Reflection in a Uniform Behavioral Object Model[*]

Randal J. Peters and M. Tamer Özsu

Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

We present the uniform meta-architecture of the TIGUKAT object model and show how it provides *reflection* in object management systems. *Reflection* is the ability for a system to manage information about itself and to access (or reason about) this information through the regular access primitives of the model. The TIGUKAT object model is purely *behavioral* in nature, supports full encapsulation of objects, defines a clear separation between primitive components such as *types, classes, collections, behaviors, functions, etc.*, and incorporates a *uniform* semantics over objects. The architecture of its meta-system is uniformly represented within itself, which gives a clean semantics for reflection.

1 Introduction

To meet data and information management requirements of new complex applications, object management systems (OMSs)[1] are emerging as the most likely candidates. The general acceptance of this new technology depends on the increased functionality it can provide. Obviously, one measure lies in the ability to model complex domains of information. The ability for a model to manage information about itself is seen as a strength since meta-information, like the schema, becomes a first-class component of the objectbase and the access primitives of the model can be used on them uniformly. This uniformity of representation and access is a basis for *reflection*.

Reflection is the ability for a system to manage information about itself and to access (or reason about) this information through the regular "channels" of information retrieval. It is natural for an OMS to manage information about itself since an OMS is a complex application defined by a model.

There are several advantages in providing reflection in an object model. For example, the primitives of the model are used to manage all forms of information (including meta-information) as first-class components (*uniformity of representation*), and information retrieval is uniformly handled by the model's access primitives regardless of the information's type or "status" (*uniformity of access and manipulation*). With these two abilities, a system is capable of *reflection*.

Relational systems provide reflective capabilities by using relations to store information (i.e., schema) about relations. However, the attributes of relations are restricted to the atomic domains of a particular system (i.e., integers, strings, dates, etc.), which limits the semantic richness of the meta-information and makes it awkward to model. With the richer type structures of object models, self management and reflection is more natural.

In a uniform object model, the structures used to manage information about "normal" real-world objects like persons, houses, maps, or complex applications (e.g., a geographic information system) are also used to manage meta-information like types, classes, behaviors, and functions. Furthermore, the access primitives to all these forms of information are uniform, which means there is no distinction, for example, between accessing information about persons and accessing information about types. In this paper, we present a uniform architecture for managing meta-information within an object model and show how the model provides reflective capabilities through uniformity. This work is done within the context of the TIGUKAT[2]

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

[1] We prefer the terms "objectbase" and "object management system" over the more popular terms "object-oriented database" and "object-oriented database management system", since not only data in the traditional sense is managed, but objects in general, which includes things such as code in addition to data.

[2] 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 to Canada with an ancestry originating in the Arctic regions of the country.

Appears in *Proc. of the 12th Int'l Conf. on Entity-Relationship Approach* (ERA'93), pages 37-49, December 1993.

project which has a behavioral object model [10] that supports full encapsulation of objects, defines a clear separation between primitive components such as *types, classes, collections, behaviors, functions, etc.*, and incorporates a *uniform* semantics over objects. All these forms of information are uniformly accessible through the regular access primitives of the model, and in TIGUKAT, this is the application of behaviors to objects. Although our work is within the context of TIGUKAT, the results reported here extend to any system based on a uniform behavioral object model where the application of behaviors to objects is the operational semantics.

The remainder of the paper is organized as follows. In Section 2, we discuss various reflection models that have been proposed in the past and relate our model to these. In Section 3, we give a brief overview of the TIGUKAT object model, which outlines the fundamental features of the model and gives a brief specification of the primitive type lattice. In Section 4, a simplified Geographic Information System (GIS) is presented as a running example to demonstrate the results of this paper. In Section 5, we define the uniform architecture of the TIGUKAT meta-system, and in Section 6, we show how this meta-system supports reflection. A number of example queries are given to illustrate the uniformity over objects, which is our basis for reflection. Concluding remarks and results of the paper are summarized in Section 7.

2 Related Work

In recent years, work on reflection in object-oriented languages (OOLs) has resulted in the identification of two basic models of reflection [2]:

1. The first is called *structural reflection* and was advocated by Cointe [1] in the design of ObjVlisp. The model is based on a uniform instance/class/meta-class architecture where everything is an object, and meta-classes are proper classes in the sense that they can have a number of instances and can be subclassed. The distinction between meta-classes, classes and other instances is a consequence of inheritance and not a type distinction. This is in contrast to Smalltalk-80 [4] where meta-classes are anonymous objects and there is a one-to-one correspondence between a class and its meta-class.

2. The second is called *computational reflection* and was pursued by Paes [7] in the development of 3-KRS. This approach essentially introduces a meta-object for each object to handle both the structural and computational aspects of the object. The work was done within the context of a model that did not support the traditional class/instance structure of object-oriented systems such as Smalltalk, ObjVlisp, and TIGUKAT. Therefore, the structural aspects of objects were represented by the meta-objects as well. In a class/instance model, the structural aspects of an object can be handled by the type (or class) of the object and therefore, meta-objects are only useful for computational aspects in these systems.

   Three models of computational reflection have been identified for object-oriented systems:

   (a) *the meta-class model*, where the meta-object for an object is the class of the object,

   (b) *the specific meta-object model*, where in addition to classes, objects also have specific meta-objects,

   (c) *the meta-communication model*, which is based on the reification of messages sent to objects. That is, messages are objects and can be sent messages to process themselves.

   Some work has been done on adding computational reflection to Smalltalk-80 [3] and work on the ABCL/R2 language [8] is striving towards an efficient implementation of a reflective OOL with concurrency.

Our model supports structural reflection similar to (1) and computational reflection is provided by a meta-class model as in (2a). We did not choose a meta-object model (2b) because of the additional overhead involved. One form of overhead is the introduction of a meta-object for (potentially) every object in the system. Another, more important one in our perspective, is the additional dispatch processing needed for **every** behavior applied to an object. The application of behaviors to objects is the fundamental information access primitive of TIGUKAT. We have gone to great lengths in our implementation to speed up the execution of behavior application [5] and have traded space requirements for execution speed. In a meta-object approach, every behavior application needs to perform an additional check to see if the object has a meta-object and to dispatch the behavior to meta-object if it does. We find this overhead unacceptable because we feel there are only a few occasions where objects need to support the semantics of meta-objects, and the additional cost for each behavior application is too high. Besides, we argue that the semantics of meta-objects can be supported through subtyping and schema evolution, which are features required of an OMS anyway. We expand on this discussion after the introduction of our meta-architecture in Section 5. Another anomaly with the meta-object approach is that some information is at the type level and some information is at the object level. The distribution of some type information on a per object basis has implications for persistent object management (e.g., where to

store the meta-object: with the type, with the object or somewhere else?). Finally, since behaviors are objects in TIGUKAT, some form of the meta-communication model (2c) could be integrated with our system. We are currently investigating the incorporation of these semantics into TIGUKAT.

