

TIGUKAT: A Uniform Behavioral Objectbase Management System

M. Tamer Özsu, Randal Peters, Duane Szafron,
Boman Irani, Anna Lipka, Adriana Muñoz

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

Abstract

We describe the TIGUKAT objectbase management system that is under development at the Laboratory for Database Systems Research at the University of Alberta. TIGUKAT has a novel object model whose identifying characteristics include a purely behavioral semantics and a uniform approach to objects. Everything in the system, including types, classes, collections, behaviors, functions as well as meta-information, is a first-class object with well-defined behavior. In this way, the model abstracts everything, including traditional structural notions such as instance variables, method implementation and schema definition, into a uniform semantics of behaviors on objects. Our emphasis in this paper is on the object model, its implementation, the persistence model and the query language. We also (briefly) present other database management functions that are under development such as the query optimizer, the version control system and transaction manager.

1 Introduction

The penetration of data management technology into new application areas with more demanding requirements than business data processing has generated a search for appropriate data models and system architectures to support these requirements. Some examples of these application areas are engineering design systems, knowledge base system applications, office information systems, and multimedia systems. It is now commonly accepted that relational database management systems (DBMSs), with their flat representation of data, do not have sufficient power to fulfill these requirements. The fundamental difficulty relates to the recognized semantic mismatch between the entities that are commonly encountered in these application domains and the representation provided by the underlying DBMS.

Object-oriented technology is the topic of intense study as the major candidate to successfully meet the requirements of advanced applications that use data management services. At the Laboratory for Database Systems Research at the University of Alberta, we are engaged in the design and development of an objectbase management system (OBMS)¹, called TIGUKAT² which follows the object-oriented methodology in its own design. Consequently, all database functionality is incorporated within an extensible object model. In this paper, we provide a general overview of TIGUKAT with special emphasis on its object model, its implementation and the persistence model. Some of the novel features of TIGUKAT are the following:

¹ We prefer to use the terms “objectbase” and “objectbase management system” rather than the more popular “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 like 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.

1. It has a purely behavioral object model where the user (a person or an application program) interacts with the system only by applying behaviors to objects. In this way, full abstraction of modeled entities is accomplished since the users do not have to differentiate between attributes and methods.
2. Its object model is uniform. Everything in the system, including types, classes, collections, behaviors, functions and meta-information, is a first-class object with well-defined behavior. Thus, there is no separation between objects and values so the schema information is a natural part of the database that can be queried just like other objects.
3. This uniformity extends to other system entities (e.g., queries, transactions, views) which are treated as objects that can be created, stored, manipulated and queried like any other object.

Two different approaches have been followed in the development of OBMSs. The first approach is to adopt the type system of an object-oriented programming language as the object model of the OBMS. For example, ObjectStore [LLOW91] adopts the type system of C++ [Str86] while Gemstone [BOS91] follows the type system of Smalltalk [GR83]. The second alternative is what is known as *language-independent* or *generic* object models where the OBMS defines its own object model and appropriate mappings are provided from languages to this object model. TIGUKAT follows the second approach as does, for example, O₂ [Deu91]. A database programming language is being designed that is tightly integrated with the TIGUKAT object model. In addition, mappings will be provided from other programming languages.

TIGUKAT is an experimental system that is under constant development and revision. We have, therefore, chosen to follow an extensible system design approach. The uniformity of the model, which treats all system entities as objects, is the basis of TIGUKAT's extensibility. The general architecture of the system is depicted in Figure 1. To date, most of the development and implementation work has concentrated on the object model, the query model, and the implementation of query languages. The architectural framework of the query optimizer has also been developed [Mun93], however the details of the optimizer (e.g., the full set of transformation rules, the detailed cost functions) have yet to be implemented.

The organization of the paper is as follows. We start, in Section 2, with an overview of the TIGUKAT object model, presenting the primitive type system. We include an example database application design to demonstrate the features of TIGUKAT. Section 3 describes some of the more important implementation design decisions and the approach we have taken. This is followed, in Section 4, with a description of the persistence model of TIGUKAT. Section 5 presents the query model with emphasis on the user-level languages. A more detailed description of the object and query models are given in [PLÖS93a, Pet94]. In Section 6, we provide a brief overview of our approach to providing the common database management functions such as query optimization, version management and transaction management. Finally, in Section 7 we end with a discussion of our future research directions.

2 Object Model

The TIGUKAT object model is defined *behaviorally* with a *uniform* object semantics. The model is *behavioral* in the sense that all access and manipulation of objects occurs through the application of behaviors (operations) on objects. The model is *uniform* in that every concept within the model has the status of a *first-class object*.

Uniformity in TIGUKAT is similar to the approaches of DAPLEX [Shi81] and its object-oriented counterpart OODAPLEX [Day89]. However, our definition of uniformity is complete in that it unconditionally extends over all forms of information, including the system components such as the schema, meta-information, query model, query optimizer, view manager, transaction manager, and so on. We adopt another significant aspect of these models: their functional approach to defining behaviors. TIGUKAT enhances this approach by providing a separation of behavior, which is a semantic notion, from function, which is a means of implementing behavioral semantics.

The TIGUKAT model defines a number of primitive objects that include: *atomic entities* (such as *reals*, *integers*, *strings*, etc.); *types* for defining common features of objects; *behaviors* for specifying the semantics of

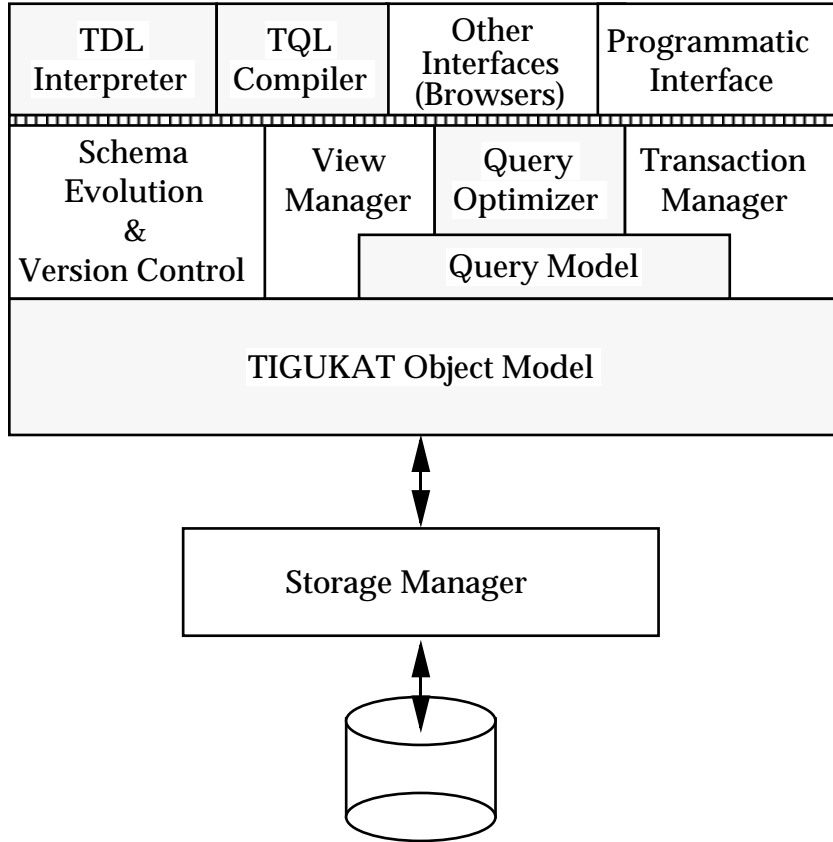


Figure 1: The TIGUKAT System Architecture

the operations that may be performed on objects; *functions* for specifying the implementations of behaviors over various types³; *classes* for the automatic classification of objects based on their type⁴; *collections* for supporting general, heterogeneous, user-definable groupings of objects; and higher-level constructs to uniformly represent meta-information (i.e., schema) as objects with well-defined behavior. This last feature gives the system *reflective capabilities* [PÖ93].

The primitive type system of TIGUKAT is shown in Figure 2 with the type **T_object** as the root of the lattice and type **T_null** as the base. The type **T_null** defines objects that can be returned by behaviors when no other result is known (e.g., null, undefined, etc.). These are necessary because the result of every behavior application in TIGUKAT must be a reference to an object. There are no dangling references in TIGUKAT.

As a notational convenience, the prefix **T_** refers to a type, **C_** refers to a class, **L_** refers to a collection, **B_** refers to a behavior and **F_** refers to a function. Each prefix also has its own font variation for the string following it. For example, **T_city** is a type reference, **C_city** is a class reference, **L_historicSites** is a collection reference, **B_population** is a behavior reference, **F_calcPopulation** is a function reference and a reference such as **Edmonton** without any prefix represents some other application specific object reference.

³Associations between behaviors and functions form the support mechanism for *overloading* and *late binding* of behaviors.

⁴Types and classes are separate constructs in TIGUKAT.

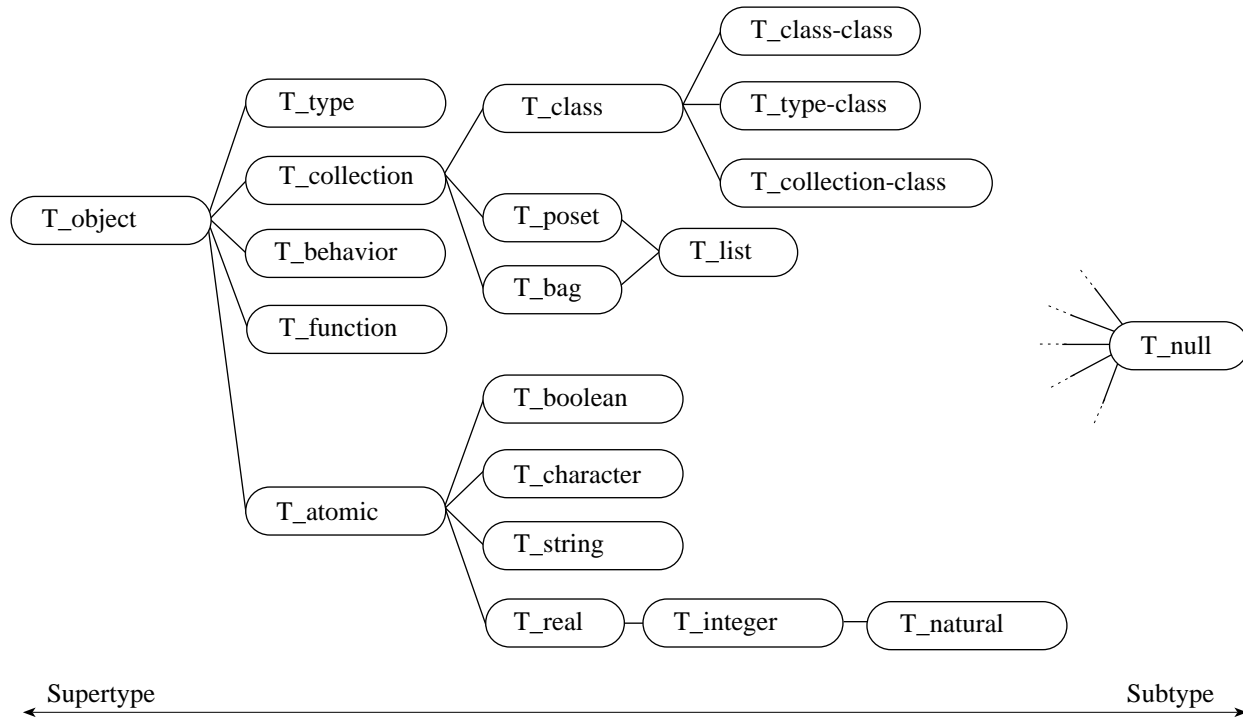


Figure 2: Primitive type system of TIGUKAT.

2.1 Behaviors and Functions

The access and manipulation of objects occurs exclusively through the application of behaviors. This is similar to the message-based approach of Smalltalk [GR83] and OODAPLEX [Day89]. Appendix A lists the signatures for the native behaviors defined by the primitive types of Figure 2.

We separate the definition of a behavior from its possible implementations, which are represented by TIGUKAT functions (corresponding to *methods* in other models). The benefit of this approach is that common behaviors over different types can have a different implementation in each of the types (known as *overloading* the behavior). This gives the model the ability to *dynamically bind* behaviors to implementations at run time (known as *late-binding*).

There are two kinds of implementations for behaviors. One is a *computed function*, which consists of runtime calls to executable code and the other is a *stored function*, which is a reference to an existing object in the objectbase. Stored functions eliminate the need for instance variables, which limit reuse [WBW89b]. The uniformity of TIGUKAT conceptually transforms each behavioral application into the invocation of a function, regardless of whether the function is stored or computed. This allows designers to concentrate on semantic responsibilities rather than on data attributes [WBW89a]. For example, the type designer is free to develop a purely behavioral specification of a type while the type implementor decides whether the behaviors are implemented by stored or computed functions.

The semantic definition of a behavior has many forms. A simple approach, common in other models, is a *signature* expression consisting of a behavior name, parameter types and a return type. Signatures are useful and necessary for describing the semantics of behaviors, but they are inadequate for characterizing the full semantics. For now, we assume that a proper semantic specification mechanism exists. In the current model design, a behavior is specified only by its signature. However, the extensibility of the model makes it easy to incorporate a more complete semantic specification when one is developed. The only extension required is to modify the implementation of the *B_semantics* behavior on **T_behavior** to correspond to the new, more

complete semantics. We are currently investigating specification techniques and denotational semantics as a complete semantic description mechanism for behaviors.

