restrictions on the management of collections.

—The class **C_class** forms the set of *class schema objects* denoted as CSO. Note that $CSO \subseteq LSO$ because every class is a collection. Classes in Tigukat mirror types in that a class is responsible for managing the entire extent of a type. Thus, the subset inclusion structure of classes is directly related to the subtyping lattice in the axiomatic model. Not every type requires a class, only those that have actual objects in their extent need a class. This allows the formation of *abstract types*, which just define properties of objects and do not allow the creation of objects of that type.

—Any other object is not a schema object and is identified by the *other object* category. □

DEFINITION 4. *Schema:* The *schema* of a Tigukat OBS is the union of all schema object sets:

$$\mathit{schema} \equiv TSO \cup BSO \cup FSO \cup LSO \cup CSO$$

Recall that $CSO \subseteq LSO$, but it is included here for completeness. □

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

	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 V. Classification of schema changes.

4.3 Semantics of Change

In this section the modifications that affect the schema (i.e., the bold entries of Table V) and their axiomatization 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 can be defined in terms of the type-related basic operations.

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

The **MT-DB (Modify Type - Drop Behavior)** operation drops a behavior as an essential component of a type. To drop behavior b from type t, b is removed from $N_e(t)$ and the sets $N(t)$, $H(t)$, $I(t)$ are derived. The resulting sets are reflected in $t.B\_native$, $t.B\_inherited$, and $t.B\_interface$, 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 the sets $P(t)$, $PL(t)$, $H(t)$, $N(t)$, and $I(t)$ are derived. 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 the sets $P(t)$, $PL(t)$, $H(t)$, $N(t)$, and $I(t)$ are derived. 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 because it is always essential.

The **AT (Add Type)** operation creates a new type and integrates it with the existing lattice. Creating a type adds it to TSO ($\mathcal{T}$ in the axiomatic model), which in turn adds it to the schema. 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. 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 sets $P(t)$, $PL(t)$, $H(t)$, $N(t)$, and $I(t)$ are derived. 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 by removing it from TSO, which removes it from the schema. To drop a type t, the type is removed from **C_type** ($\mathcal{T}$ in axiomatic model) and from the P_e of all subtypes of t. The axiomatic model does not specifically define a subtypes property, but this is trivial to define 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 an easy task.

In Tigukat, there is the restriction that the primitive types of the model (i.e., those in the primitive type system of Figure 3) cannot be dropped. When a type is dropped, the type's associated class and all the instances in the shallow extent of the class are dropped as well. With the use of object migration techniques, the instances can be migrated 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 and uniquely associates it with a particular type to manage its extent. Creating a class adds it to CSO, which in turn adds it to the schema. 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, which removes it from CSO and from the schema as well. The instances of a dropped class are also dropped. As mentioned above, object migration techniques can be used to migrate objects to another class before dropping the class in order to preserve the objects.

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 several types and, thus, needs to be dropped from these types.

The **DB (Drop Behavior)** operation drops a given behavior, which could possibly remove it from BSO and, therefore, from the schema as well. 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 (i.e., it is dropped from all N_e containing the behavior).

The **MB-CA (Modify Behavior - Change Association)** operation changes the implementation of a behavior by associating it with a different function. 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 [Peters 1994]. Conflict resolution of high-level behaviors (or properties) 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 must also be a schema change because the function may be associated as the implementation of a behavior in some type.

The **DF (Drop Function)** operation drops a given function, which could possibly remove it from FSO and, therefore, from the schema as well. 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, thereby removing it from LSO and from the schema as well. Unlike classes, dropping a collection does not drop its members. That is, the instances of a dropped collection are not affected.

The **AL (Add Collection)** operation adds a new empty collection to the schema. Collection addition is collection creation as defined by the primitive model. Creating a collection adds it to LSO, which in turn adds it to the schema. The behavior *B_new* defined classes can be applied to **C_collection** to create a new collection object.

The remaining entries in Table V 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 affect 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 to the 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 operations are related to the contents of the collection and, therefore, are not part of the schema evolution problem.