3 Object Model Overview

The TIGUKAT object model [10] is *behaviorally* defined with a *uniform* object semantics. The model is *behavioral* in the sense that all access and manipulation of objects is based on the application of behaviors to objects. The model is *uniform* in that every component of information, including its semantics, is modeled as a *first-class object* with well-defined behavior.

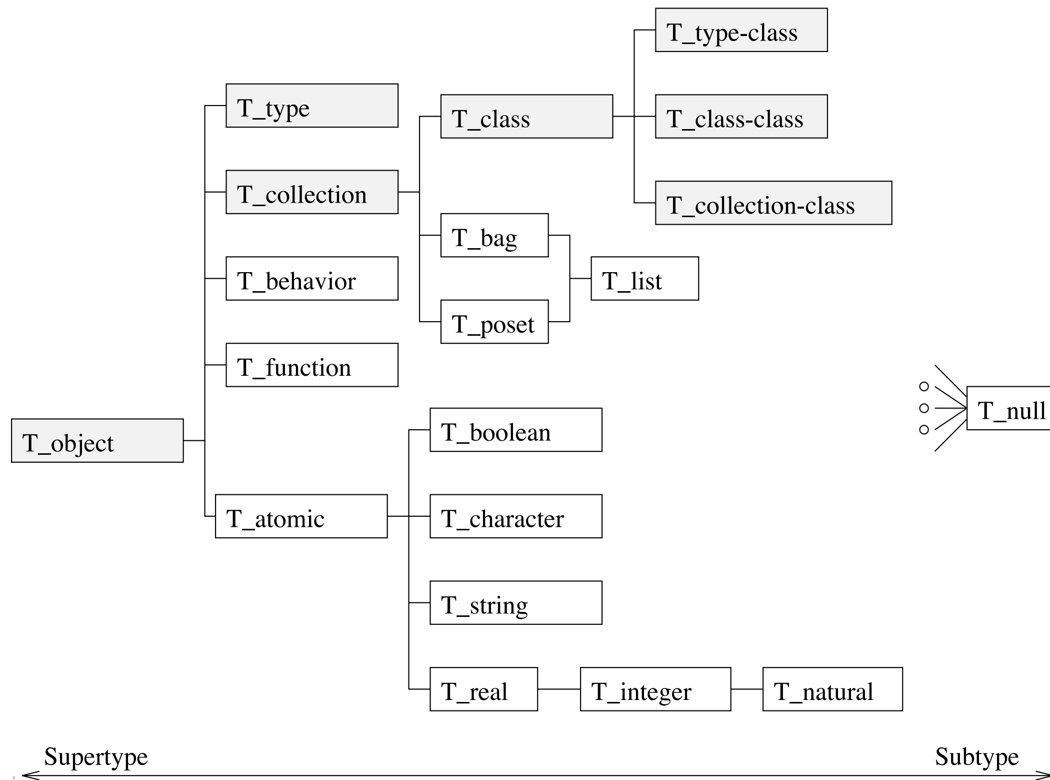

Figure 1: Primitive type system of the TIGUKAT object model.

The primitive type system of the TIGUKAT object model is shown in Figure 1. Each box represents a type. Types define the behaviors applicable to their instances. The type T_object is the root of the type system and T_null is the base. Space limitations do not allow us to give the full behavioral definition of the primitive types. However, the shaded types in the figure and some of their behaviors are elaborated on to show how they support reflection in the model. For a complete definition of the model, including the primitive behaviors, see [10].

The primitive objects of the model include: *atomic entities* (reals, integers, strings, etc.); *types* for defining common features of objects; *behaviors* for specifying the semantics of operations that may be performed on objects; *functions* for specifying implementations of behaviors over types[3]; *classes* for automatic classification of objects based on type[4]; and *collections* for supporting general heterogeneous groupings of objects. In this paper, a reference prefixed by T_ refers to a type, C_ refers to a class, L_ refers to a collection, and B_ refers to a behavior. For example, T_person is a type reference, C_person a class reference, L_seniors a collection reference and B_age a behavior reference. A reference such as David, without a prefix, represents some other application specific reference.

Objects consist of *identity* and *state*. The identity is a unique, immutable identifier assigned by the system and *state* represents the information carried by the object. Thus, the model supports *strong object identity* [6]. This does not preclude

[3] Behaviors and functions form the support mechanism for *overloading* and *late binding* of behaviors.
[4] Types and their extents are separate constructs in TIGUKAT.

application environments such as object programming languages from having many *references* (or *denotations*) to objects, which need not be unique. The *state* of an object *encapsulates* the information carried by that object, meaning it hides the structure and implementation of the object. Conceptually, every object is a *composite* object. By this we mean every object has references (not necessarily implemented as pointers) to other objects.

The access and manipulation of an object's state occurs exclusively through the application of behaviors. We clearly separate the definition of a behavior from its possible implementations (functions/methods). The benefit of this approach is that common behaviors over different types can have a different implementation in each of the types. This is direct support for behavior *overloading* and *late binding* of implementations to behaviors. The meta-system uses this feature to specialize the implementation of behaviors for managing the meta-model.

The model separates the definition of object characteristics (a *type*) from the mechanism for maintaining instances of a particular type (a *class*). A *type* specifies behaviors and encapsulates behavioral implementations and state representation for objects created using that type as a template. The behaviors defined by a type describe the *interface* to the objects of that type. The interface is separated into *inherited* and *native* behaviors, which distinguish the behaviors inherited by the type from those explicitly defined by the type. Types are organized into a lattice-like structure through *subtyping*. TIGUKAT supports *multiple subtyping*, meaning the type structure is a directed acyclic graph (DAG). However, this DAG is rooted by the type T_object and *lifted* with the base type T_null. The type T_null defines objects that can be assigned to behaviors when no other result is known (e.g., null, undefined, etc.).

A *class* ties together the notions of *type* and *object instance*. A *class* is a supplemental, but distinct, construct responsible for managing the instances of a particular type that exist in the objectbase. The entire collection of objects of a particular type is known as the *extent* of the type. This is separated into the notion of *deep extent*, which refers to all objects of a given type, or one of its subtypes, and the notion of *shallow extent*, which refers only to those objects of a given type without considering its subtypes. In general, we use *extent* in place of *deep extent* and explicitly mention *shallow extent* when required.