Behaviors are applied to objects. The application of a behavior, say $B_population$, to an object, say `Edmonton`, using some arguments, say a_1, \dots, a_n , can be denoted by $(B_population(Edmonton))(a_1, \dots, a_n)$ or by use of the dot notation `Edmonton.B_population(a_1, \dots, a_n)`. In either case, the object `Edmonton` is called the *receiver* of the behavior.

Behaviors are instances of the type `T_behavior` and functions are instances of the type `T_function`. We use an arrow “ \rightarrow ” in function type specifications and curry multiple argument functions. A function type is of the form $\mathcal{A} \rightarrow \mathcal{R}$ where \mathcal{A} represents the argument type expression of the function and \mathcal{R} represents the result type. In general, the argument and result types may be any type specification, including a function type. Then, by currying, multiple argument functions may be specified.

As defined in more detail in Section 2.3, types are related to each other via *subtyping* (also referred to as *behavioral inheritance*). A behavior defined on a type `T_x` is *inherited* in the type if and only if the behavior is defined in a supertype of `T_x`. A behavior defined on a type `T_x` is *native* in the type if and only if the behavior is not defined in any supertype of `T_x`.

Inherited behaviors do not necessarily borrow their implementation from their supertypes (although this can be set as the system default). Therefore, we define a separate reuse mechanism for implementations called *implementation inheritance*. An implementation of a behavior in a type `T_x` is *inherited* if and only if the behavior is inherited and the function implementing the behavior in `T_x` is the same as a function implementing the behavior in a supertype of `T_x`. Otherwise, the implementation of the behavior is *redefined* (or *overridden*) in `T_x`.

TIGUKAT supports *multiple subtyping*. However, the separation of behaviors from functions introduces the need for separating behavioral inheritance from functional inheritance and defining separate conflict resolution schemes for both. Implementation inheritance conflicts are resolved using an approach similar to the one used in Modular Smalltalk [WBW88]. Specifically, it is an error for a type to inherit two different implementations (i.e., two instances of `T_function`) for the same inherited behavior. The error is resolved by explicitly redefining⁵ the `T_function` for that behavior. Note that one choice for redefinition is one of the two conflicting `T_functions`. No separate mechanism is required to solve inheritance conflicts between instance variables because there are no instance variables. Stored function conflicts are resolved in the same uniform manner as computed function conflicts. Furthermore, in the context of a complete behavioral semantics, there are no behavioral inheritance conflicts. That is, the inherited behavior in the multiple supertypes will be semantically equivalent or not. When they are equal, only one behavior is defined in the subtype. When they are not equal, multiple behaviors are defined in the subtype.

2.2 Objects

An *object* is a fundamental primitive in TIGUKAT because the conceptual level of the model deals uniformly with objects. Objects are defined as unique (*identity, state*) pairs where *identity* represents a unique, immutable system managed object identity (or *oid*) and *state* represents the information carried by the object. There are system defined mappings $oid(o)$ and $state(o)$ that accept an object o and return the *oid* or *state* of o , respectively. These are internal mappings used only by the system and are not visible to the user. The existence of unique oids does not preclude application environments such as object programming languages from having many *references* (or *denotations*) to objects, which need not be necessarily unique and may even change depending on the scoping rules of the application.

In TIGUKAT, every object can be viewed as a *composite* object, meaning every object has references/relationships (not necessarily implemented as pointers) to other objects. These other objects are returned as results of behavior applications, but it does not matter whether the behaviors are implemented by stored or computed functions. For example, even integers are composite objects since they have behaviors that return objects.

⁵Redefinition may be the explicit writing of a new function or simply choosing an existing function.

Object existence, access, and manipulation in TIGUKAT is based on the notions of *reference*, *scope* and *lifetime*. This is similar to other model proposals (e.g., [Sny90, Ken90, FKMT91]) in that the only user-expressible representation of an object is a *reference* within a particular *scope*. A *scope* defines the visibility, access paths, and lifetime of object references. The *lifetime* of an object is independent of the *lifetime* of a reference to that object within a particular scope. That is, when a reference to an object disappears at the end of a scope, the object being referenced does not necessarily disappear along with it. This can depend on the definition of the scope and the persistence of the object. From the database perspective there is also the issue of explicit deletions and the dangling reference problem that follows. That is, when an object is explicitly deleted, all references to that object should no longer point to the object and somehow be invalidated. In TIGUKAT, every behavior application is a reference to an object. Thus, we do not invalidate references, but rather bind them to an object whose type is `T_null`. That is, when an object is explicitly deleted, the object is changed to an instance of type `T_null` (called *undefined*) so that all references to it remain valid. Garbage collection is used to reclaim the storage of deleted objects. The deletion semantics is explained in more detail in Section 4 since a similar approach is used when a persistent object is made transient. The similarity stems from the fact that subsequent programs will not see the persistent object that was made transient and it will appear as though the object was deleted. Another condition for object deletion and storage reclamation is if an object no longer has references through its class.

Operations on objects are performed through behaviors and object access is specified through references. Therefore, an operation on an object reference in a particular scope represents the application of a behavior to the actual object that is referenced. We define several behaviors on the type `T_object` that are inherited by all types and, therefore, are applicable to every object. A mechanism is required to determine if two object references refer to the same object. This requirement is met by the behavior `B_equal`. For any two object references R_i and R_j , the result of $(B_equal(R_i))(R_j)$ is true if and only if $oid(R_i)$ and $oid(R_j)$ map to the same object identity. The above operation is more commonly specified as $R_i = R_j$.

This is the only kind of equality that the primitive model defines. It is quite strong in that the only way two object references are equal is if they refer to the same object (with the same identity). Our notion of object equality is the same as “identity equal” defined in [KC86] or “0-equality” defined in [LRV88]. We do not define, at this level, any notions of *shallow* or *deep* equality found in other models [KC86, LRV88, Osb88] or extended versions of these, which determine equality at various levels [SZ90]. These notions can be defined as equivalence relationships on the behavioral characteristics of objects and, therefore, should be left to customized interpretations at the behavioral level rather than being part of the primitive model definition. For example, one may define person equality based on the equality of their social insurance numbers. The implementation of `B_equal` in a type `T_person` can be overridden to implement this semantics.

Objects in the model are strongly-typed⁶ in the sense that each object is associated with a single type. A type defines all the behaviors applicable to the objects of the type. The `B_mapsto` behavior, when applied to object o , returns the type of that object. It is important in type-checking and query processing to know the type of an object [SÖ90b].

Another behavior defined on `T_object` is the identity mapping behavior `B_self`, which maps every object to itself. That is, for any object o , $B_self(o) = o$. There are additional behaviors whose presentation depends on other primitive concepts. We introduce them as these concepts are defined.

2.3 Types

A *type* defines behaviors and encapsulates hidden behavioral implementations (including state) for objects created using the type as a template. The behaviors defined by a type describe the *interface* for the objects of that type. Types are organized into a lattice structure using the notion of *subtyping* which promotes software reuse and incremental type development. Since TIGUKAT supports multiple subtyping, the type structure is potentially a directed acyclic graph (DAG). However, this DAG is transformed to a lattice by rooting it at `T_object` and *lifting* with the primitive type `T_null`.

⁶Note that this differs from another common meaning of strong typing that refers to static type-checking.

The uniformity of TIGUKAT implies that types are also objects with their own state and identity along with their own type. The type which describes all type objects is **T_type** and it is accessible in the same manner as any object. Thus types, in addition to serving as descriptions of objects, are objects themselves and the type **T_type** serves as a description for all other types, including itself. This is known as the **type: type** property [Car86] in programming languages. The state of a type object consists of a structural specification of its instances (a template), references to the encapsulated behaviors it defines, references to its subtypes and supertypes, and a reference to its associated class.

Two relationships on types have been identified [ÖSP94]. One is the concept of a type *specializing* another type in a manner similar to what is described in [MZO89]. The other is the more popular, and stronger, notion of explicitly creating a type to be a *subtype* of another type [Car84]. A type **T_1** *specializes* a type **T_2** if **T_1** defines all the behaviors of **T_2** (and possibly more). A type **T_1** is explicitly created as a *subtype* of a type **T_2** which means **T_1** specializes **T_2** and all the instances of **T_1** are also instances of **T_2**. Thus, subtyping implies specializes and defines a subset inclusion relationship on type extents. Conversely, specializes does not imply subtyping. Furthermore, subtyping supports *substitutability* [SZ90] because an object of type **T_x** can be used (substituted) in any context specifying a supertype of **T_x**. Specialize on its own does not support substitutability. Specialize is a semantic property derived from the behaviors defined on types, while subtyping is an explicit use of this property to define a partial order on types and a subset inclusion relationship on their extents.

A behavior is required on types that determines the class of a given type. In order to create objects of a particular type, there must be a class associated with the type to manage its instances. However, types do not require an associated class if there are no instances of that type (e.g., *abstract types*). **T_type** defines behavior *B_classof* for accessing the unique class (if it exists) associated with a particular type.

2.4 Classes and Collections

A *class* ties together the notions of *type* and *object instances*. The entire group of objects of a particular type, including its subtypes, is known as the *extent* of the type and is managed by its *class*. We refer to this as the *deep extent* and introduce a *shallow extent* that refers only to those objects created from the given type without considering its subtypes. The deep extent imposes a subset inclusion relationship on classes. We refer to this as *subclassing*, which has a direct relationship to subtyping on types. That is, a class **C_x** is a *subclass* of a class **C_y**, meaning the deep extent of **C_x** is a subset of the deep extent of **C_y**, if and only if the type associated with **C_x** is a subtype of the type associated with **C_y**.

Objects of a particular type cannot exist without an associated class and every class is uniquely associated with a single type. Another feature of classes is that object creation occurs only through a class using its associated type as a template for the creation. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes* which imply *types*. Defining object, type, and class in this manner introduces a clear separation of these concepts. This separation is important in schema evolution which manipulates type objects into new subtype relationships and need not be concerned with the overhead of classes. Furthermore, many object-oriented systems include *abstract types* whose sole purpose is to serve as placeholders for common behaviors of subtypes and are never intended to have any instance objects. In these cases, there is no reason to manage classes for abstract types, because there are no instances of these types.

We define a *collection* as a general user-definable grouping construct (other constructs include *bags* for maintaining duplicates, *posets* for partially ordered collections, and *lists* that encompass both properties). A *collection* is similar to a *class* in that it groups objects, but it differs in the following respects. First, object creation may not occur through a collection; object creation occurs only through classes. This means that collections only form user-defined groupings of existing objects. Second, an object may exist in any number of collections, but is a member of the shallow extent of only one class. Third, the management of classes is *implicit* in that the system automatically maintains classes based on the type lattice whereas the management of collections is *explicit*, meaning that the user is responsible for their extents. Finally, a class groups the entire extension of a single type (shallow extent), along with the extensions of its subtypes (deep extent). Therefore, the elements of a class are homogeneous up to inclusion polymorphism. A collection is

heterogeneous in the sense that it can contain objects of types unrelated by subtyping. Furthermore, there is no distinction between shallow and deep extents for collections.

In TIGUKAT, we define **T_class** as a subtype of **T_collection**, which introduces a clean semantics between the two and allows the model to utilize them in an uniform way. For example, the targets and results of queries are typed collections of objects and since classes are a subtype of collection, they may be used in queries as well. This approach provides flexibility and expressiveness in formulating queries and gives *closure* to the query model, which is often regarded as an important feature [Bla91, YO91].

2.5 Higher Level Constructs and Reflection

The types **T_class-class**, **T_type-class**, and **T_collection-class** in Figure 2 make up the *meta* type system. Their placement within the type lattice is in direct support of the extensibility of the model. Identifying characteristics of the meta-model are its ability to uniformly represent meta-information as first-class objects with well-defined behavior and to maintain the behavior application abstraction on these constructs. This means that all properties of the model apply to this higher-level information uniformly. This property has been referred to as *reflection* [PÖ93].

The higher-level objects are called *meta-objects* because they provide support for other objects. For example, **T_type** provides support for types and **C_class** manages the class objects in the system. These meta-objects are uniformly managed by means of the primitives. This is possible through the introduction of higher level constructs called *meta-meta-objects*. Our model defines a three tiered structure for managing objects as 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 instance relationships with the head of the arrow being the instance and the tail being the class it belongs to.

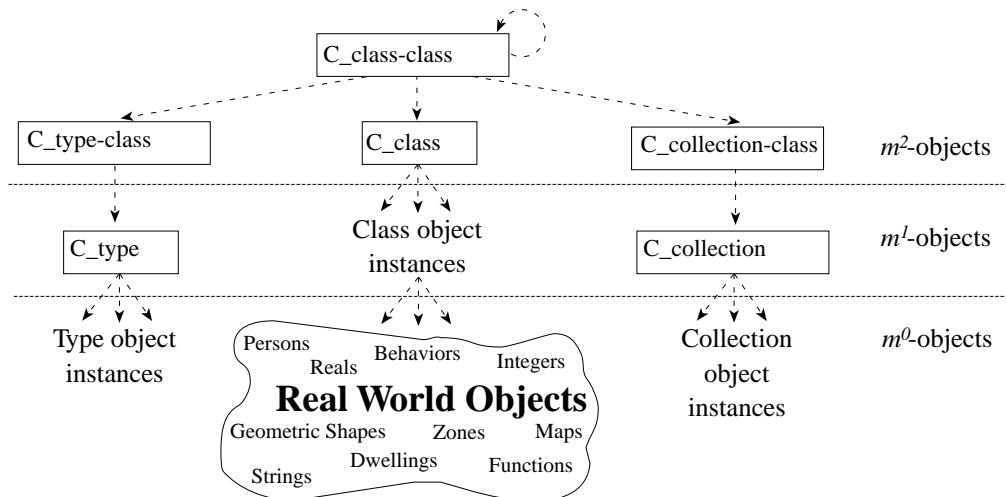


Figure 3: Three tiered instance structure of TIGUKAT objects.