5. DISCUSSION AND COMPARISON

One advantage of the axiomatic model presented in this paper is the precision that it offers in specifying DSE in OBSs. Another important advantage is the possibility of using the axiomatic model in practice to compare and evaluate OBSs based on a common framework that the model provides. In this section, we compare the DSE operations in Tigukat and Orion based on their reductions to the axiomatic model. The end result allows certain judgements to be made about the functionality and expressibility of each system, which serves to illustrate a practical use for the axiomatic model.

Eight fundamental DSE operations are defined in Orion and the authors state that these are inclusive of all "interesting" schema changes. The soundness and completeness of a thorough taxonomy of schema change operations are proven using these eight operations. Soundness in Orion shows that the schema operations generate only valid class lattices, while completeness shows that every legal class lattice is attainable through the schema operations they define. The notions of soundness and completeness used in Orion differ from those used in Section 2.2. Orion deals with its specific identified taxonomy of schema change operations, while Section 2.2 deals with how general schema operations are carried out. We show how the semantics of the eight fundamental operations in Orion can be represented by the axiomatic model. As a result, both notions of soundness and completeness are preserved by the mapping. For the purpose of comparison, in this section, parallels are made between the Orion terminology of *class*, *subclass*, and *superclass* and the Tigukat terminology of *type*, *subtype*, and *supertype*, respectively.

5.1 Axiomatization of Orion

In mapping the Orion class structure to the axiomatic model, P_e represents the superclasses of an Orion class. There is no notion of a set of minimal superclasses, P, in Orion so it can be ignored. There are also no explicit superclass lattices in Orion, but one is implied by the superclass relationships formed among classes.

The superclasses in Orion are ordered to guide conflict resolution and it is trivial to define a total ordering on P_e for this purpose. The ordering is external to the axiomatic model, but this does not adversely affect the axiomatization of Orion because conflict resolution does not alter the interface of a class – only the underlying encapsulated implementations are changed. That is, conflict resolution is at a lower level than the high-level semantics captured by the axioms. The members of P_e are the same in Orion regardless of whether the set is ordered or not. For the sake of completeness, an ordered P_e and conflict resolution are considered in the axiomatization of Orion.

In mapping properties, N_e represents the defined or re-defined properties of an Orion class. There is no notion of the minimal native, N, or inherited, H, properties in Orion. Inherited properties of a class C in Orion can be expressed as $I(C)-N_e(C)$ in the axiomatic model. The interface, I, of a class has the same meaning in Orion and in 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. Note that for comparison purposes the Orion class structure can be *lifted* with a base class $\perp$ = NULL, for example.

The eight fundamental operations of Orion and their semantics can be expressed in terms of the axiomatic model 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:** Remove 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., $P_e(S)$). The operation is rejected if S is the last superclass of C and S is the class OBJECT. The following algorithm illustrates the procedure:

```
if $P_e(C) = \{S\}$ then                        // Is $S$ the last superclass of $C$?
    if $S$ = OBJECT then REJECT operation      // If last superclass is OBJECT, then reject.
    else $P_e(C) = P_e(S)$                      // Else link $C$ to superclasses of $S$.
else remove $S$ from $P_e(C)$                   // $S$ can be safely removed as a superclass.
```

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. In effect, remove S from $P_e(C)$ and then remove S from the lattice.

OP8. **Change the name of a class C:** Change occurrences of C in the P_e of every class to the new name.

Since each of the fundamental operations is specified with an equivalent semantics in the axiomatic model, we conclude that Orion is reducible to the axiomatic model. Furthermore, both our definitions and Orion's definitions of soundness and completeness are preserved. An analogy of this reduction is that one may consider the axioms as a formal assembly language that can be used to precisely specify the semantics of the higher-level schema operations. The reduction of the 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 capabilities of Orion. In Section 5.2, we compare the similarities and differences of schema evolution in Tigukat and Orion in terms of their axiomatizations.