Objects of a particular type cannot exist without an associated class and every class is uniquely associated with a single type. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes* which imply *types*. Another unique quality of classes is that object creation occurs only through them. Defining object, type and class in this manner introduces a clear separation of these concepts. This separation is important to uniformly define the model within itself and to build a foundation for reflection.

In addition to classes, we define a *collection* as a general grouping construct. A *collection* is similar to a *class* in that it groups objects, but it differs in the following respects. First, object creation cannot occur through a collection; object creation occurs only through classes. Second, an object may exist in any number of collections, but is a member of the shallow extent of only one class. Third, classes are automatically managed by the system based on the subtype lattice whereas the management of collections is *explicit*, meaning the user is responsible for their extents. Finally, the elements of a class are homogeneous up to inclusion polymorphism, while a collection may be heterogeneous in the sense that it may contain objects of types that are not in a subtype relationship with one another.

We define *class* as a subtype of *collection*. This introduces a clean semantics between the two and allows the model to utilize both constructs in an effective manner. For example, we have defined a query model [9] where the targets and results of queries are typed collections of objects. Since classes specialize collections, a class may be used as a target of a query.

The remaining subtypes of T_class, namely, T_class-class, T_type-class and T_collection-class, make up the *meta* type system. Their placement within the type lattice directly supports the uniformity of the model. Section 5 describes the semantics of the behaviors defined on these types and the architecture of the corresponding class and instance structure of the types. This meta-model (within the model) is the foundation of reflective capabilities.

4 Example Objectbase

In this section, we present a geographic information system (GIS) as an example OMS application to demonstrate the reflective capabilities of the model. This example is selected because it is among the application domains that can potentially benefit from the advanced features offered by object-oriented technology.

A type lattice for a simplified GIS is given in Figure 2. The example includes the root types of the various sub-lattices from the primitive type system to illustrate their relative position in an extended application lattice.

The GIS example defines abstract types for representing information on people and their dwellings; these include T_person, T_dwelling and T_house. Geographic types to store information about the locations of dwellings and their surrounding areas are defined; these include T_location, T_zone and its subtypes that categorize the various zones of a geographic area, along with T_map that defines a collection of zones suitable for displaying in a window. Displayable types for presenting information on a graphical device are defined; these include T_display, T_window, and T_map. Finally, T_shape defines the geometric shape of

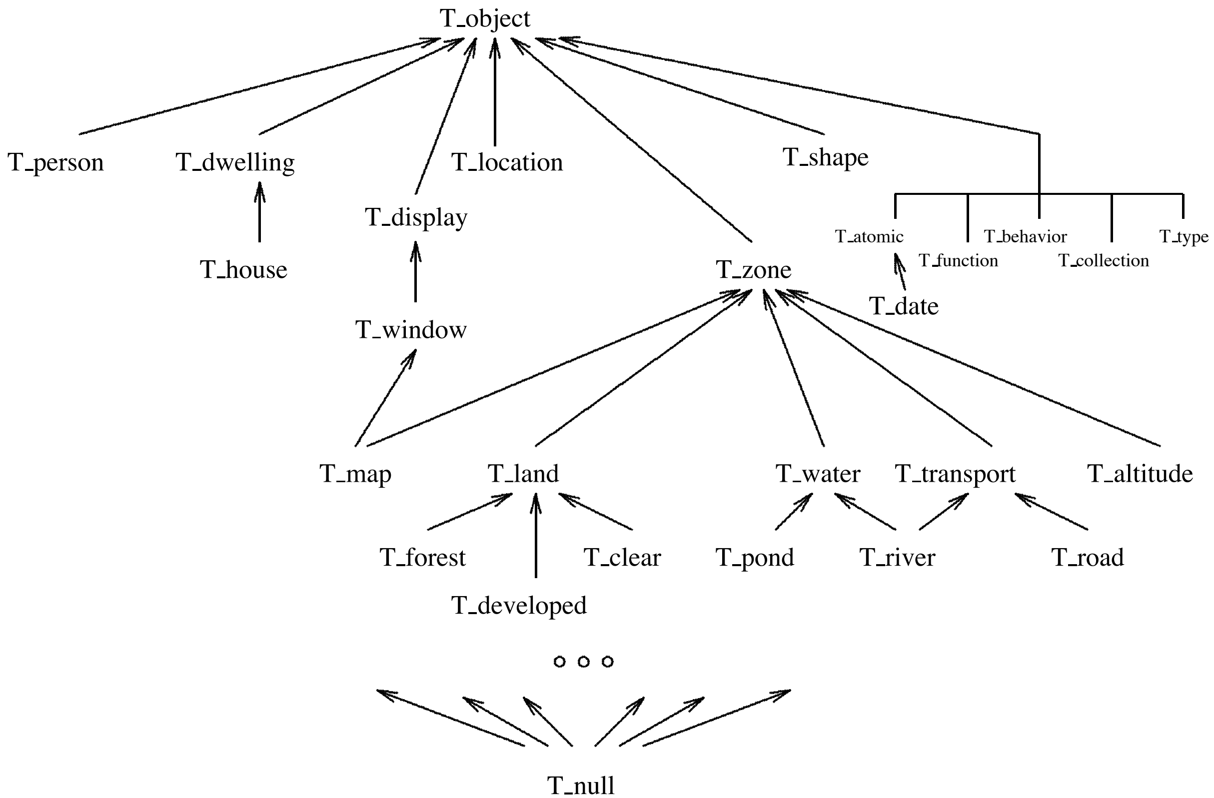

Figure 2: Type lattice for a simple geographic information system.

regions representing the various zones. For our purposes, we only use this general type, but in more practical applications this type would be specialized into subtypes representing polygons, polygons with holes, rectangles, splines, and so on.

Table 1 lists the signatures of the behaviors defined on the GIS specific types. The signatures do not include the type of the receiver object, which is implied from the type that the behavior is defined on. Furthermore, the notation T_collection$\langle T \rangle$ is used to denote a collection type whose members are of type T. By this notation we do not mean parametric collection types.

5 The Uniform Meta-Architecture

Several of the primitives introduced in Section 3 are referred to as *meta-information* because they are objects that provide support for other objects. For example, the type T_type provides support for types by defining the structure and behaviors of type objects, and the class C_class supports classes by managing class objects in the system. In a uniform model, these meta-objects are objects themselves and are managed within the model as first-class objects. The support for this semantics lies in the introduction of higher-level constructs we call *meta-meta-objects* or m^2-objects.

The meta-system of TIGUKAT is a three tiered structure for managing objects. This structure is depicted in Figure 3. Each box in the figure represents a class and the text within the box is the common reference name of that class. The dashed arrows represent shallow extent instance relationships between these objects with the head of the arrow being the instance and the tail being the class that manages the object.