The lowest level of our instance structure consists of the “normal” objects that depict real world entities such as *integers*, *dwellings*, *maps*, *behaviors* and so on. Type and collection objects also reside at this level, which illustrates the uniformity in TIGUKAT. We define this level as m^0 and classify its objects as m^0 -objects. The second level defines the class objects whose associated types maintain schema information for the objects below it. These include **C_type**, **C_collection** and most other classes in the system. This level is denoted as m^1 and its objects are m^1 -objects. The reasons for placing the classes at this level are that classes maintain the objects of the system (objects cannot exist without classes) and classes are associated with types that define the schema information of their instances (classes cannot exist without types). Thus, classes represent the binding management between objects and the operations that can be performed on them

as defined by their type. The upper-most level consists of the meta-meta-information (labeled m^2), which defines the functionality of the m^1 -objects and is used to give definitional properties to these objects. The structure is closed off at this level because the m^2 -class **C_class-class** is an instance of itself as illustrated by the looped instance edge.

In the following discussion, we show the interactions among the various levels of the structure and how they contribute to the uniformity of TIGUKAT, which in turn forms the foundation for reflection. We refer the reader to the primitive type lattice in Figure 2 and a portion of its companion primitive class lattice shown in Figure 4. Each **C_x** class in Figure 4 is associated with the corresponding **T_x** type in Figure 2.

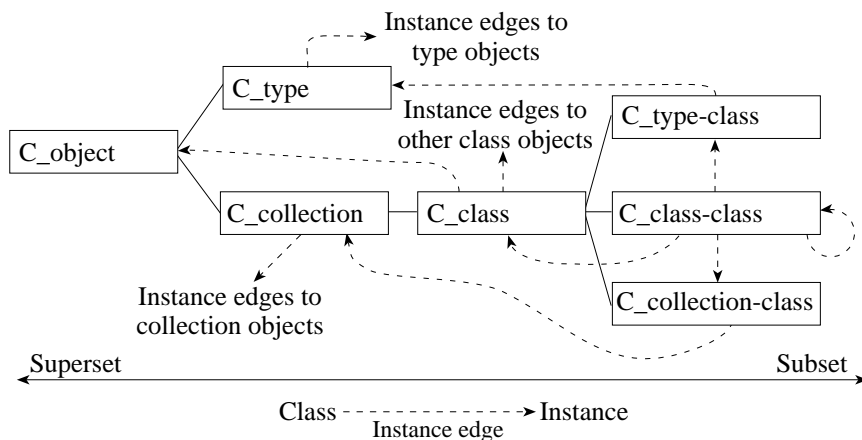


Figure 4: Subclass and instance structure of m^1 and m^2 objects.

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, we explain the relationships between these classes and their instances. 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_city** is an instance of the class **C_class** and **C_city** is a collection of city objects. The deep extent of **C_class** manages all classes in the system such as **C_object**, **C_type**, **C_city**, 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.

This meta-architecture is sufficient for managing all objects, including meta-information, in a uniform way. This provides the foundation for reflective capabilities such as the support for class behaviors and reflective queries. To support class behaviors, each class can be made an instance of its own meta-class instead of the common meta-class **C_class**. For example, to define a class behavior $B_{averageAge}$ on **C_person** that computes the average age of the persons, we can uniformly extend the meta-model by creating a type **T_person-class** as a subtype of **T_class**, defining the behavior $B_{averageAge}$ on **T_person-class**, creating an m^2 -class **C_person-class** as the associated class of **T_person-class**, and creating **C_person** as an instance of **C_person-class**. Now, we can create person instances of **C_person** in the usual way and $B_{averageAge}$ is applicable to **C_person** and returns the average age of all persons in **C_person**. We can define many other class behaviors on **T_person-class**, including various object creation and initialization behaviors. This approach is in contrast to the usual way of making **C_person** a direct instance of **C_class**. If this is done, it is difficult to define class behaviors for **C_person** since **C_class** typically has many class

instances and any class behavior defined on `T_class` would apply to all class objects. Our approach is superior to an approach that defined an extra m^2 -class for every class (e.g., Smalltalk), since it has smaller space overhead.

More powerful extensions are also possible. For example, although `C_person-class` is a separate m^2 -class for `C_person`, it can be used to group other related classes such as `C_student`, `C_employee`, etc., simply by creating them as instances of this class. Behavior `B_averageAge` would then be applicable to all these additional classes. Our approach provides a good balance between the flexibility of defining class behaviors with the efficiency of grouping common classes under a single m^2 -class.

Reflective queries can be expressed naturally in TIGUKAT without any meta-level extensions to the query languages. The reason is that the query model incorporates the behavioral paradigm of the object model and since the meta-system is uniformly represented by objects with well-defined behaviors, the meta-objects can be used in queries just like any other objects. For example, it is natural (through behavior applications) to express a query that returns the types that define a behavior `B_age` with the same implementation as one of its supertypes. Additional examples include a query that returns a collection of all types that don't have an associated class (i.e., all abstract types), a query that returns types that define a certain implementation for a certain behavior, a query that returns the classes that have a greater cardinality than all other collections in the system, and so on. Moreover, we can use reflection to infer the result type of a query during its execution. An example reflective query is given in Section 5.

Our meta-class structure is similar to ObjVlisp [Coi87] and is a generalization of the one-to-one class/meta-class architecture of Smalltalk [GR89]. The generalization of Smalltalk stems from the fact that we do not necessarily define an m^2 -class for every class, which is required in Smalltalk. We can group several classes under a common m^2 -class. Full details of the reflective features of TIGUKAT and its comparison with other meta-models are presented in [PÖ93].

The introduction of the m^2 -objects adds a level of abstraction to the model that encapsulates the schema as first-class objects. The benefit of this approach is that the entire model is consistently and uniformly defined within itself. Every object has well-defined behavior and, therefore, we can uniformly apply behaviors to the higher-level objects.

2.6 Temporality

Temporality is introduced into TIGUKAT through an extensible set of primitive *time* types. A rich set of behaviors are defined on these types to model the various notions of time elegantly [GM93].

We use the concept of a *timeline* to represent an axis over which time can be specified. A timeline is comprised of a collection of time references. A time reference is a means by which time can be specified (e.g., 5 seconds, t3, July 31, [1967,1968], 3 years, 9:17:54:20). We have identified three basic types of time references: a time *instant* (*moment*, *chronon*, etc.), a time *span* (*duration*), and a time *interval*. These are used to construct instant timelines, span timelines, and interval timelines.

We can model different kinds of timelines depending on (i) their domain (*discrete*, *dense*, or *continuous*), (ii) their boundedness (*bound* or *infinite*), and (iii) their ordering (*linear* or *branching*). Any combination of these three features is possible in forming a timeline. This gives applications built on TIGUKAT substantial flexibility in choosing timelines to suit their needs.

Behavior histories are used to manage the properties of objects over time. A subtype of `T_behavior` is introduced to specialize behaviors with temporal qualities for managing histories. Instances of this subtype are called *temporal behaviors*. Temporal behaviors specialize non-temporal ones and, thus, encompass all the functionality of non-temporal behaviors. This introduces *temporal transparency* in the sense that a temporal behavior can be used anywhere a non-temporal behavior is expected. In other words, a user unconcerned with temporality can use temporal behaviors as if they were non-temporal. This has the benefit of integrating temporal applications smoothly into an existing system.

Temporal behaviors have the ability to manage *valid time* histories (when a value for the behavior is valid) and *transaction time* histories (when a value for a behavior is committed to the objectbase) independently. Our approach adheres to the well recognized orthogonal nature of these two times [SA85] and allows us

to support valid time, transaction time, and bitemporal models.

2.7 An Example System Design

In this section, we present the design of a simplified geographic information system (GIS). This example is used throughout this paper to demonstrate various features of TIGUKAT. The GIS example is selected because it is usually listed among the application domains which require the advanced features offered by object-oriented technology.

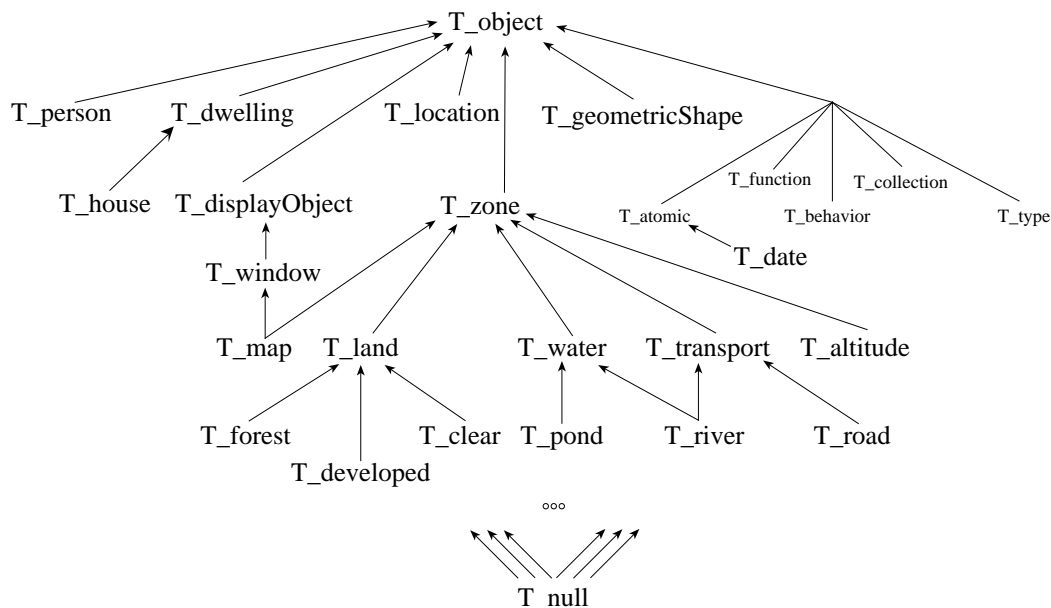


Figure 5: Type lattice for a simple geographic information system.

A type lattice for a simplified GIS is given in Figure 5. 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 the types **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 the type **T_location**, the type **T_zone** along with its subtypes which categorize the various zones of a geographic area, and the type **T_map** which defines a collection of zones suitable for displaying in a window. Displayable types for presenting information on a graphical device are defined. These include the types **T_displayObject** and **T_window** which are application independent, along with the type **T_map** which is the only GIS application specific object that can be displayed. Finally, the type **T_geometricShape** defines the geometric shape of the regions representing the various zones. For our purposes we will only use this general type, but in more practical applications this type would be further specialized into subtypes representing polygons, polygons with holes, rectangles, squares, splines, and so on. Table 1 lists the signatures of the behaviors defined on GIS specific types.

3 Implementation Considerations

The persistence issues related to the implementation of TIGUKAT are discussed in the next section. In this section, we discuss some of the other issues that arise in the implementation of a uniform and generic object

Type	Signatures
T_location	<i>B_latitude</i> : T_real <i>B_longitude</i> : T_real
T_displayObject	<i>B_display</i> : T_displayObject
T_window	<i>B_resize</i> : T_window <i>B_drag</i> : T_window
T_geometricShape	
T_zone	<i>B_title</i> : T_string <i>B_origin</i> : T_location <i>B_region</i> : T_geometricShape <i>B_area</i> : T_real <i>B_proximity</i> : T_zone → T_real
T_map	<i>B_resolution</i> : T_real <i>B_orientation</i> : T_real <i>B_zones</i> : T_collection(T_zone)
T_land	<i>B_Pollutants</i> : T_collection(T_string)
T_water	<i>B_volume</i> : T_real <i>B_Pollutants</i> : T_collection(T_string)
T_transport	<i>B_efficiency</i> : T_real
T_altitude	<i>B_low</i> : T_integer <i>B_high</i> : T_integer
T_person	<i>B_name</i> : T_string <i>B_birthDate</i> : T_date <i>B_age</i> : T_natural <i>B_residence</i> : T_dwelling <i>B_spouse</i> : T_person <i>B_children</i> : T_person → T_collection(T_person)
T_dwelling	<i>B_address</i> : T_string <i>B_inZone</i> : T_land
T_house	<i>B_inZone</i> : T_developed ^a <i>B_mortgage</i> : T_real

^aBehavior was refined from supertype T_dwelling.

Table 1: Behavior signatures pertaining to example specific types of Figure 5.

model such as TIGUKAT. There are three issues that we discuss: the implementation of the primitive type system, behavior application, and the implementation of behavioral and implementation inheritance. For more details, the reader is referred to [Ira93].