The axiomatization of Orion has two useful results. The first is the unification of DSE on attributes and methods into the single framework of properties, thus simplifying the specification of DSE in Orion and bringing it a step closer to the uniform model. The second is the clarification of DSE semantics in Orion. In particular, the informal definition of OP4 is much better understood in its axiomatic form. Actually, the specification of OP4 as given above is our best interpretation from its informal definition available in the literature. This interpretation may not be entirely correct, which stresses the need for a formal model.

Other systems such as GemStone and Sherpa define DSE policies that follow those of Orion. Like Orion, these definitions are informal and lack a common framework for comparison purposes. Due to similarities with Orion, the axiomatization of these systems are not shown. The axiomatic model serves as a basis for comparing DSE across these systems and other OBSs. In this section, we only compare the axiomatization of DSE in Tigukat and Orion, but this is sufficient to illustrate the salient features of the model.

5.2 Comparison of DSE in Tigukat and Orion

In relation to the design space, Tigukat is an object-oriented OBS that supports all the subcategories for DSE and can benefit from the advantages they provide. Orion is also an object-oriented OBS, but it only supports a proper subset of the subcategories and, therefore, cannot exploit the advantages of the ones missed. Orion is a *closed* system that supports an overall type *lattice* and an implicit notion of a lattice for individual type. The lattice in Orion is *acyclic* and *rooted*, but not *pointed*. Furthermore, Orion does not support *direct supertyped* types nor the management of *native* properties of types.

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 observation 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 is only 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. Due to property inheritance, a user only needs to observe the direct subtype relationships to understand the complete inheritance functionality of a type.

In comparing the schema operations of the two systems, we focus on the operations of Tigukat outlined in Table V and the eight fundamental operations identified in Orion. We first present a side-by-side comparison of the common DSE operations in the two approaches. Second, we discuss the DSE operations that are specific to a particular system. Finally, we offer general comments about the two approaches and express some judgements about their DSE semantics in relation to one another.

The following list associates a DSE operation in Tigukat with its corresponding operation in Orion by separating the two with a slash. The commonality between each Tigukat/Orion operation association is justified and subtle variances are highlighted.

MT-AB/OP1 and MT-DB/OP2. The operations of adding and dropping properties from types (classes) are virtually identical in both systems. The only subtle difference is that Tigukat explicitly maintains the minimal native properties while Orion does not.

MT-ASR/OP3. The operation of adding a subtype relationship between two types (an edge between two classes) is very similar in both systems. The differences are that Orion maintains an ordered supertype list for conflict resolution and Tigukat explicitly maintains a minimal supertype set.

MT-DSR/OP4. 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 regardless of the order in which they are dropped. That is, subtype relationship dropping is order-independent in Tigukat, but not in Orion. Based on the axiomatization of the two systems, we make the judgement that the dropping of subtype relationships in Tigukat is much simpler and is more uniform than Orion.

AT/OP6. 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.

DT/OP7. The operation of dropping a type (class) is different in Tigukat and Orion because both systems base this operation on the operation of dropping subtype relationships (edges). Since the two systems differ in the handling of the latter operation, they differ in the former as well. The differences are equivalent to those defined in the comparison of MT-DSR and OP4 above.

Aside from the common DSE operations above, there are others that vary slightly in the two approaches. The operation of changing the ordering of classes (OP5) is introduced in the axiomatization of Orion to deal with conflict resolution. This is an implementation detail that is abstracted out in the axiomatization of Tigukat because, in general, conflict resolution can differ in various systems.

Similarly, the operation of changing the name of a class (OP8) 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. Objects in Tigukat cannot be renamed with different identities as in Orion because Tigukat objects are created with a unique, *immutable* object identity.

The drop behavior operation (DB) in Table V is defined in terms of MT-DB and so it comparison with Orion in terms of MT-DB/OP2 discussed above. The remaining DSE operations of Tigukat not discussed (namely, AC, DC, MB-CA, DF, AL, DL) all deal with the change propagation issue, which is not addressed in the axiomatic model.