The lowest level of our structure consists of the "normal" objects that depict real world entities such as *integers, persons, maps, etc.*, plus most of the primitive object system is integrated at this level, including types, collections, behaviors and functions. This illustrates the uniformity in TIGUKAT. We denote this level as m^0 and classify its objects as m^0-objects.

The next level defines the class objects that manage the objects in the level below and maintain schema information for these objects. These include C_type, C_collection and all other classes in the system, except for the classes in the level above. This level is denoted as m^1 and its objects as m^1-objects. The reasoning for placing classes at this higher level is that classes maintain objects of the system, every class is associated with a type, and types define the semantics of objects through behaviors. Thus, a class together with its associated type is a form of meta-information.

Type	Signatures	
T_zone	B_title:	T_string
	B_origin:	T_location
	B_region:	T_shape
	B_area:	T_real
	B_proximity:	T_zone → T_real
T_map	B_resolution:	T_real
	B_orientation:	T_real
	B_zones:	T_collection⟨T_zone⟩
T_land	B_value:	T_real
T_water	B_volume:	T_real
T_transport	B_efficiency:	T_real
T_altitude	B_low:	T_integer
	B_high:	T_integer
T_person	B_name:	T_string
	B_birthDate:	T_date
	B_age:	T_natural
	B_residence:	T_dwelling
	B_spouse:	T_person
	B_children:	T_person → T_collection⟨T_person⟩
T_dwelling	B_address:	T_string
	B_inZone:	T_land
	B_age:	T_natural
T_house	B_inZone:	T_developed
	B_mortgage:	T_real

Table 1: Behavior signatures pertaining to example specific types of Figure 2.

The upper-most level consists of the meta-meta-information (labeled m^2), which defines the functionality of the m^1-objects. The structure is closed off at this level because the m^2-object C_class-class is an instance of itself as illustrated by the looped instance edge. The introduction of the m^2-objects adds a level of abstraction to the type lattice and instance structures. The need for this three-tiered structure comes from the fact that every object belongs to a class and every class is associated with a type that defines the semantics of the instance objects in the class. Regular objects (level m^0) belong to some class (level m^1). Since classes are objects, the class objects (level m^1) belong to some class (level m^2). The m^2 class objects belong to the m^2-class C_class-class which closes the lattice. The types associated with these classes are all managed as regular objects at level m^0. The outcome of this approach is that the entire model is consistently and uniformly defined within itself.

The grayed portion of the type lattice in Figure 1 is shown as a companion subclass lattice in Figure 4 where C_x in Figure 4 is the associated class of type T_x in Figure 1. Figure 4 illustrates the subset inclusion and instance structure of some of the m^0, m^1 and m^2-objects in relation to one another. Starting from the left-side of the lattice structure the class C_object is an m^1-object that maintains all the objects in the objectbase (i.e., every object is in the deep extent of class C_object). Two other m^1-objects in the figure are subclasses of C_object, namely, C_type and C_collection. These two classes maintain the instances of types and collections, respectively. Class C_collection is further subclassed by the m^2-object C_class, because every object that is a class is also a collection of objects. For example, the class C_person is an instance of the class C_class, and C_person is a collection of person objects as well. The class C_class manages the instances of all classes in the system like C_object, C_person and so on. Finally, C_class is subclassed by m^2-objects C_type-class, C_class-class and C_collection-class. Intuitively, C_type-class is a class whose instances are classes that manage type objects. Similarly, C_class-class is a class whose instances are classes that manage class objects, and C_collection-class is a class whose instances are classes that manage collection objects.

In understanding the meta-system, it is important to remember that the following general concept holds throughout the model, including the meta-system.

Tenet of Uniformity: *The behaviors defined on a type are applicable to the objects in the class associated with that type.*

For the following discussion, we use the dot notation r.B_something$(a_1, \ldots, a_n)$ to denote the application of behavior B_something to the receiver object r using objects a_1 through a_n as arguments. Furthermore, we use $\{o_1, \ldots, o_n\}$ to denote a collection of objects, and $o \leftarrow B$ to denote the assignment of the result of behavior B to an object o.

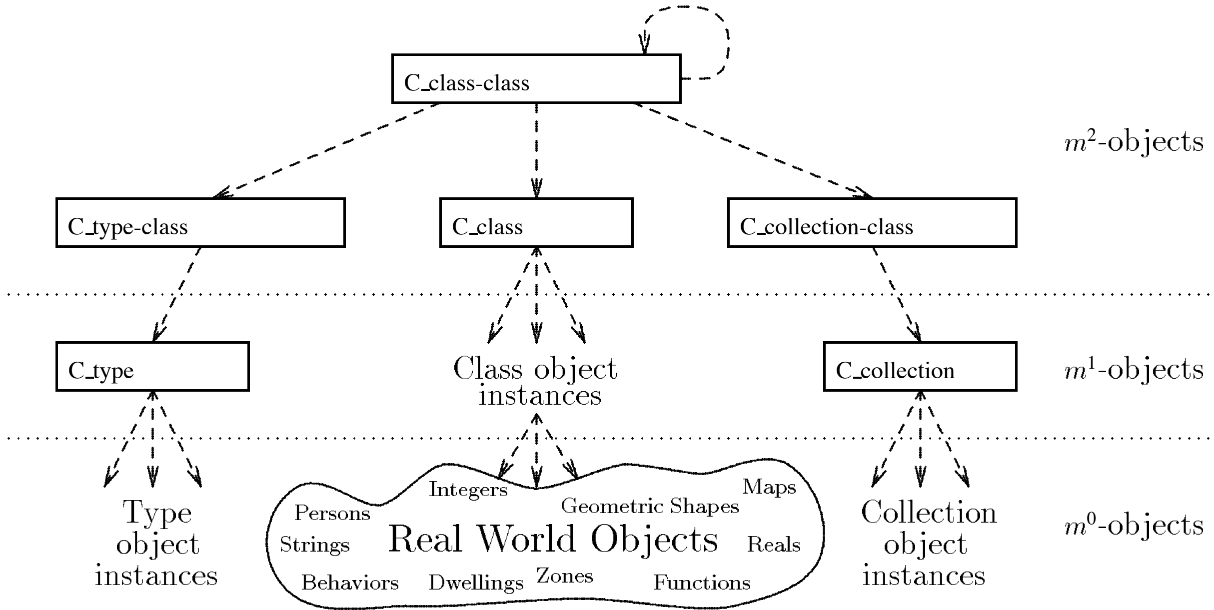

Figure 3: Three tiered instance structure of TIGUKAT object management.