3.1 Implementation of Primitive Type System

TIGUKAT is implemented in g++ which is GNU's implementation of C++. However, since TIGUKAT has a generic object model, there is no one-to-one mapping between TIGUKAT types and C++ classes (i.e., we do not create a C++ class for each TIGUKAT type that is defined). Instead, there exists a single foundation C++ class, **TgObject**, which is the principal template for instantiation of all TIGUKAT objects. That is, every TIGUKAT object (*type* object, *class* object, *behavior* object, *collection* objects, *function* object, *instance* object, *atomic* object, and other primitive or user-defined object) is an instance of this fundamental C++ class. This approach ensures the uniform representation of all objects in the system since they may each be treated as an instance of **TgObject**. The TIGUKAT type, class, etc. semantics is embedded within the **TgObject** structure. Following this approach, the TIGUKAT model is implemented within itself.

From the structural viewpoint, every instance of **TgObject** comprises of an array of records as depicted in Figure 6. These can be thought of as the *attributes* (data fields) of that particular instance. **TgObject**

is a dynamic array where each element is either an integer, a character, or `TgObject`. Integers, reals and characters are stored directly while all other objects, including the atomic objects such as sets, strings, bags, lists and posets, have only references to them stored in the slots. This decision was made to ensure efficient use of memory. For any object, the first slot always contains a pointer to that object's type which was the template used for its creation. Thus, in line with the model, every object carries knowledge about its type.

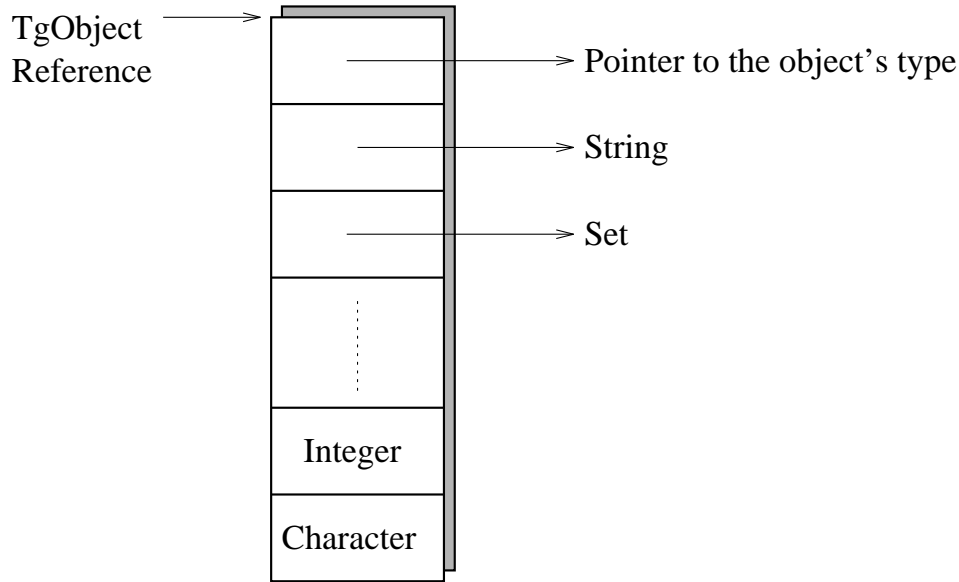


Figure 6: Representation of the Generic `TgObject` Structure

To implement uniform treatment of everything as first-class objects, we have implemented different kinds of C++ object instances in the system viz. *type* objects, *class* objects, *object* objects, *behavior* objects, *function* objects, *collection* objects and *atomic* objects. Although these template instances all are `TgObjects`, they differ in their structural contents. For example, a type object has a fixed number of slots dedicated for maintaining information such as its corresponding class (implemented as a reference to another C++ instance which is a *class* object), its subtypes set (reference to a C++ set instance), its supertypes, etc. We do not discuss the detailed data structures of each of these objects; we will only discuss the structure of *type* objects since this information is relevant to the subsequent discussion on behavior application and inheritance implementation.

3.2 Behavior Application

Dispatching is the process by which the application of a behavior on an object (message sending) is bound to a particular function (implementation of that behavior). In the event that the applied behavior's implementation is not clearly evident (as a result of subtyping), the right function associated with that applied behavior for the type of the receiver object must be invoked. This requires what is called *dynamic binding*. Behavior application thus involves the retrieval and application of an appropriate piece of binary code that is contingent on the receiver's type and the selector for that behavior.

Dispatching may be considered as a special case of what is called *resolution* [ZM90]. *Resolution* has been defined as a runtime interpretation process that selects a particular value from a possibly ambiguous set of values. Method dispatch (behavior application), hence, seeks to select an appropriate function object (method) whose code needs to be executed, from a set of function objects each of which implement the same named behavior object over different types. In order to correctly make this decision some additional information (actual type of the receiver and the method selector) relevant to the context is required.

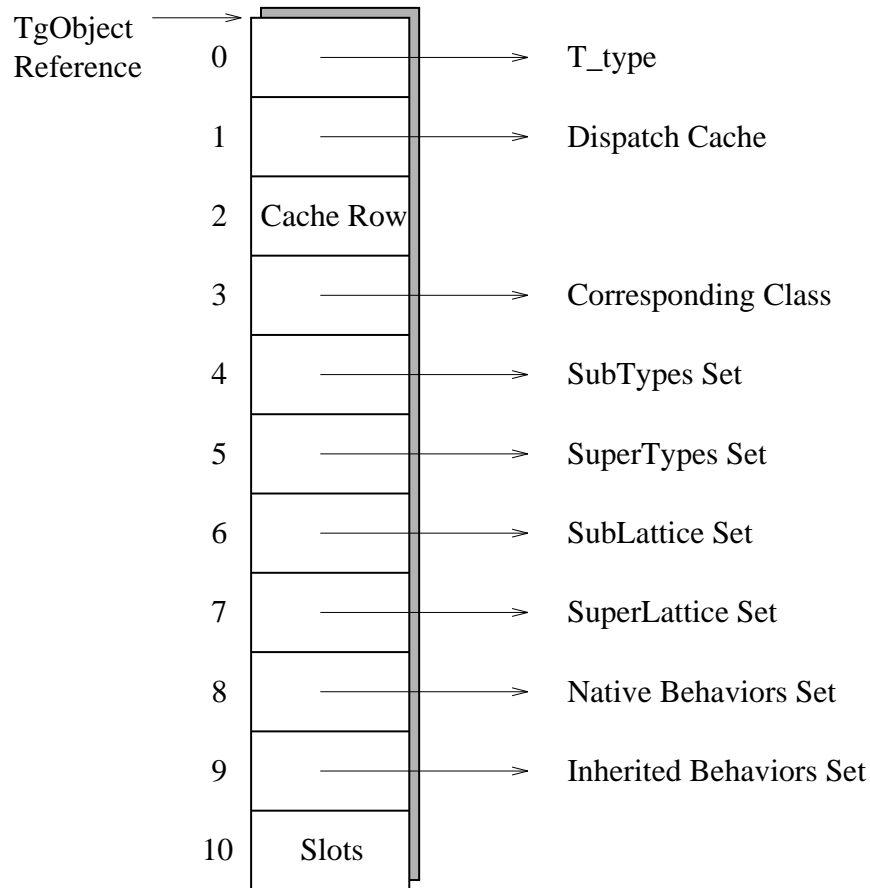


Figure 7: The Type Object's Structure

Since behavior application is such a fundamental operation in TIGUKAT, it is important to have an efficient dispatch implementation. We have opted for a relatively simple but fast mechanism at the cost of bearing the consequential memory overhead. The system maintains a *dispatch cache* which consists of a slot for each behavior-type pair that exists in the system. This cache is a statically allocated volatile structure which needs to be reinitialized on program startup. The size of this lookup table is accordingly proportional to the total number of unique behaviors in the system and the total number of types in existence. We sacrifice memory usage for quick response time during execution, but as proposed in [AR92, DMSV89], an incremental coloring algorithm could be used to drastically reduce memory consumption. We have not implemented this optimization in the current version of TIGUKAT.

Each entry in the dispatch cache is a function pointer to some executable code which implements that behavior (column) for the concerned type (row). Every unique behavior has a unique integer mapping associated with it. We call this integer mapping the *method selector*. The method selector provides access to the appropriate column of the cache. That column is said to “belong” to the behavior. The addresses stored in the slots of this column may be different or identical, depending on which of the subtypes have inherited the same implementation of that behavior and which have had that behavior redefined, overridden or *reassociated* (associated with a different function). The process of filling the cache row with appropriate values during the creation of a new type has been termed *implementation inheritance* and our system handles it automatically up to a certain degree of complexity as discussed in the next subsection.

Behaviors may be reassociated with functions at any time (redefinition of behaviors) and this makes it

imperative that we support the dynamic binding of behaviors and perform dispatch *on the fly*. Although it is evident that *static* (compile time) dispatching is more efficient [Cat91], this will seldom be possible in our system. The reference to an object of a particular type may potentially be referencing an object of any of this type’s subtypes. The ambiguity about which function should be invoked can only be resolved at runtime when knowledge about which type’s instance is being referenced becomes available. Thus, the actual type of a receiver object needs to be identified prior to function execution. We note that although dynamic binding might render static type checking difficult it does not entirely preclude it.

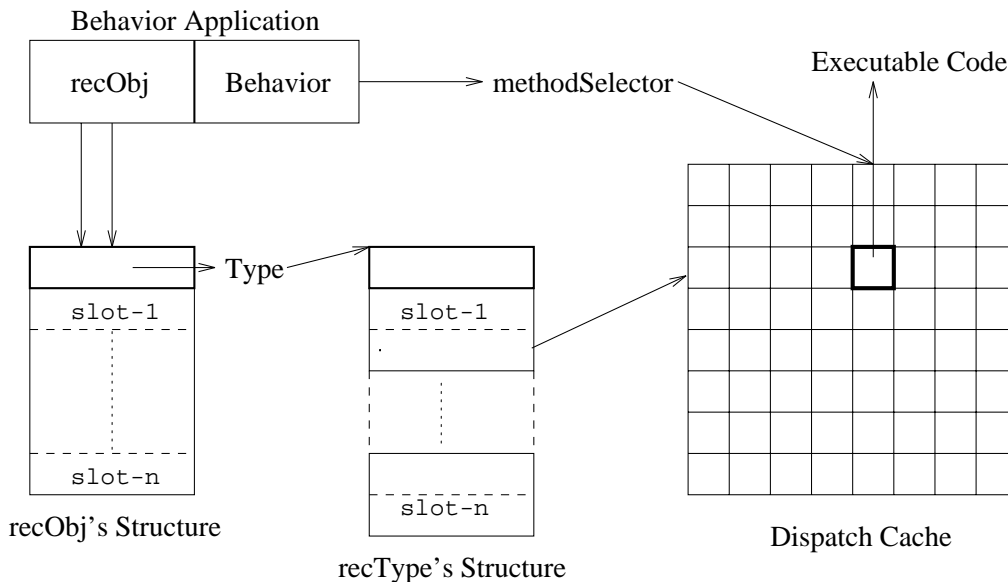


Figure 8: The Behavior Application Process

The behavior application process for computed functions in TIGUKAT involves the following procedure. With reference to Figure 8, given an object, say *recObj*, as the receiver of a particular message, we extract its type, say *recType*, which is readily available since every object knows its type. All types have knowledge of their unique cache row (See Figure 7). From the applied behavior object we extract the method selector, *methodSelector*. This integer value indexes into a unique column in the dispatch cache. The slot in the determined row and column contains the address of the function code to be executed. The list of arguments passed to the behavior is supplied to the function after relevant type checking is done. Behavior application is conveniently reduced to the execution of a single line of code:

$JMP(recObj \rightarrow recType \rightarrow dispatchCache[methodSelector])$

where *recObj* is a pointer to the object on which the behavior is to be applied (receiver object reference), *recType* is the receiver object’s type, *dispatchCache* is the matrix of executable addresses and *methodSelector* gives access to the appropriate column in the dispatch cache. Therefore, the two basic requisites for binding an executable piece of code to the applied behavior at runtime are the type of the receiver object and the method selector for the behavior.

3.3 Behavioral and Implementation Inheritance

As indicated in Section 2.1, two kinds of inheritance are supported by TIGUKAT: *behavioral* and *implementation* inheritance. The implementation strategy for behavioral inheritance (subtyping) involves taking the union of the interface sets of all the types declared as immediate supertypes of the new type being created. This set forms the contents of the new type’s *inherited* set and comprises the minimum set of behaviors that all objects of this type should conform to. The nature of the functions that these behaviors have been

associated with is of no consequence to the behavioral inheritance mechanism. The implemented algorithm iterates through the relevant interfaces and selects all the behaviors with unique signatures as candidates for insertion into the new type’s inherited set. This is a relatively straightforward technique.

Implementation inheritance facilitates code reuse by ensuring that all code is at a level where the maximum number of types can share it [ABD⁺89]. If only single inheritance is present, the inherited set of the new type is precisely the contents of the interface set of its sole supertype. No conflict resolution is necessary and all entries in the dispatch cache and the supplementary cache are merely duplicated in the row allocated for the new type for the complete set of inherited behaviors. This implies that all implementations (function addresses) for the inherited set of behaviors are inherited too. However, the type implementor⁷ has the liberty to reassociate any or all of these inherited behaviors.

With multiple inheritance, the situation is more complex since conflict resolution has to take place. Figure 9 depicts an inheritance graph with multiple subtyping. The arrows indicate a subtyping relationship from the tail to the head and the dotted arrow indicates an instance of the type. The dashed boxes contain the interface sets of the corresponding types while the matrices shown as DC and SC are the dispatch cache and the auxiliary cache, respectively. The auxiliary cache SC is a bit cache that records whether a function is stored or computed. Execution of the stored function simply sets or gets the contents of one of the receiver’s slots without executing any code. In this case, the slot number, rather than the function address, is stored in DC.