Using the axiomatic model as a basis to represent DSE operation semantics enables comparisons such as the above. Based on this comparison, it is easy to observe the similarities and differences of the two systems, Tigukat and Orion, with respect to schema operations on the type (class) lattice and their properties. Tigukat is a uniform model and this is reflected in its handling of schema changes. The major difference between the two systems is in the handling of dropping subtype relationships (MT-DSR and OP4). In Tigukat, the operation is order-independent, while in Orion it is not. We make the judgement that this difference makes DSE in Tigukat simpler and more uniform. One result of this difference is its ramifications on the consistency of the schema when considering distributed interoperable OBSs. For example, in a distributed system there may be replicated portions of the schema managed by the various semi-autonomous sites. With Orion-like systems the order of the type dropping is important and needs to be carefully scrutinized across the replicated schemas in order to ensure their consistency. On the other hand, for Tigukat-like systems the type dropping order is irrelevant and this may allow for more concurrency among these systems. 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. This is directly related to the dichotomy introduced between the handling of the semantics of change issues *versus* the change propagation issues.

6. RELATED WORK

Various systems have proposed solutions to the problems of *semantics of change* and *change propagation* for DSE in OBSs. 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 access to objects 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 needs to be applied to a particular object. 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 the modification of schema, 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 handles 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 behaviors written according to one version accessing objects of a different version. For example, if a property is dropped from a type, then a filter can be defined on the new version that produces a default value if a method written according to the old type version accesses the dropped property in an object created according to the new version, which does not contain the property. 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.

Orion [Banerjee et al. 1987; Kim and Chou 1988] is the first system to introduce the *invariants* and *rules* approach as a structured way of describing DSE in OBSs. 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. The allowed schema changes in Orion are classified into three categories that affect different components of the schema. The categories describe (1) changes to the contents of a class such as its attributes and methods, (2) changes to the subtyping relationships between classes, and (3) changes to classes as a whole such as adding and dropping entire classes. Orion's taxonomy of changes represents the majority of typical schema modifications allowed in most OBSs. Change propagation in Orion is handled through screening that coerces out-of-date objects to new schema definitions when the objects are accessed. In Section 5, we present a complete axiomatization of the semantics of change for DSE in Orion and compare it to the axiomatization of Tigukat.

Schema evolution in GemStone [Penney and Stein 1987] 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 [Skarra and Zdonik 1986; Skarra and Zdonik 1987] define a framework for versioning types in the Encore object model as a support mechanism for evolving type definitions. Their work is more focused on dealing with change propagation rather than semantics of change. Their schema evolution operations are similar to Orion. A generic type consists of a collection of individual versions of that type. This is known as the *version set* of the type. Every change to a type definition results in the generation of a new version of that type. Since a change to a type can also affect its subtypes because of specialization requirements, new versions of the subtypes may also need to be generated. By default, objects are bound to a specific type version and must be explicitly coerced to a newer version in order to be updated. 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*, that act as filters for managing the semantic differences between versions – such as the missing information problem. This approach is one of the first to address the issue of maintaining behavioral consistency between versions of types.

One result of Skarra and Zdonik's work is a design methodology for defining *handlers*. A *handler* is defined on a type version and specifies an "on condition" that traps read and write access to undefined or invalid properties in that type version. Furthermore, the handler defines an appropriate action to take if such an access occurs. Consider the missing information example above. A handler can be defined on the type version that is the missing property so that it returns a default value, a nil value, or simply generates an error. Using this approach, a handler can be defined for each semantic difference between type versions in order to filter object access and to trap any inconsistent accesses that may occur. This is a filtering approach to change propagation that maintains the semantics of properties between different versions of types. A downside of the approach is that defining handlers on various type versions can become confusing and unmanageable in systems with a large number of types that change often.

Nguyen and Rieu [Nguyen and Rieu 1989] 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 [Skarra and Zdonik 1986] and a complete class resembles a version set.