Recall that objects are created through classes. The type T_class defines a behavior B_new for creating objects. Applying B_new to a class object creates an object according to the type specification associated with that class.

The type T_type-class is associated with the class C_type-class, which has the single instance object C_type. Following the tenet of uniformity, the behaviors defined on type T_type-class are applicable to the object C_type. The type T_type-class refines the behavior B_new from T_class for creating new "type" objects. This behavior accepts a collection of types, say $\mathcal{T}$, and a collection of behaviors, say $\mathcal{B}$, as arguments and produces a type as a result. Its semantics is to create a new type as an instance of C_type such that it is a subtype of the types in $\mathcal{T}$ and it defines the behaviors in $\mathcal{B}$ as native behaviors. For example, to create a new type for modeling mobile homes (as a subtype of T_dwelling) that adds a native behavior "B_numberOfMoves:T_natural" (assumed to be defined), we apply B_new to C_type as follows and assign the result to a type reference T_mobileHome:

$$\text{T_mobileHome} \leftarrow \text{C_type.B_new}(\{\text{T_dwelling}\}, \{\text{B_numberOfMoves}\})$$

The type T_collection-class is associated with the class C_collection-class that has the single instance object C_collection. The type T_collection-class refines B_new for creating a new collection object. The behavior accepts a type argument that is used as the membership type of the new collection. Its semantics is to create a new collection object as an instance of C_collection and to set the membership type to the argument. For example, to create a new collection of map objects, we apply B_new to C_collection and assign the result to L_mobileHomeParks as follows:

$$\text{L_mobileHomeParks} \leftarrow \text{C_collection.B_new}(\text{T_map})$$

The previous two examples illustrate how specialization and overriding of implementations (basic modeling concepts) are used to develop the components of the meta-system. B_new has the same semantics of creating a new object as an instance of a particular receiver class, but the implementation of this behavior depends on the type of the receiver class.

The class C_class-class is associated with T_class-class and maintains all the m^2-classes. Its instances include itself, C_type-class, C_collection-class and C_class. The type T_class-class refines B_new for creating a new instance of a class that manages other classes. For the model, this means that we can create additional classes for managing types (additional instances of C_type-class), for managing collections (additional instances of C_collection-type), for managing classes (additional instances of C_class), and for managing classes that manage classes (additional instances of C_class-class).

There are several features that arise from modeling objects in a uniform way, including the ability to perform reflection. We briefly explore these aspects by comparing an m^2 class structure with a "normal" meta-class structure. Following this, we discuss additional uses of the m^2-classes and the uniformity they provide. Section 6 describes how our architecture supports reflection.

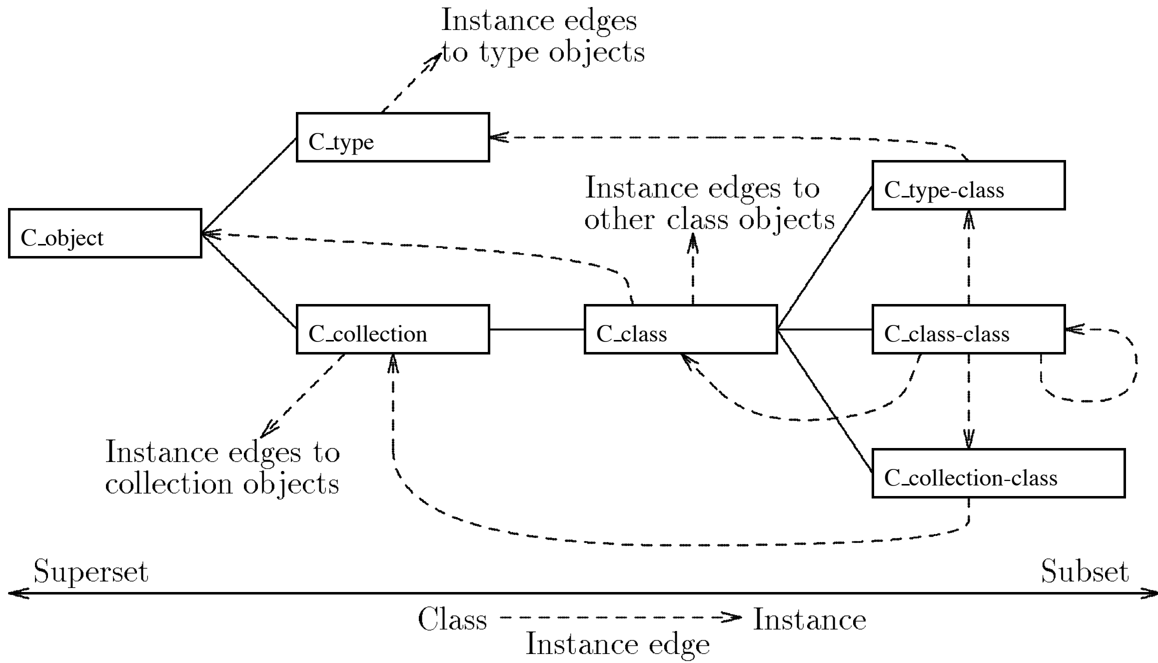

Figure 4: Subclass and instance structure of m^1 and m^2 objects.

One use for this modeling capability is to uniformly define an m^2-class whose associated type includes behaviors for creating default objects of a particular type. For example, consider the GIS objectbase of Section 4 and assume that type T_person and class C_person are defined. The "normal" class and instance structure for this scenario is shown in Figure 5. Instances of T_person are created by applying B_new to class C_person. However, the B_new behavior used in this case is the one defined on T_class, which has a generic implementation of creating a new "empty" object as an instance of the receiver class (i.e., a new "empty" person instance of C_person). Most existing models allow some form of specialized *new* behavior on classes. However, they are usually defined in a roundabout and non-uniform way by stating that a class can have a *new* behavior defined that is applicable to itself (e.g., C++ [11]). This is non-uniform since a class defines some behaviors that are applicable to its instances and some that are applicable to itself. Other models get around this by stating that every class is an instance of itself (e.g., Modular Smalltalk [12]), but in a uniform model this approach raises the question: is the class of persons a person? We want a uniform way of defining a behavior B_new for C_person that creates new objects of type T_person with some default information. It would not make sense to define this behavior on type T_class since then it would be applicable to all classes and we only want it to apply to C_person. The solution lies in the m^2-objects.