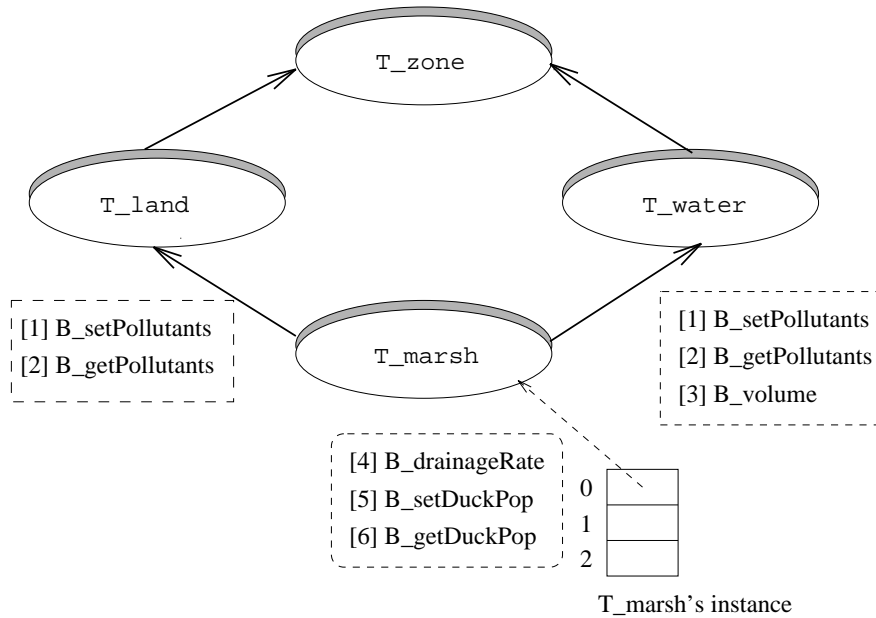
Consider the GIS example that we introduced earlier. We create a new type **T_marsh**, as a subtype of **T_land** and **T_water**, with the native behaviors *B_drainageRate* (to calculate the rate of water leaving or entering the marsh) and *B_DuckPop* (to store the population of ducks in the marsh). This inheritance structure has a clash in behaviors that the system is unable to resolve automatically and requires the type implementor’s intervention. The conflict resolution policy fails because the behaviors *B_setPollutants* and *B_getPollutants* are defined in the interfaces of both the direct supertypes (**T_land** and **T_water** are immediate supertypes of **T_marsh** and have conflicting implementations associated in each of these types, being computed in **T_water** but stored in **T_land** (as depicted in auxiliary cache SC). We have assumed that the type implementor opted for the stored implementations to be inherited and therefore each instance of **T_marsh** requires a total of three slots: slot 0 holds the reference to the type, slot 1 holds the reference to the collection of pollutants, and slot 2 holds the value of the duck population.

We iterate over each of the behavior objects in the inherited interface of **T_marsh** generated during behavioral inheritance. If a behavior exists in only one supertype’s interface, this signifies a conflict-free condition; thus no conflict resolution is required. The implementation for that behavior may be safely inherited together with its associated function (stored or computed). The appropriate entry in the supplementary cache, indicating a stored or computed association, is inserted. If the association is with a computed function then the address of that function is also inserted into the dispatch cache. All the stored functions will possess a NULL entry in the dispatch cache until class creation time. At that time, slots will be assigned to all the stored functions, one slot per pair of set-get accessors. This may require a reallocation of slots to behaviors which is entirely system managed.

For each conflicting behavior, the conflict resolution policy has to be applied. The supplementary cache values for that behavior are examined. If they happen to indicate a computed function for all the conflicting supertypes, the values of the addresses of the functions from the dispatch cache are examined. If these are identical for each of the types in the set of conflicting supertypes, then this behavior’s implementation is safely inherited and the corresponding address is inserted into the dispatch cache. A *computed* indication is placed in the supplementary cache.

If the conflicting behavior is implemented by a stored function in all the entries for the supertypes, the corresponding value of **T_function** is examined for each type. If these match, then a *stored* indicator is placed in the supplementary cache and a NULL is entered into the dispatch cache. Recall that for all the

⁷We identify three classes of users. The *type specifier* is the person who designs the inheritance hierarchy for the user application. The *type implementor* is the one who actually implements this required hierarchy using TDL. The *end user* refers to the person or application program that may query the existing system and instantiate new objects, but may not be authorized to modify the existing type structure.



	1	2	3	4	5	6
T_zone						
T_land	1	1				
T_water						
T_marsh	1	1			2	2

DC

	1	2	3	4	5	6
T_zone						
T_land	s	s				
T_water	c	c	c			
T_marsh	s	s	c	c	s	s

SC

Figure 9: Implementation Inheritance Requiring Conflict Resolution

stored functions, the dispatch cache will hold the corresponding slot number to access (an identical value for each paired set-get) instead of the address of the executable code. These slot numbers will only be inferred and allocated during class creation, at which time it will be possible to determine the total number of all the associated stored functions.

In the event that an inherited behavior is associated with a stored function in one of the supertypes and a computed function in another, or there is mismatch in the values of function pointers, then no conflict resolution is possible by the system and a NULL is entered in both caches. It is the type implementor's responsibility to associate this behavior with an appropriate implementation of his choice or to specify which of the supertype's implementations is to be inherited. A message requesting intervention will be displayed. The cache values for this behavior must be inserted (i.e. each behavior must be associated with some function) before class creation in order that the newly established type be considered functionally complete.

4 Persistence Model

A fundamental decision governing the implementation of an OBMS is the strategy employed for managing persistent objects. *Persistence* is defined as the ability of an object to survive across multiple application program executions and a *persistent object* is one that has this property. The persistence model of TIGUKAT adheres to the following principles:

1. *Persistence is transparent to the user.* TIGUKAT Query Language (TQL) and TIGUKAT Control Language (TCL) provide a declarative specification for indicating that an object is persistent. Users do not perform any explicit input/output operations, nor do they open and close files. TIGUKAT coordinates with the low-level storage manager to provide persistence transparently.
2. *Persistence is orthogonal to the type of an object* [AB87]. A type can be made persistent or transient. The instances of a type can be either persistent and transient. The only dependency is that if an object is made persistent, then its type must also be made persistent because an object cannot exist without a type. These are described as *persistence side-effects* (PSEs) below.
3. *Persistence is independent of the query model* [AB87]. Queries do not differentiate between transient and persistent objects. Both are queried in a uniform way using the same language constructs. This principle is followed in the development of a programming language interface to TIGUKAT.

Five basic approaches to persistence have been identified [ZM90]. The first strategy requires that a decision about persistence be made prior to object creation. Depending on whether a persistent or transient object is needed, an appropriate object creation routine is invoked on the object. Thus, there are separate routines for creating transient and persistent objects.

The second approach is called *reachability* based persistence. This methodology, pioneered in PS-Algol [ABC⁺83] and incorporated by O₂ [BDK92], requires that persistent objects *hang* off a persistent root via a direct or indirect reference. When an object *o* is made persistent, all objects in the transitive closure (i.e., reachable from *o*) are made persistent. Object *o* becomes a root for persistence. In this scheme, every object reachable from a root is made persistent or transient when the root is made persistent or transient.

The third approach is *allocation* based persistence. This approach restricts the persistence of an object by requiring it to be allocated within a persistent container (collection) during object creation. This requires the existence of a persistent storage space with variables naming locations within that space. Objects written into persistent variables are guaranteed to be persistent as long as they are maintained in the persistent variable. ObjectStore [LLOW91] takes this approach, although it renders garbage collection difficult due to the dangling references problem.

The fourth approach is *type-based* persistence where some types are declared to be persistent and an object is persistent if it is an instance of a persistent type. The E language [RC89, SCD90, RCS89] uses a similar approach and maintains a parallel hierarchy of persistent and corresponding non-persistent types.

The fifth approach (which we follow) associates persistence with individual objects and requires explicit declaration of persistence, which may occur anytime during an object's existence. We define a primitive behaviors *B_persistent* and *B_transient* on `T_object` that are applicable to all objects in the system. This behaviors coerce the receiver object to be persistent or transient, respectively. The TIGUKAT user languages provide declarative constructs for making individual objects or collections of objects persistent or transient. The system translates these requests to applications of *B_persistent* or *B_transient* on the affected objects.

We opted for object based persistence because it best maintains the uniformity of object access and does not restrict the use of types for persistent or non-persistent purposes. Any object created during a session (either a query session or an application program execution session encapsulated as a transaction) can be explicitly made persistent (or transient) at any time during the session. Thus, all TIGUKAT objects are *potentially persistent*.

The support for persistence is a behavioral extension to the model. Behaviors *B_persistent* and *B_transient* are added to the type `T_object` and, thus, are applicable to any object. This clarifies the fact that all objects are potentially persistent (or transient) in TIGUKAT. The language constructs for persistence in TQL and TCL invoke these behaviors.

TIGUKAT queries operate on collections and return collections as results. Since collections are objects, we permit the existence of persistent as well as transient collections. Transient and persistent collections can contain a mixture of both transient and persistent member objects. The transient members of a collection must cease to exist at the end of a particular session, even if the collection is persistent. On the other hand, the persistent members of a collection must continue to exist in their respective class extents after a

particular session ends. This is true even if the collection is transient and ceases to exist at the end of a session. This does not cause a problem since the persistent objects in a transient collection reside in the (persistent) class associated with the type of these objects. Thus, these objects are available following the session even if the collection is not. All collections generated as a result of query execution are initially transient. The semantics of handling each case of transience and persistence of objects, collections, classes, and types are described by implementations for the *B_persistent* and *B_transient* behaviors, which we discuss below.

Coercing an object to be persist might result in *persistence side-effects* (PSEs), which propagates persistence to type and class objects related to the original object. The *persistence matrix* shown in Figure 10 depicts the various alternative strategies involved in making a TIGUKAT object persistent. Reading across rows, a “+” entry indicates a PSE while a “-” entry indicates PSE-free persistence (the diagonal entries are not a concern and, therefore, are PSE-free). Making a *type* persistent is PSE-free. Its corresponding class and instances, if they exist, are not required to be persistent. If a *class* object is made persistent, a PSE occurs which makes its corresponding type persistent as well. However, the instances of this class do not need to be made persistent. The final case is when a particular instance object is made persistent. This causes PSEs that make both its class and its corresponding type persistent. This protects against the object being stored as a persistent instance of a transient type and sometime later being erroneously accessed as an instance of a non-existent type (if its transient type disappears in the meantime). The primitive types, classes, behaviors, and functions are by default perpetually persistent and cannot be deleted. This is necessary for the integrity of the system.

	Type	Class	Inst
Type	X	-	-
Class	+	X	-
Inst	+	+	X

Figure 10: The persistence matrix.

The compliment of the persistence matrix is the *transience matrix* (not shown). This matrix derives the repercussions of making persistent objects transient (by applying a behavior *B_transient* defined on `T_object` for example). The effects are precisely the opposite of those described in the persistence matrix (i.e. making an instance transient will not effect its type or class, making a class transient does not effect its corresponding type but all its instances will be made transient, and making a type transient will make its corresponding class and all its instances transient).

This model of persistence is fairly low-level and the referential integrity between objects is a problem to consider. In particular, when a transient object disappears, how are dangling references to this object handled?

One approach is to offload the responsibility onto the application programmer who must update references to transient objects before the end of a session. This approach is unacceptable for obvious reasons. Another approach is to use reachability persistence, which disallows persistent objects to reference transient ones, since, when an object is made persistent, the transitive closure of all objects reachable from that object are made persistent as well. Reachability-based persistence is not useful in a uniform model like TIGUKAT because conceptually, all objects in the entire objectbase are reachable from any object. Consider an arbitrary object. Since every object knows its type, the type of the object is reachable and must be made persistent. Every type knows its class and, therefore, the class is reachable and should be made persistent. Every class knows its instances and all instances of that class are made persistent. Every type knows its supertypes and

subtypes and, thus, the class/instance persistence propagates over the entire lattice and makes all objects persistent.

The approach that we use has the net effect of transforming transient objects into perpetually persistent `undefined` objects at the end of a session (or transaction). This is always safe because `undefined` is an instance of `T_null`, which is a subtype of all types. The substitutability property allows us to use `undefined` anywhere an instance of a supertype is used.

Operations on a TIGUKAT objectbase occur within a given user session (which will be modeled as a transaction when the programming language is developed). A session defines a scope for the transience of objects. There are **save** (commit) and **quit** (abort) statements that can be used in a session. In this sense, a session serves as a simple, flat transaction model. At the end of a session, all transient objects are logically replaced by the perpetually persistent `undefined` object. This can be efficiently implemented by pointer swizzling. That is, we modify the oid mapping so that it appears as though the transient object was written to stable storage at the location where the persistent `undefined` object exists. Then, all persistent objects that referenced the transient object will now reference the persistent `undefined` object and there will be no dangling references.

In this approach, there is the potential for wasted stable storage when a persistent object is made transient. The transformation to the persistent `undefined` object occurs as usual, but we must somehow reclaim the storage occupied by the object when it was persistent. With a central oid to disk address mapping we can simply update this mapping and reclaim the storage immediately. If, however, objects hold the disk addresses directly, then there may be other persistent objects that reference the old disk address and we cannot simply reclaim the space without updating these references. In this case, a garbage collector can be used to manage reference counts and reclaim the storage after all references have been updated. In the meantime, the storage must be transformed into a persistent `undefined` object so that objects referencing it will not see the old persistent object, but rather the `undefined` object. This transformation is easily implemented by encoding the information in the header of the old object on disk.