The properties of relevant classes are characterized automatically by selecting from the powerset of instance variables and constraints defined in a complete class definition. The selection is restricted to only those combinations that are meaningful with respect to certain semantic rules [Nguyen and Rieu 1987]. Objects are instances of exactly one relevant class, which characterizes a partial definition of that object. The purpose of relevant classes is to evaluate the side-effects of propagating schema changes to the instances and to guide this propagation.

Relationships between relevant classes can be characterized as a graph where the nodes are relevant classes and the edges are labeled with schema changes that take one relevant class definition to another. As the schema evolves, relevant classes are used to evaluate the changes and test their semantic consistency. Objects are migrated between relevant classes to effect the changes made to them. This migration is essentially object coercion. The propagation of objects within a set of relevant classes can have a large overhead, but it is argued that relevant classes group objects into smaller sub-classifications so that the number of objects affected by a change within a class is reduced, thereby increasing performance. This approach is valid in systems that consider partial definitions of objects within a class.

In the Farandole 2 model [Andany et al. 1991], a structure called a *context* and the maintenance of versions within contexts are proposed as a basis for schema evolution. A *context* is a partial view of the overall schema that serves a dual purpose: it defines a subset of objects in the database, and a subset of operations that can be performed on these objects. A global database schema can be derived from the set of all contexts. The typical schema changes that follow those of Orion are allowed. A context is represented by a connected graph where the nodes are classes and the edges are attributes denoting relationships between classes. Thus, contexts are similar to entity-relationship diagrams. Schema changes are characterized into graph operations and rules for maintaining graph integrity are defined.

The elements contexts can be shared by other contexts and objects must maintain information about the contexts in which they participate. One must consider the amount of extra space needed to store this information in the objects rather than the types. The focus of the work is on managing changes to schema and no propagation techniques are explicitly stated – although it seems that conversion or screening could be used. There is a brief discussion on how the model improves independence between programs and changing schema, which suggests a filtering approach, but it is unclear how the model achieves this feature. The authors argue that a context provides a smaller group of objects that need to be modified as a result of schema changes, which is intended to improve performance.

Osborn [Osborn 1989] describes an algebra that utilizes inclusion polymorphism to define equivalence of queries on different versions of a schema. The work does not describe how schema changes are propagated to the object instances. Two kinds of schema modifications are considered. The first involves changing simple atomic attributes like strings and integers to more complex aggregates of these simple types (the opposite direction of changing aggregates to simple types is also discussed). Only one level of aggregation is considered. That is, the aggregation of aggregate types is not discussed. The second modification considered is that of specializing aggregate types. The schema is modified by specializing previous types and it is shown how the equivalence of queries are preserved (or not preserved) through polymorphism. The results are interesting, but the full scope of schema evolution is not considered.

In OTGen [Lerner and Habermann 1990], 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. A simple language is provided to describe transformations. Before releasing a transformer, a database administrator can edit the entries in the table to override the default transformations. This provides additional flexibility in defining how changes are filtered.

The semantics of change for DSE in Tigukat is discussed in Section 4. Change propagation is handled by a filtering approach that uses behavior histories [Peters 1994], which are based on the temporal aspects of the object model [Goralwalla et al. 1995]. 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 of behaviors. 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 behavior at a time. This means that some behaviors of the 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.

7. CONCLUSIONS AND FUTURE WORK

Two issues in schema evolution are identified: (i) the *semantics of change* issue describes the possible schema changes and (ii) the *change propagation* issue describes how schema changes affect object instances. In this paper, we address semantics of change by introducing a sound and complete axiomatic model for dynamic schema evolution (DSE) in objectbase systems (OBSs). We develop solutions for change propagation in the Tigukat OBS using the temporal aspects of its object model [Peters 1994; Goralwalla et al. 1995], but a formal treatment of change propagation and its integration with the axiomatic model is part of our future work.

An axiomatization of DSE provides a foundation for describing the schema evolution policies of different systems using a single, underlying framework. The axiomatic model proposed in this paper has the power to provide this basis to OBSs supporting subtyping and property inheritance. Furthermore, the common foundation offered by the model provides a means to better compare the DSE facilities of various systems. A design space for OBSs based on the inclusion/exclusion of axioms is developed and can be used to classify, compare, and differentiate the features of OBSs. To illustrate the power of the axiomatic model, the schema evolution policies of Tigukat and Orion were reduced to the model and compared.