We first create a new type called T_person-class as a subtype of T_class which will specialize B_new. The following behavior application performs this task:

$$\text{T_person-class} \leftarrow \text{C_type.B_new}(\{\text{T_class}\},\{\ \})$$

Following this, we redefine the implementation of the inherited behavior B_new so that it creates person objects with some default information (i.e., age set to 0, birthdate set to current date, etc.). In the following discussion we assume this is done. Next, we create and associate an m^2-class with type T_person-class and call it C_person-class.

$$\text{C_person-class} \leftarrow \text{C_class-class.B_new(T_person-class)}$$

Now, it is semantically consistent for the instance C_person-class to have the behavior B_new (the one defined on T_class-class) applied to it. The final step is to create a class, called C_person, as an instance of C_person-class and associate it with the type T_person. This is accomplished by the following behavior application:

$$\text{C_person} \leftarrow \text{C_person-class.B_new(T_person)}$$

This series of behavior applications results in a class and instance structure shown in Figure 6. Now, the class C_person is an instance of C_person-class. Thus, the B_new behavior (the one defined on T_person-class) may be applied to it to create

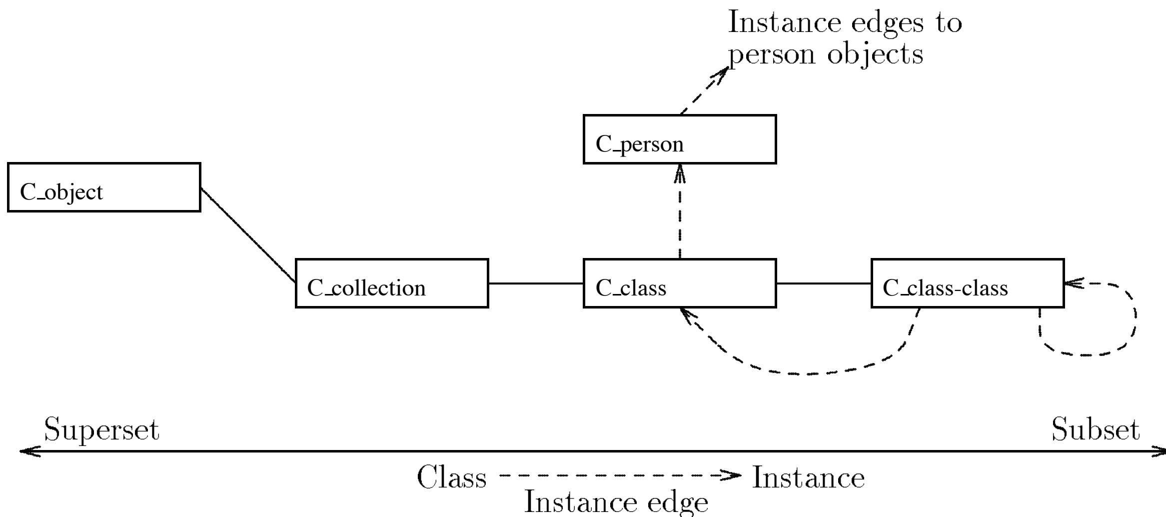

Figure 5: A "normal" class and instance structure for C_person.

a new person with default information (i.e., C_person.B_new() creates a new person with defaults as dictated by the particular implementation). This defines a uniform semantics for the creation and management of objects. Furthermore, the example meta-system for persons was created in a uniform way using the primitives of the TIGUKAT object model.

Different "flavors" of *new* behaviors can be defined for m^2-objects. For example, T_person-class can define *new* behaviors that accept variations of arguments (such as name, age, address, etc.) and create a new person with the given arguments. Furthermore, we can define a variety of default *new* behaviors that create person objects with various defaults (e.g., B_newBorn, B_newYouth, B_newSenior, etc.).

Another feature of the meta-system is that the m^2-classes support a uniform definition of *class behaviors* (i.e., behaviors that are applicable to classes). For example, a behavior B_averageAge can be defined on T_person-class that computes the average age of persons in a class. Now, this behavior is applicable to the class C_person and applying it as C_person.B_averageAge() yields the average age of the persons in the objectbase. If we subtype T_person with T_student and want the same semantics with the class C_student, then we can create C_student as an instance of C_person-class. Then B_averageAge is applicable to C_student and computes the average age of the students in the objectbase. Any number of "person-like" classes (employee, teaching assistant, etc.) can be created in this way and have these semantics attached to them. A similar approach can be used to generalize the concept to collections. That is, define *collection behaviors*, such as B_averageAge, that are applicable to collections and compute various results from the members of collections.

Our meta-system architecture is similar to the meta-class structure in ObjVlisp [1] and is a generalization of the Smalltalk-80 [4] parallel one-to-one class/meta-class lattice in the sense that our approach is entirely uniform. Every class, including the m^2 classes are proper class that, in general, have multiple instances and can be subclassed (i.e., their associated types can be subtyped). One advantage is that there is less overhead for classes that do not need additional class behaviors or do not need to specialize class behaviors. For example, both C_person and C_student can be defined as instances of C_person-class if C_student does not require additional class behaviors or specialization of existing ones. Furthermore, classes that do not require any class behaviors can be instances of the general C_class. This illustrates that m^2 classes are classes whose instances are class objects. A potential problem is that the schema needs to be reorganized if at a later time it is decided that additional class behaviors are required for certain classes that were grouped as instances of one meta-class (e.g., if we decide that additional behaviors applicable to C_student, but not to C_person are needed). This kind of "evolution" can be viewed as correcting design problems of an application (i.e., it was a design mistake to create C_student as an instance of C_person-class). The problem is corrected by subtyping T_person-class with T_student-class, defining the new behaviors and specializations on this type, creating an associated class C_student-class, and migrating C_student to be an instance of C_student-class. The reason for this reorganization is because both structural and computational reflection are handled by the type. We feel the need for this kind of schema reorganization will be minimal. Nonetheless, with the development of our schema evolution and object migration policies, these kinds of changes will follow naturally since some form of them must be supported in a full-featured OMS.