Our approach to single object persistence and the maintenance of the PSEs are described in the *F_makePersistent* and *F_makeTransient* functions below that serve as implementations for the *B_persistent* and *B_transient* behaviors defined on `T_object`.

F_makePersistent(o)

This is the implementation of the *B_persistent* behavior defined on `T_object`.

INPUT: An object *o* to be made persistent.

```

if object o is transient then
  Call storage manager to write o to stable storage and update log
  Apply B_persistent to the type and class of o:
    o.B_mapsto.B_persistent
    o.B_mapsto.B_classof.B_persistent
  if object o is a class then
    Apply B_persistent to the associated type of the class:
      o.B_typeof.B_persistent
  endif
endif

```

The recursion of the *F_makePersistent* implementation is ended by making primitive `T_type` and primitive classes `C_type` and `C_class-class` perpetually persistent. `T_type` and `C_type` represent the end of the type chain while `C_class-class` represents the end of the meta-class chain. Note that these are the minimal primitive persistent objects. In practice, the recursion is ended much sooner because many more primitive objects are perpetually persistent (like `C_class`, for example).

At commit time (or the end of a session), the transaction management facility ensures that persistent objects are written out to stable storage. No changes are made to persistent objects with respect to references to transient objects. Dangling references are avoided by the transformation described above.

The implementation for the *B_transient* behavior is as follows:

F_makeTransient(o)

This is the implementation of the *B_transient* behavior defined on **T_object**.

INPUT: An object *o* to be made transient.

```

if object o is persistent then
  Call the storage manager to mark object o transient and update log
  if object o is a class then
    Apply B_transient to every member in the shallow extent of the class
  endif
  if object o is a type then
    Apply B_transient to the associated class of the type:
      o.B_classof.B_transient
  endif
endif

```

At commit time, all transient objects are replaced by the persistent **undefined** object. This ensures that there will be no dangling references to the transient objects because persistent objects that reference the transient object will now reference the persistent **undefined** object. Implementation issues of this approach are discussed above.

The explicit deletion semantics for persistent and transient objects are closely related to the *F_makeTransient* implementation and the transient-to-**undefined** object transformation. The reason is that when an object is explicitly deleted, there is still the problem of dangling references to consider. The *B_drop* behavior defined on **T_object** can be used to explicitly delete an object. The deletion semantics is related to schema evolution when the object to be dropped is part of the schema (i.e., a type, class, collection, behavior, or function). Schema evolution is beyond the scope of this paper, but is addressed in [Pet94].

The only difference in the transient object and deleted object semantics is the timing of events. When an object is deleted (whether it be transient or persistent), it is immediately replaced by the persistent **undefined** object, rather than at the end of a session as is the case for transient objects. A simplified implementation of the *B_drop* behavior for deleting objects is defined as follows and the similarities to *F_makeTransient* are apparent:

F_deleteObject(o)

A simplified implementation of the *B_drop* behavior defined on **T_object**.

INPUT: An object *o* to be deleted.

```

Call the storage manager to mark object o as deleted and update log
if object o is a class then
  Apply B_drop to every member in the shallow extent of the class
endif
if object o is a type then
  Apply B_drop to the associated class of the type:
    o.B_classof.B_drop
endif

```

endif

Perform schema evolution operations if o is a schema object

The single object persistence approach can be transitively applied to all objects referenced by the object being made persistent. This can proceed any number of levels until the transitive closure is reached. Thus, we can identify the boundaries for the transitive application of persistence. The lower bound is when only a single object is made persistent (our approach). The upper bound is when all objects in the transitive closure are made persistent (reachability persistence). In a finite objectbase, there are a finite number of levels between these two boundaries. We call the lower limit 0 -persistence, the upper limit n -persistence, and any level between these two i -persistence. For example the **persistent all** construct of TQL and TCL performs 1 -persistence on a collection argument. That is, the collection and all of its members (1 level of reference) are made persistent. We showed that in a uniform model like TIGUKAT, the transitive closure from any object is the entire objectbase and so n -persistence is not useful.

5 Query Model and Language

An identifying characteristic of the TIGUKAT query model is that it is a direct extension to the object model. In other words, it is defined by type and behavior extensions to the primitive model. We define a type **T_query** as a subtype of **T_function** in the primitive type system. This means that queries have the status of *first-class objects* and inherit all the behaviors and semantics of objects. Moreover, queries are functions and can be used as implementations of behaviors, they can be compiled, they can be executed, and so on.

Incorporating queries as a specialization of functions is a natural and uniform way of extending the object model to include declarative query capabilities. The major benefits of this approach are as follows:

1. Queries are *first-class* objects, meaning they support the uniform semantics of objects, they are maintained within the objectbase as another kind of object and they are accessible through the behavioral paradigm of the object model.
2. Since queries are objects, they can be queried and can be operated on by other behaviors. This is useful in generating statistics about the performance of queries and in defining a uniform extensible query optimizer.
3. Queries are uniformly integrated with the operational semantics of the model and, thus, queries can be used as implementations of behaviors (i.e., the result of applying a behavior to an object can trigger the execution of a query).
4. The query model can be extended by subtyping **T_query**. This can be used to specialize the notion of queries into additional types that can be incrementally introduced and developed as new kinds of queries are discovered. For example, we subtype **T_query** into **T_adhocQuery** and **T_productionQuery** and then define different evaluation strategies for both in the query optimizer. *Ad hoc* queries may be interpreted without incurring high compile-time optimization strategies while production queries are compiled once and executed many times.

The languages for the query model include a complete object calculus, an equivalent object algebra, and an SQL3-like user language. The TIGUKAT object calculus is a first-order predicate language. Predicates of the calculus are defined on collections (essentially sets) of objects and a calculus expression returns a collection as a result. This gives the language *closure*. The calculus includes a function symbol for behavior evaluation in order to incorporate the behavioral paradigm of the object model. This allows the specification of *path expressions* (or *implicit joins*) in calculus formulas. The calculus is object-creating and supports a controlled creation and integration of new collections, classes, types, and objects into the existing schema.

The safety of the calculus is based on the *evaluable* class of queries [GT91], which is arguably the largest decidable subclass of the domain independent class [Mak81]. We extend this class by making use of equivalence ($=$) and membership (\in) operators in queries for *object generation*. This alleviates the need of explicit *range* specifications for those variables that can be generated from the given operators.

The TIGUKAT object algebra has a behavioral/functional basis as opposed to the logical foundation of the calculus. Algebraic operators are modeled as behaviors on the primitive type `T_collection`. Like the calculus, the algebra is *closed* in that every algebraic operator works on collections and returns a collection as a result.

The operators of the algebra include typical set operations, a collapse operator for flattening nested collections, a select for returning objects that satisfy a predicate, an operator for applying a series of behaviors to a collection of objects, an operator to project behaviors, an operator for unconditionally combining objects, a join for combining objects based of a join predicate, a generating join for producing objects from other objects and joining the generated objects with the ones from which they were generated, and a reduction operator for separating joined objects into their original components.

The first-order expressiveness of the calculus, its safety, as well as the equivalence of the calculus and algebra are proven elsewhere [Pet94, PÖLS94]. The following section describes the user language of TIGUKAT, with a focus on its constructs for managing persistence and for querying the objectbase.

5.1 TIGUKAT Language

The main function of the TIGUKAT language is to support the definition, manipulation, and retrieval of persistent (and transient) objects in an objectbase. The language consists of three parts: the TIGUKAT Definition Language (TDL), which supports the definition of meta-objects (i.e., types, collections, classes, behaviors, and functions), the TIGUKAT Query Language (TQL), which is used to manipulate and retrieve objects, and the TIGUKAT Control Language (TCL), which supports the session specific operations (open, close, save etc.). We focus on TQL and TCL in this paper; the complete specification of all three languages is given in [PLOS93b, Lip93].

The TIGUKAT query language (TQL) has a syntax based on the SQL3 *select-from-where* structure and a formal semantics dictated by the TIGUKAT object calculus. Thus, TQL combines the power of declarative query languages with object-oriented features in the forum of the international data-speak of SQL. The broad acceptance of SQL as a standard query language in relational databases, together with the current efforts on SQL3 to extend the syntax and semantics with object-oriented features [Gal92] are the main motivations for our SQL basis.

The semantics of TQL is defined in terms of the object calculus. In fact, there is a complete reduction from TQL to the object calculus [Lip93]. In addition, TQL accepts path expressions (implicit joins [KBC⁺89]) in the *select*, *from*, and *where clauses*. Object equality is defined on the primitive type `T_object`, thus explicit joins are also supported by TQL. The results of queries can be queried, since queries operate on collections and always return a finite collection as a result. Queries can be used in the *from* and *where clauses* of other queries (i.e., nested queries). Objects can be queried regardless of whether they are persistent or transient.

Note that the syntax for the application of aggregate functions is not explicitly supported in the current implementation of TQL. However, as the underlying model is purely behavioral, these functions are defined as behaviors on the `T_collection` primitive type, and can be applied to any collection including those returned as a result of a query.

TQL consists of the four basic operations: **select**, **insert**, **delete**, and **update**, along with three binary operations: **union**, **minus**, and **intersect**. In this paper, we only discuss the *select*, *union*, *minus*, and *intersect* statements.

The basic query statement of TQL is the *select statement*, which has the following syntax⁸:

⁸The notation used throughout this section is as follows: all bold words and characters correspond to terminal symbols of the language (keywords, special characters, etc.). Nonterminal symbols are enclosed between '<' and '>'. Vertical bar '|' separates alternatives. The square brackets '[', ']' enclose optional material which consists of one or more items separated by vertical bars.

< select statement >: **select** *< object variable list >*
 [**into** [**persistent** [**all**]] *< collection name >*]
 from *< range variable list >*
 [**where** *< boolean formula >*]

The *select clause* in this statement identifies objects to be returned in a new collection. There can be one or more object variables of different formats (constant, variables, path expressions or index variables) in this clause. They correspond to the free variables in an object calculus formula. The *into clause* declares a reference to a new collection that will hold the result. This collection can optionally be made persistent by specifying the **persistent** keyword. This does not make the members of the collection persistent. In order to do this, the **all** keyword must be specified as well. If the *into clause* is not specified, a new transient collection is created. There is no reference to this collection and it disappears at the end of a query. In this case, the result cannot be retained for later use by another query. It can only be printed to the screen, for example. The *from clause* declares the ranges of object variables in the *select* and *where* clauses. Every object variable can range over an existing collection or a collection returned as the result of a subquery. A subquery is a nested *select-from-where* clause that can be given explicitly or specified as a reference to an existing query object. A range variable statement in the *from clause* is as follows:

< range variable > : *< identifier list >* **in** *< collection reference >* [+]
< collection reference >: *< term >* | (*< query statement >*)

The collection reference in the range variable definition can be followed by a plus '+'. The plus refers to the shallow extent of a class, which is a collection of objects. The default is the deep extent for classes. The *term* in the collection reference definition is either a constant reference, or a variable reference, or a path expression.

The *where clause* defines a boolean formula that must be satisfied by the objects returned by a query. Boolean formulas have the following syntax:

< boolean formula >: *< atom >*
 | **not***< boolean formula >*
 | *< boolean formula >* **and** *< boolean formula >*
 | *< boolean formula >* **or** *< boolean formula >*
 | (*< boolean formula >*)
 | *< exists predicate >*
 | *< forAll predicate >*
 | *< boolean path expression >*

where an atom is defined as follows:

< atom >: *< term >* = *< term >* | *< identifier >* ∈ *< term >*

and a *term* is either a variable reference, a constant reference, or a path expression.

Two special predicates are added to TQL boolean formulas to represent existential and universal quantification. The existential quantifier is expressed by the *exists predicate* of the form:

< exists predicate >: **exists** *< collection reference >*

The *exists predicate* is *true* if the referenced collection is not empty. The universal quantifier is expressed by the *forAll predicate*, which has the following structure:

< forAll predicate >: **forAll** *< range variable list >* *< boolean formula >*

The syntax of the *range variable list* is the same as in the *from clause* of the *select* statement. It defines variables that range over specified collections. The *boolean formula* is evaluated for every possible binding

of the variables in this list. Thus, the entire *forall predicate* is *true*, if for every element in every collection in the range variable list, the boolean formula is satisfied.

The last component of the boolean formula definition is the *boolean path expression* defined simply as:

$$\langle \textit{path expression} \rangle = \mathbf{TRUE/FALSE}$$

To avoid such an artificial construct, we include a boolean path expression in the definition of a TQL formula under two conditions. First, all invoked functions are *side-effect-free* and second, the result type of the entire path expression must be a boolean type.

The following queries on the GIS example objectbase illustrate the expressive constructs of TQL and how the persistence of results are specified.

Example 5.1 Return the zones that are part of some map and are within 10 units from water. Project the result over *B_title* and *B_area*. Place the result into a persistent collection called **L_floodZones** and make all members persistent.

```
select o[B_title, B_area]
into persistent all L_floodZones
from p in C_map, o in p.B_zones(), q in C_water
where o.B_proximity(q) < 10
```

Example 5.2 Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map. The result is a transient collection that disappears at the completion of the query.

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

The following is an example of a reflective query and illustrates that no new constructs are needed in the language to query the schema.