A primary interest is the implementation of DSE in Tigukat based on its reduction to the axiomatic model. A prototype implementation of the core Tigukat object model is complete [Irani 1993] and its extension with efficient algorithms for DSE is currently under development. The completion of this task will provide the necessary empirical evidence of its performance characteristics. Additionally, a formal complexity analysis of our implementation techniques will provide the theoretical basis of performance. From this work we intend to define a taxonomy of performance characteristics of DSE operations in OBSs. Whereas DSE reductions to the axiomatic model provide a basis for theoretical comparison, the performance taxonomy can be used for empirical performance analysis and benchmarking of DSE in existing OBSs.

Another direction being pursued is the extension of the axiomatic model to support schema integration of heterogeneous systems for solving interoperability problems. The formalism of the axiomatic model has the potential to serve as a framework for precisely specifying the semantics of schema integration. We are also investigating the use of a similar approach to define closed views and view management in OBSs.

ACKNOWLEDGMENTS

The authors are grateful to the referees and the Associate Editor for their valuable suggestions that improved the content and presentation of the paper.

REFERENCES

ANDANY, J., LÉONARD, M., AND PALISSER, C. 1991. Management of Schema Evolution in Databases. In *Proc. of the 17th Int'l Conf. on Very Large Databases*, pp. 161–170.

BANERJEE, J., KIM, W., KIM, H.-J., AND KORTH, H. 1987. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pp. 311–322.

GORALWALLA, I., LEONTIEV, Y., ÖZSU, M., AND SZAFRON, D. 1995. A Uniform Behavioral Temporal Object Model. Technical Report TR95-13, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

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

KIM, W. AND CHOU, H.-T. 1988. Versions of Schema for Object-Oriented Databases. In *Proc. of the 14th Int'l Conf. on Very Large Databases*, pp. 148–159.

LERNER, B. AND HABERMANN, A. 1990. Beyond Schema Evolution to Database Reorganization. In *Proc. ECOOP/OOPSLA Conf.*, pp. 67–76.

NGUYEN, G. AND RIEU, D. 1987. Expert Database Support for Consistent Dynamic Objects. In *Proc. of the 13th Int'l Conf. on Very Large Databases*, pp. 493–500.

NGUYEN, G. AND RIEU, D. 1989. Schema Evolution in Object-Oriented Database Systems. *Data & Knowledge Engineering 4*, 43–67.

OSBORN, S. 1989. The Role of Polymorphism in Schema Evolution in an OODB. *IEEE Transactions on Knowledge and Data Engineering 1,* 3 (Sept.), 310–317.

ÖZSU, M., PETERS, R., SZAFRON, D., IRANI, B., LIPKA, A., AND MUÑOZ, A. 1995. TIGUKAT: A Uniform Behavioral Objectbase Management System. *The VLDB Journal 4,* 3 (July), 445–492. Special issue on persistent object systems.

PENNEY, D. AND STEIN, J. 1987. Class Modification in the GemStone Object-Oriented DBMS. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 111–117.

PETERS, R. 1994. TIGUKAT: A Uniform Behavioral Objectbase Management System. Ph. D. thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada. Available as University of Alberta Technical Report TR94-06.

PETERS, R. AND ÖZSU, M. 1993. Reflection in a Uniform Behavioral Object Model. In *Proc. of the 12th Int'l Conf. on Entity–Relationship Approach*, pp. 37–49.

SKARRA, A. AND ZDONIK, S. 1986. The Management of Changing Types in an Object-Oriented Database. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 483–495.

SKARRA, A. AND ZDONIK, S. 1987. Type Evolution in an Object-Oriented Database. In *Research Directions in Object-Oriented Programming*, pp. 393–415. MIT Press.

WEGNER, P. 1987. Dimensions of Object-Based Language Design. In *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 168–182.