Another solution is to introduce *meta-objects* to handle the computational aspects of objects [7, 2]. This avoids schema reorganization by allowing behaviors to be redefined in the meta-objects instead of the type. However, it requires some

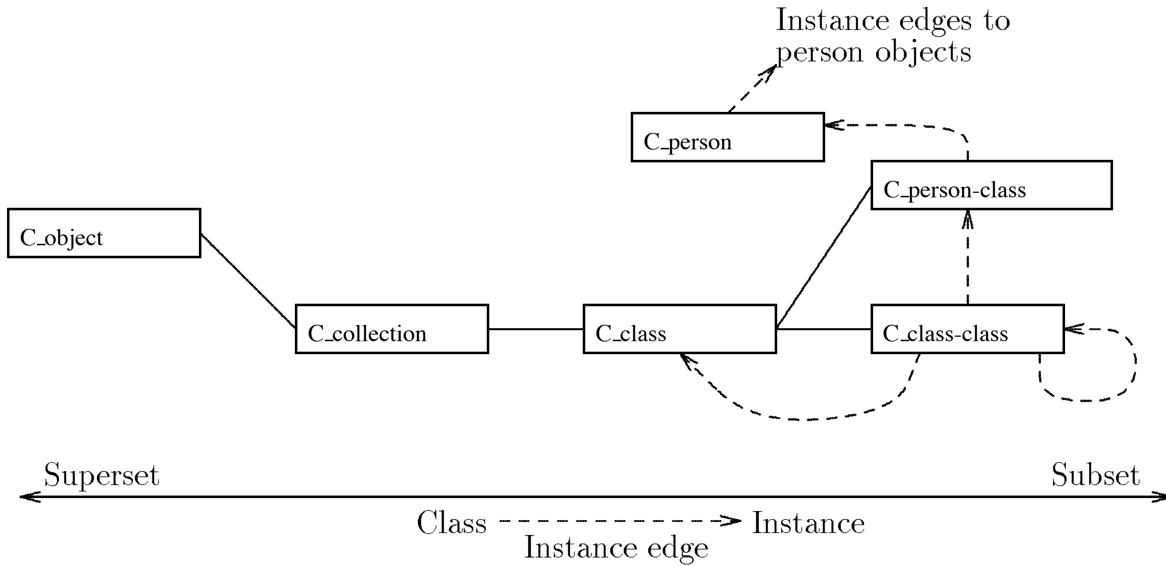

Figure 6: An m^2 class and instance structure for C_person.

additional dispatch processing to determine if an object has a meta-object and if it does, to tell the meta-object to handle the behavior. If the object does not have a meta-object, then the regular type dispatch should occur. Furthermore, there are additional space requirements since every object can potentially have a meta-object. The drawback of this approach in an objectbase environment is that efficient query processing is a necessity and the overhead of the additional dispatch processing for **every** behavior application can be quite significant in queries where many behaviors are being applied. Thus, we have chosen to trade-off the flexibility of meta-objects (that we can get through subtyping instead) for speed.

It is easy to see how the tenet of uniformity carries through for all objects and forms a basis for reflection. For example, the object David is a person, David is in the extent of class C_person, the associated type of C_person is T_person, the behaviors defined by T_person are applicable to David. The object C_person is a class, C_person is in the extent of class C_class (or C_person-class in the m^2 example of Figure 6), the associated type of C_class is T_class (or T_person-class), the behaviors defined by T_class (or T_person-class) are applicable to C_person. The same line of reasoning can be applied to T_person, T_person-class, C_class, T_type and uniformly to all objects in TIGUKAT. The base (fixpoint) of the type chain is T_type and the base of the class chain is C_class-class. This defines the closure of the lattice and instance structure.

6 Reflective Capabilities

Reflection is the ability for a system or model to manage information about itself and to access this information using the regular "channels" of information retrieval. The uniform meta-architecture described in Section 5 is consistent with the modeling capabilities of the TIGUKAT object model and the model uniformly manages information about itself. The access primitives of the model (i.e., the application of behaviors to objects) can be uniformly applied to all objects, including meta-information. Thus, uniformity in TIGUKAT supports reflection.

We have developed a query model with an SQL-like language called TQL (TIGUKAT Query Language) [9]. The *select-from-where* clause of the language is an object-oriented extension of SQL. We use the basic structure of this clause to present some queries that illustrate the reflective capabilities of TIGUKAT. First, to get a feel for the syntax, we give some example queries on "normal" real-world objects. These examples also serve to illustrate that the method of querying real-world objects is uniform with querying meta-information like schema.

Example 6.1 Return land zones valued over $100,000 or that cover an area over 1000 units.
select o
from o **in** C_land
where (o.B_value() $>$ 100000) **or** (o.B_area() $>$ 1000)

Example 6.2 Return all zones that have people living in them (the zones are generated from person objects).

```
select  o
from    p in C_person
where   o = p.B_residence().B_inZone()
```

Example 6.3 Return all maps that describe areas strictly above 5000 feet.

```
select  o
from    o in C_map
where   forAll p in (select q from q in C_altitude, q in o.B_zones()) p.B_low() > 5000
```

Example 6.4 Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map.

```
select  p, q.B_title()
from    p in C_person, q in C_map
where   p.B_residence().B_inZone() in q.B_zones()
```

The above queries introduce variables (i.e., o,p,q) that range over classes and collections. The queries apply behaviors to the variables and other object references to extract information about the objects and return the information (in the form of objects) as part of the query result.

By using the behavioral application paradigm on meta-objects, we can uniformly access information about the meta-system of the model. For example, we can retrieve information about types by querying the class C_type. Some example *reflective* queries are given below. The examples reference several behaviors defined on types, behaviors, collections, and objects in general. The signatures for these behaviors, along with a brief explanation, are given in Table 2 (signatures do not include the type of the receiver as in Section 4). There are many other behaviors defined on these types, but space limitations do not allow us to specify them.

Type	Signature		Explanation
T_object	B_mapsto:	T_type	*Return the type of the receiver object.*
T_type	B_interface:	T_collection⟨T_behavior⟩	*Return the full interface of the type.*
	B_native:	T_collection⟨T_behavior⟩	*Return only the native behaviors of the type.*
	B_inherited:	T_collection⟨T_behavior⟩	*Return only the inherited behaviors of the type.*
	B_supertypes:	T_collection⟨T_type⟩	*Return the immediate supertypes of the type.*
	B_super-lattice:	T_poset⟨T_type⟩	*Return all supertypes of the type, partially ordered by subtyping.*
	B_sub-lattice:	T_poset⟨T_type⟩	*Return all subtypes of the type, partially ordered by subtyping.*
T_behavior	B_impl:	T_type → T_function	*Return the implementation (i.e., function) of the behavior in the given type.*
T_collection	B_memberType:	T_type	*Return the membership type of the collection.*
	B_cardinality:	T_natural	*Return the cardinality of the collection.*

Table 2: Some behavior signatures for certain primitive types.