Example 5.3 Return the types that define the behavior *B_age* with the same implementation as one of its supertypes. Place the result into a persistent collection called **L_inheritedAgeTypes**, but do not make the members persistent.

```
select t
into persistent L_inheritedAgeTypes
from t in C_type, r in t.B_supertypes()
where B_age ∈ t.B_interface() and B_age ∈ r.B_interface()
and B_age.B_implementation(t) = B_age.B_implementation(r)
```

TQL also supports three binary operations: **union**, **minus**, and **intersect**. The syntax of these statements is specified below. The *jcollection referencej* field can be specified as a subquery or as a reference to an existing (transient or persistent) collection.

$$\begin{array}{lll} \langle \textit{collection reference} \rangle & \mathbf{union} & \langle \textit{collection reference} \rangle \\ \langle \textit{collection reference} \rangle & \mathbf{minus} & \langle \textit{collection reference} \rangle \\ \langle \textit{collection reference} \rangle & \mathbf{intersect} & \langle \textit{collection reference} \rangle \end{array}$$

TQL has a proven equivalence to the formal languages making it easy to perform logical transformations and argue about its safety. The theorems and proofs of equivalence are given in [Lip93].

The TIGUKAT control language (TCL) defines statements for controlling operations within an objectbase session. In the absence of a computationally complete programming language, TCL serves to provide a scope for execution and interaction with a TIGUKAT objectbase.

Since everything in TIGUKAT is treated as a first-class object, sessions are also represented by objects. Specifically, session objects are instances of `T_session` type which is a direct subtype of `T_object`. Every TIGUKAT user has at least one instance of `T_session` which is referred to as a *root session*. Other sessions can be opened and manipulated from this session by issuing session-specific TCL operations: **open**, **close**, **save** (commit), and **quit** (abort). TCL also provides an assignment statement for creating object references as well as two forms of a *persistent* operation whose syntax is as follows:

1. **persistent** *<object reference list>*
2. **persistent all** *<collection reference>*

The semantics of the first form is to make every object in the given object reference list persistent according to the rules defined in Section 4. The second form requires the argument to be a collection. The semantics is to make the collection persistent and all of its members persistent as well.

The inverse operations of the **persistent** statements are the **transient** statements whose syntax is as follows:

1. **transient** *<object reference list>*
2. **transient all** *<collection reference>*

6 Other DBMS Functionalities

In addition to the powerful object and query models that TIGUKAT provides, the system is enhanced by a number of other functions commonly associated with DBMSs. In this section, we provide a brief overview of three functions that have been under development: query optimizer, the versioning scheme, and the transaction manager.

6.1 Query Optimizer

The goal of query optimization is the choice of the “optimum” execution plan for a query from a set of equivalent execution plans specified as algebraic expressions. The set of equivalent execution plans are obtained by the application of algebraic transformation rules and the optimum strategy is the one with the lowest cost according to a cost function. Thus, in order to characterize a query optimizer, three things need to be specified:

1. The transformation rules that generate the alternative query expressions which constitute the *search space*;
2. a *search algorithm* that allows one to move from one state (i.e., execution plan) to another in the search space; and
3. the *cost function* that is applied to each state.

The TIGUKAT query optimizer [Mun93] follows the philosophy of representing system concepts as objects and is along the lines of [LV91]. The search space, the search strategy and the cost function are modeled as objects (see Figure 11). The incorporation of these components of the optimizer into the type system provide extensibility via the basic object-oriented principle of subtyping and specialization.

The states in the search space are modeled as processing trees (PT) whose leaf nodes are references to collections and non-leaf nodes denote behavior applications whose results are other objects. Those nodes which correspond to algebraic operator behaviors return temporary collections as result.

Algebraic operators (e.g., *B_Select*, *B_Join*) are defined as behaviors of the `T_collection` type. They are modeled as instances (shown as dashed boxes in Figure 11) of type `T_algebra` which is a subtype of

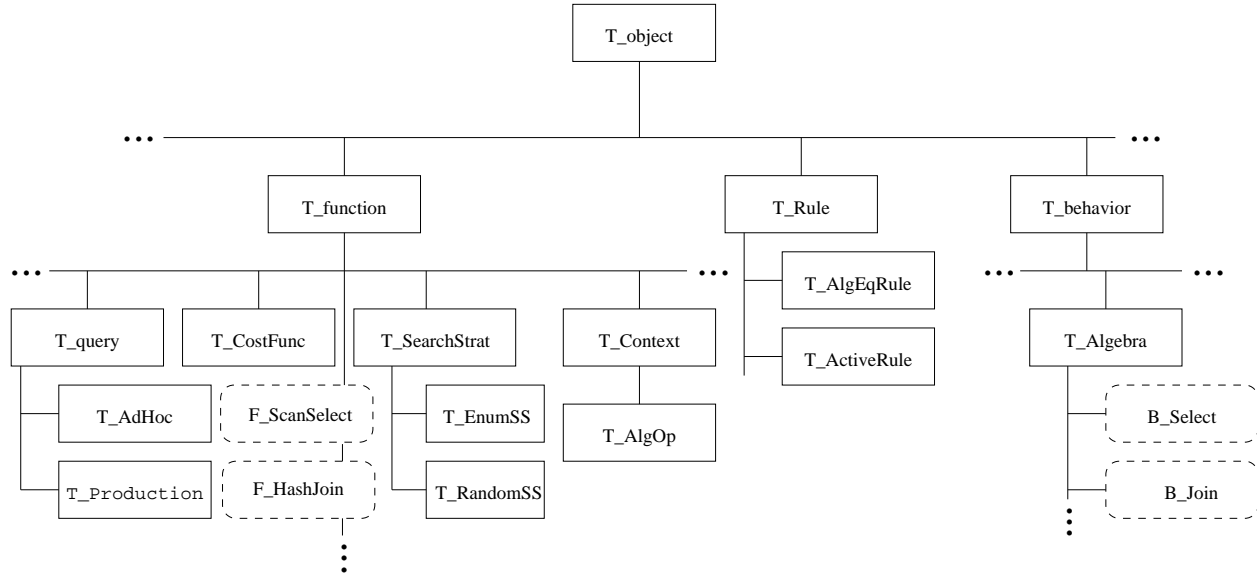


Figure 11: Optimizer as part of the type system

type **T_behavior**. The implementation (execution) algorithms for these algebraic operators are modeled as function objects (e.g., *F_HashJoin*, *F_ScanSelect*). These implementation functions cannot be used as nodes of a PT, since these nodes should represent execution functions all of whose arguments have been marshalled. Therefore, **T_AlgoOp** type is defined whose instances are functions with marshalled arguments and represent nodes of a PT. In this fashion, each node of a PT represents a specific execution algorithm for an algebra expression.

Search strategies are similarly modeled as objects, but separate from the search space. **T_SearchStrat** is defined as a subtype of type **T_function**, and it can in turn be specialized. Figure 11 shows the specialization of **T_SearchStrat** into enumerated search strategies **T_EnumSS** and randomized search strategies **T_RandomSS**. The algebraic transformation rules that control the movement of the search strategy through the search space are implemented as instances of **T_AlgoEqRule** which is a subtype of **T_Rule**.

Cost functions (instances of **T_CostFunc**) are defined as special types of functions, making them first-class objects. Each function is associated a cost through the behavior *B_costFunction*. Application of this behavior to a function object *f* (i.e., *f.B_costFunction*) returns another function object *g* of type **T_CostFunc** that implements the computation of the cost of executing function *f*. This allows definition of parameterized cost functions whose values are dependent upon a number of factors.

Modeling the building blocks of a cost-based optimizer as objects provides the query optimizer with the extensibility inherent in object models. The optimizer basically implements a control strategy that associates a search strategy and a cost function to each query.

6.2 Versioning

Traditionally, a *version* of a particular modeled entity (e.g., object, type, schema, objectbase, etc.) is perceived as a state of that entity as it existed at a particular time during its evolution. *Version control* is the ability to effectively and selectively manage versions of entities. For example, engineering design applications may track versions of components that have been put into production, stock market and taxation analysis applications may use versions of a futures model to evaluate “what if” scenarios and provide alternate futures scenarios, collaborative systems may have different design teams working on different versions of an overall design, and a system may even version the schema as it evolves so that old and new objects can coexist in the system without having to perform conversions on the instances of the schema [SZ86]. Some work [Sci94] have

separated user-level versions from system-level versions and then limited the version model to encompass user-level versions only. With uniform object models such as TIGUKAT, both user-level and system-level versions can seamlessly coexist, and a single version model suffices to support both. The version model developed for TIGUKAT [PGÖ] uniformly supports both user-level and system-level versions.

Temporal behaviors and branching time (i.e., branching behavior histories) are the framework for version support in TIGUKAT. A behavior can be temporal or non-temporal. The non-temporal behaviors maintain the most recent (i.e., snapshot) result while the temporal behaviors maintain a history of results as the behavior changed over time. This history may be represented by a linear time-model or a branching time-model. We propose to use the latter where each branch represents an alternate future (or version) of the behavior history. The unique aspects and advantages of our approach are the following:

1. The model is general in that it can be applied to any history tracking system that incorporates branching time. For example, it can be used on both valid time and transaction time as long as (i) they are modeled as histories and (ii) branching time is supported. Other systems that support valid and transaction time histories include [DW92, RS91], however, branching time is not directly supported in these systems (branching time is supportable in the model discussed in [DW92], but the burden of developing a branching model is left up to the user).
2. A portion of a behavior history (called a *version slice*) can be defined by specifying a start time and an end time on the history timeline. A version slice denotes the initial history of a temporal behavior for a given version and only that portion of the original behavior history is visible in the version. This is useful for excluding parts of the behavior history from the version. Version slicing is unique in that other temporal versioning models define a version based on the entire behavior history up to a certain end time.
3. Each version slice can spawn an independent branch on the timeline after the end of a slice. This is useful since it allows the behavior to temporally evolve along this branch independent of any other versions. We are unaware of any other model that allows version slices and versions to temporally evolve independent of other versions in this manner.
4. A version slice can *mirror* or *copy* the portion of the history on which it is defined. A *mirrored slice* reflects all changes to the slice in both the original and the version (i.e., updates to the version or the original within the slice are visible to both). A *copied slice* is a separate independent copy of the original behavior history that becomes part of the new version (i.e., the original and the version have their own copy of the slice and updates to the version or the original within the slice are not visible in the other).
5. The version model is general and when incorporated into a uniform object model like TIGUKAT, system-level versions such as versions of schema and versions of the entire objectbase can be modeled in addition to user-level versions without the need for extensions. This unifies user-level versions and system-level versions within a single framework.

We have completed the design of the branching time version model, defined a uniform behavioral representation of this model within TIGUKAT, and developed user language support for managing versions. The versioning approach has been mapped to other approaches such as versions of types, versions of schema, and versions of the entire objectbase, which are useful for schema evolution. This signifies the uniform feature of the version model as an underlying framework to support all types of versioning approaches. We are currently undertaking the implementation of the version model.

6.3 Transactions

Conventional transaction management involves the synchronization of simple read/write access to a shared database in an environment that is not failure-free. Both the transaction models and the synchronization

principles that are used in these environments are simple compared to those that are needed in OBMSs. The complexity of the application domains that the OBMS technology is expected to serve is reflected in the type of transaction management support that they require. In these systems there is a recognized need for more general and powerful transaction models [Elm92]. An overview of transaction management concerns in OBMSs is given in [Özs94].

One important characteristic of the relational data model – which is the basis of most current commercial systems – is its lack of a clear update semantics. The model, as it was originally defined, clearly spells out how the data in a relational database is to be retrieved (by means of the relational algebra operators), but does not specify what it really means to update the database. The consequence is that the consistency definitions and the transaction management techniques are orthogonal to the data model. It is possible – and indeed it is common – to apply the same techniques to non-relational DBMSs or even to non-DBMS storage systems.

The independence of the developed techniques from the data model may be considered an advantage since the effort can be amortized over a number of different applications. Indeed, the existing transaction management work on OBMSs have exploited this independence by porting the well-known techniques over to the new system structures. During this porting process the peculiarities of OBMSs such class (type) lattice structures, composite objects and object groupings (class extents), but the techniques are essentially the same.

In TIGUKAT, we are taking a different approach. It is our claim that in OBMSs, it is not only desirable to model update semantics within the object model, but that it is indeed essential for the correct operation of these systems. The arguments are as follows:

1. In OBMSs, what is stored are not only data but operations on data (which are called methods, behaviors, operations in various object models). Queries that access an object-oriented database refer to these operations as part of their predicates. In other words, the execution of these queries invoke various operations defined on the classes (types). To guarantee the safety of the query expressions, existing query processing approaches restrict these operations to be side-effect free, in effect disallowing them to update the database. This is a severe restriction that should be relaxed by the incorporation of update semantics into the query safety definitions.
2. Transactions in OBMSs effect the type (class) lattices. Thus, there is a direct relationship between dynamic schema evolution and transaction management. Many of the conventional techniques employ locking on this lattice to accommodate these changes. However, locks (even multi-granularity locks) severely restrict concurrency. Definition of what it means to update an objectbase and the definition of conflicts based-on this definition of update semantics would allow more concurrency.

It is interesting to note again the relationship between changes to the type (class) lattice and query processing. In the absence of a clear definition of update semantics and its incorporation into the query processing methodology, most of the current query processors assume that the database schema (i.e., the type (class) lattice) is static during the execution of a query [SÖ90a].