Example 6.5 Return the types that have behaviors B_name and B_age defined as part of their interface.

```
select  t
from    t in C_type
where   B_name in t.B_interface() and B_age in t.B_interface()
```

Example 6.6 Return the types that define behavior B_age with the same implementation as one of the supertypes.

```
select  t
from    t in C_type, r in t.B_supertypes()
where   B_age in t.B_interface() and B_age in r.B_interface() and B_age.B_impl(t) = B_age.B_impl(r)
```

Example 6.7 Return all types that inherit behavior B_age, but define a different implementation from all types in the super-lattice that define behavior B_age.

```
select  t
from    t in C_type
where   B_age in t.B_inherited()
        and forall r in t.B_super-lattice()
            ((not r = t) or (not B_age in r.B_interface()) or (not B_age.B_impl(t) = B_age.B_impl(r)))
```

Example 6.8 Return all subtypes of T_zone.

```
select  r
from    r in T_zone.B_sub-lattice()
```

Example 6.9 Return pairs consisting of a subtype of T_zone and the native behaviors that the subtype defines.

```
select  r, r.B_native()
from    r in T_zone.B_sub-lattice()
```

Example 6.10 Return pairs consisting of an object in collection L_stuff together with the type of the object, but only if it is a subtype of T_water.

```
select  o, o.B_mapsto()
from    o in L_stuff
where   o.B_mapsto() ∈ T_water.B_sub-lattice()
```

The following *reflective* queries are defined on classes and collections. They illustrate how uniformity carries through to these kinds of objects.

Example 6.11 Return all the classes in the objectbase.

```
select  o
from    o in C_class
```

Example 6.12 Return only those classes that make up the meta-meta-system.

```
select  o
from    o in C_class-class
```

Example 6.13 Return all collections that contain the object Dallas, but only if the membership type of the collection is T_land or one of its subtypes.

```
select  o
from    o in C_collection, p in o
where   o.B_memberType() in T_land.B_sub-lattice() and p = Dallas
```

Example 6.14 Return the classes that have a greater cardinality than any collection in the system without considering the cardinality of other classes.

```
select  o
from    o in C_class
where   forall p in C_collection
            ((not p in C_class) or o.B_cardinality() > p.B_cardinality())
```

Example 6.15 Return pairs consisting of an m^2-class and the collection of native class behaviors defined by that m^2-class.

```
select  c, c.B_memberType().B_native()
from    c in C_class-class
```

Example 6.16 Return objects from L_things that exist in at least one other collection without considering their existence in a class.

```
select  o
from    o in L_things, p in C_collection
where   (not p = L_things) and (not p in C_class) and (o in p)
```

The paradigm of behavioral application can be applied uniformly to all objects in TIGUKAT since every object belongs to the extent of some class, every class is associated with a type, and every type defines behaviors that are applicable to the objects in the extent of the associated class. Note that some examples (6.13) intermix access to "normal" real-world objects with access to types and collections. This is a consequence of uniformity.

The object model approach differs from relational systems, which use relations to store information about schema, in that the attributes of relations are limited to the atomic domains of a particular system (i.e., integers, strings, dates, etc.) while an object model has a richer type system for representing complex objects and a more sophisticated execution model that allows the application of general behaviors to objects. Thus, representing schema information in a uniform object model is more natural and easier to manage. As a consequence, the access primitives apply naturally to all forms of information.

7 Conclusions

In this paper, we describe the uniform meta-architecture of the TIGUKAT object model, how it manages information about itself, and the uniform access primitive of applying behaviors to objects. We show how the model uniformity forms a basis for reflective capabilities. Types in TIGUKAT support both structural and computational reflection of objects through the definition and specialization of behaviors, and through subtyping.

The tenet of uniformity is defined to describe the basic property that applies to all objects in a uniform model: *behaviors defined on a type are applicable to objects in the class associated with that type*. All objects in TIGUKAT exist in some class, and every class is associated with a type, and every type defines behaviors applicable to objects in its associated class. Thus, the paradigm of applying behaviors to objects carries uniformly to all objects, including types, classes, behaviors, and so on.

Using an SQL-like query language developed for the model [9], we compare several queries on real-world objects with queries on meta-information and show that in a uniform model, there is no distinction between "normal" objects and meta-objects because everything is a first-class object with well-defined behavior. These queries retrieve information about types, classes and collections (parts of the schema) by applying behaviors to objects in a uniform way. Some queries even mix regular and meta-objects in a single query to further illustrate the uniformity. The information retrieved by these queries is information about the system (i.e., schema information), which is what reflection is all about.

Our meta-system design has similarities to ObjVlisp [1] and is a uniform extension to the Smalltalk-80 [4] meta-class architecture. It is more general in the sense that it can mimic the parallel meta-class structure of Smalltalk-80, but does not force this semantics. Other differences are that any class in TIGUKAT can have many instances and any type can be subtyped. Thus, the *metaness* of an object is a consequence of inheritance and gives rise to a uniform model. One advantage is reduced overhead since not all classes require a meta-class. However, some subtype reorganization is required if at a later time a particular class needs to specialize a meta-class. These changes can be viewed as application design corrections and our schema evolution policies make these changes natural.

Since behaviors are objects in TIGUKAT, we feel that some form of the meta-communication model (model (2c) in Section 2) could be integrated with our system. We are currently investigating the incorporation of these semantics into TIGUKAT.

References

[1] P. Cointe. Metaclasses are First Class: the ObjVlisp Model. In *Proc. OOPSLA Conf.*, pages 156–167, October 1987.

[2] J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. In *Proc. OOPSLA Conf.*, pages 317–326, October 1989.

[3] B. Foote and R.E. Johnson. Reflective Facilities in Smalltalk-80. In *Proc. OOPSLA Conf.*, pages 327–335, October 1989.

[4] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[5] B. Irani. Implementation Design and Development of the TIGUKAT Object Model. Master's thesis, Dept. of Comp. Sci., University of Alberta, Edmonton, AB, Canada, 1993. Available as University of Alberta Technical Report TR93-10.

[6] S.N. Khoshafian and G.P. Copeland. Object Identity. In *Proc. OOPSLA Conf.*, pages 406–416, September 1986.

[7] P. Maes. Concepts and Experiments in Computational Reflection. In *Proc. OOPSLA Conf.*, pages 147–155, October 1987.

[8] H. Masuhara, S. Matsuoka, T. Wantanabe, and A. Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In *Proc. OOPSLA Conf.*, pages 127–144, October 1992.

[9] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. An Extensible Query Model and Its Languages for a Uniform Behavioral Object Management System. In *Proc. of the Second Int'l. Conf. on Information and Knowledge Management*, pages 403–412, November 1993. A full version of this paper is available as University of Alberta Technical Report TR93-01.

[10] R.J. Peters, M.T. Özsu, and D. Szafron. TIGUKAT: An Object Model for Query and View Support in Object Database Systems. Technical Report TR92-14, Dept. of Comp. Sci., University of Alberta, Edmonton, AB, Canada, October 1992.

[11] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1991. Second edition.

[12] A. Wirfs-Brock and B. Wilkerson. An Overview of Modular Smalltalk. In *Proc. OOPSLA Conf.*, pages 123–134, October 1988.