3. Since TIGUKAT treats all system entities, including the database schema (i.e., meta-objects) and queries, as objects that can themselves be queried, it is only natural to model transactions as objects. However, since transactions are basically constructs that change the state of the database, their effects on the database need to be clearly specified.

Within this context, it should also be noted that the application domains that require the services of OBMSs tend to have somewhat different transaction management requirements both in terms of transaction models and in terms of consistency constraints. Modeling transactions as objects enables the application of the well-known object-oriented techniques of specialization and subtyping to create various different types of transaction managers. This gives the system extensibility.

4. Some of the requirements require rule support and active database capabilities. Rules themselves execute as transactions which may spawn other transactions. It has been argued that rules should be

modeled as objects [DBM88]. If that is the case, then certainly transactions should be modeled as objects too.

Consequently, we are now working to define the update semantics of the TIGUKAT object model and are investigating a powerful transaction model (which may better be called a *workflow* following more current terminology) that meets the requirements of the application domains that OBMSs are likely to serve and is modeled in the system as objects. The concurrency control algorithms that are appropriate for these models exploit the semantics of operations and provide flexibility to the type implementors in defining the concurrent execution semantics. Our work in this area is relatively recent and more concrete results will be reported in future papers.

7 Conclusions and Future Directions

In this paper, we provide an overview of the TIGUKAT objectbase management system under development at the Laboratory for Database Systems Research at the University of Alberta. TIGUKAT has a uniform behavioral object model where everything is a first-class object and the only means of accessing the objectbase is through behavior application.

We have defined a query model for the system complete with an object calculus, an object algebra and a user language. The user-language consists of a definition language, a session language and an SQL-based query language. The interpreters for the first two and the compiler for the last one have been implemented. An extensible query optimizer has been defined and a type system to support this architecture has been implemented. The optimizer is being developed as a uniform extension to the object model and will therefore be integrated with the model just like the query model has been.

Current work on the system is progressing along five lines: (1) the incorporation of time into the object and query models, (2) the definition of the update semantics for the model, (3) the development of a view manager, (4) the development of storage structures to support query optimization (i.e., indexing and clustering issues), and (5) the definition of a transaction model and its incorporation into the model.

Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada under reserach grants OGP0951 and OGP8191 as well as by the Canadian Institute for Telecommunications Research (CITR), a federally funded Centre of Excellence.

The authors thank the anonymous referees as well as Malcolm Atkinson for numerous suggestions that improved the paper significantly.

Technical reports and papers related to TIGUKAT are available via Web at <http://web.cs.ualberta.ca/ozsu/tigukat.html>.

References

- [AB87] M. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Computer Surveys*, 19(2):105–190, June 1987.
- [ABC⁺83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.
- [ABD⁺89] M. Atkinson, F. Bancilhon, D.J. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, pages 40–57, 1989.
- [AR92] P. André and J. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *OOPSLA '92 Conf. Proc.*, pages 110–123, 1992.

- [BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System, The Story of O₂*. Morgan Kaufmann, 1992.
- [Bla91] J.A. Blakeley. DARPA open object-oriented database preliminary modeule specification: Object query module. Technical report, Texas Instruments, December 1991.
- [BOS91] P. Butterworth, A. Otis, and J. Stein. The Gemstone object database management system. *Comm. of the ACM*, 34(10):64–77, October 1991.
- [Car84] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer Verlag, 1984.
- [Car86] L. Cardelli. A polymorphic λ -calculus with Type:Type. Research Report 10, DEC Systems Research Center, May 1986.
- [Cat91] R. G. Cattell. *Object Data Management*. Addison Wesley, 1991.
- [Coi87] P. Cointe. Metaclasses are first class: The ObjVlisp model. In *OOPSLA '87 Conf. Proc.*, pages 156–167, October 1987.
- [Day89] U. Dayal. Queries and views in an object-oriented data model. In *Proc. 2nd Int. Workshop on Database Programming Languages*, pages 80–102. Morgan Kaufmann, 1989.
- [DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active object-oriented database system. In *Proc. of the 2nd Int. Workshop on Object-Oriented Database Systems*, pages 129–143, 1988.
- [Deu91] Deux, O. et al. The O₂ system. *Comm. of the ACM*, 34(10):34–48, October 1991.
- [DMSV89] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *OOPSLA '89 Conf. Proc.*, pages 211–214, 1989.
- [DW92] U. Dayal and G. Wu. A Uniform Approach to Processing Temporal Queries. In *Proc. 8th Int. Conf. on Data Engineering*, pages 407–418, August 1992.
- [Elm92] A.K. Elmagarmid (ed.). *Transaction Models for Advanced Database Applications*. Morgan Kaufmann, 1992.
- [FKMT91] E. Fong, W. Kent, K. Moore, and C. Thompson. X3/SPARC/DBSSG/OODBTG Final Report. Technical report, NIST, September 1991.
- [Gal92] L.J. Gallagher. Object SQL: Language Extensions for Object Data Management. In *Proc. 1st International Conference on Information and Knowledge Management*, pages 17–26, November 1992.
- [GM93] I. Goralwalla and M.T.Özsu. Temporal extensions to a uniform behavioral object model. In *Proc. 12th Int. Conf. on Entity-Relationship Approach*, pages 115–127, December 1993.
- [GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, Reading, Mass., 1983.
- [GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 2 edition, 1989.
- [GT91] A.V. Gelder and R.W. Topor. Safety and translation of relational calculus queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.

- [Ira93] B.B. Irani. Implementation of the TIGUKAT object model. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report, TR93-10.
- [KBC⁺89] W. Kim, N. Ballou, H.T. Chou, J.F. Garza, and D. Woelk. Features of the ORION Object-Oriented Database System. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, 1989.
- [KC86] S. N. Khoshafian and G. P. Copeland. Object identity. In *OOPSLA '86 Conf. Proc.*, pages 406-416, September 1986.
- [Ken90] W. Kent. A Rigorous Model of Object Reference, Identity and Existence. Technical Report HPL-90-31, Hewlett Packard Labs, April 1990.
- [Lip93] A.P. Lipka. The design and implementation of TIGUKAT user languages. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report TR93-11.
- [LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Comm. of the ACM*, 34(10):50-63, October 1991.
- [LRV88] C. Lecluse, P. Richard, and F. Velez. O₂, an Object-Oriented Data Model. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 424-433, September 1988.
- [LV91] R. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. 17th Int. Conf. on Very Large Databases*, pages 363-373, 1991.
- [Mak81] J.A. Makowsky. Characterizing Data Base Dependencies. In *Proc. 8th Colloquium on Automata, Languages and Programming*. Springer Verlag, 1981.
- [Mun93] A. Munoz. Extensible query optimizer architecture for TIGUKAT. Master's thesis, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report TR94-01.
- [MZO89] D. Maier, J. Zhu, and H. Ohkawa. Features of the TEDM object model. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, pages 476-495, 1989.
- [Os88] S. L. Osborn. Identity, equality and query optimization. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 346-351. Springer Verlag, 1988.
- [ÖSP94] M.T. Özsu, D.D. Straube, and R. Peters. Query processing issues in object-oriented knowledge base systems. In F.E. Petry and L.M. Delcambre, editors, *Emerging Landscape of Intelligence in Database and Information Systems*. JAI Press, 1994. In press.
- [Özs94] M.T. Özsu. Transaction models and transaction management in object-oriented database management systems. In A. Doğaç, M.T. Özsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*. Springer Verlag, 1994.
- [Pet94] R.J. Peters. *TIGUKAT: A uniform behavioral objectbase management system*. PhD thesis, University of Alberta, Edmonton, Alberta, Canada, 1994. Available as University of Alberta Technical Report TR94-06.
- [PGÖ] R.J. Peters, I. Goralwalla, and M.T. Özsu. A Unified Version Model Based on Branching Time. Submitted to *3rd Int'l Conf. on Information and Knowledge Management*.

- [PLÖS93a] 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. 2nd Int. Conf. on Information and Knowledge Management*, pages 403–412, November 1993.
- [PLÖS93b] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. The query model and query language of TIGUKAT. Technical Report TR93-01, Department of Computing Science, University of Alberta, January 1993.
- [PÖ93] R.J. Peters and M.T. Özsu. Reflection in a Uniform Behavioral Object Model. In *Proc. 12th Int. Conf. on Entity-Relationship Approach*, pages 37–49, December 1993.
- [PÖLS94] R.J. Peters, M.T. Özsu, A. Lipka, and D. Szafron. The query model and query language of tigukat. *In preparation*, 1994.
- [RC89] J. Richardson and M. Carey. Persistence in the e language: Issues and implementation. *Software — Practice & Experience*, 19(12):1115–1150, December 1989.
- [RCS89] J. Richardson, M. Carey, and D. Schuh. The Design of the E Programming Language. Technical Report 824, University of Wisconsin, February 1989.
- [RS91] E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proc. 10th Int. Conf. on Entity-Relationship Approach*, pages 205–229, October 1991.
- [SA85] R. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 236–246, May 1985.
- [SCD90] D. T. Schuh, M. J. Carey, and D. J. DeWitt. Persistence in E Revisited - Implementation Experiences. In *Implementing Persistent Object Bases: Principles and Practice. Proc. Fourth Int. Workshop on Persistent Object Systems*, pages 345–359, 1990.
- [Sci94] E. Sciore. Versioning and Configuration Management in an Object-Oriented Data Model. *The VLDB Journal*, 3(1):77–106, January 1994.
- [Shi81] D. W. Shipman. The functional data model and the language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.
- [Sny90] A. Snyder. An Abstract Object Model for Object-Oriented Systems. Technical Report HPL-90-22, Hewlett Packard Labs, April 1990.
- [SÖ90a] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.
- [SÖ90b] D.D. Straube and M.T. Özsu. Type consistency of queries in an object-oriented database system. In *Proc. ECOOP/OOPSLA '90 Conference*, pages 224–233, 1990.
- [Str86] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.
- [SZ86] A.H. Skarra and S.B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *OOPSLA '86 Conf. Proc.*, pages 483–495, September 1986.
- [SZ90] G. Shaw and S. Zdonik. A query algebra for object-oriented databases. In *Proc. 6th Int. Conf. on Data Engineering*, pages 154–162, February 1990.
- [WBW88] A. Wirfs-Brock and B. Wilkerson. An Overview of Modular Smalltalk. In *OOPSLA '89 Conf. Proc.*, pages 123–134, September 1988.

- [WBW89a] A. Wirfs-Brock and B. Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. In *OOPSLA '89 Conf. Proc.*, pages 71–75, October 1989.
- [WBW89b] A. Wirfs-Brock and B. Wilkerson. Variables Limit Reusability. *Journal of Object-Oriented Programming*, 2(1):34–40, May/June 1989.
- [YO91] L. Yu and S.L. Osborn. An evaluation framework for algebraic object-oriented query models. In *Proc. 7th Int. Conf. on Data Engineering*, pages 670–677, 1991.
- [ZM90] S. Zdonik and D. Maier. Fundamentals of Object-Oriented Databases. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, pages 1–36. Morgan-Kaufman, 1990.

A Primitive Type System

The following tables show the signatures of the behaviors for the non-atomic types (except the container types), the signatures of the behaviors for the container types and the signatures of the behaviors for the atomic types. The receiver type of a behavior is excluded because the receiver must be an object of a type that is compatible with the type defining the behavior. The notation $\mathbf{T_collection}(T)$ is used to define a collection type whose members are of type T . The type specifications for the behaviors are the *most general* types. Types for some of the behaviors are revised in the subtypes. For example, the result type of B_self is always the type of the receiver object and the result type of B_new is always the membership type of the receiver class.

Type	Signatures
T_object	<i>B_self</i> : T_object <i>B_mapsto</i> : T_type <i>B_conformsTo</i> : T_type \rightarrow T_boolean <i>B_equal</i> : T_object \rightarrow T_boolean <i>B_notequal</i> : T_object \rightarrow T_boolean <i>B_persistent</i> : T_object <i>B_transient</i> : T_object <i>B_newprod</i> : T_list(T_object) \rightarrow T_list(T_set(T_behavior)) \rightarrow T_object
T_type	<i>B_interface</i> : T_set(T_behavior) <i>B_native</i> : T_set(T_behavior) <i>B_inherited</i> : T_set(T_behavior) <i>B_specialize</i> : T_type \rightarrow T_boolean <i>B_subtype</i> : T_type \rightarrow T_boolean <i>B_subtypes</i> : T_set(T_type) <i>B_supertypes</i> : T_set(T_type) <i>B_sub-lattice</i> : T_poset(T_type) <i>B_super-lattice</i> : T_poset(T_type) <i>B_classof</i> : T_class <i>B_tmeet</i> : T_set(T_type) \rightarrow T_type <i>B_tjoin</i> : T_set(T_type) \rightarrow T_type <i>B_tproduct</i> : T_list(T_type) \rightarrow T_type
T_product	<i>B_compTypes</i> : T_list(T_type)
T_behavior	<i>B_name</i> : T_string <i>B_argTypes</i> : T_list(T_type) <i>B_resultType</i> : T_type \rightarrow T_type <i>B_semantics</i> : T_object <i>B_associate</i> : T_type \rightarrow T_function \rightarrow T_behavior <i>B_implementation</i> : T_type \rightarrow T_function <i>B_primitiveApply</i> : T_object \rightarrow T_object <i>B_apply</i> : T_object \rightarrow T_list \rightarrow T_object <i>B_defines</i> : T_set(T_type)
T_function	<i>B_argTypes</i> : T_list(T_type) <i>B_resultType</i> : T_type <i>B_source</i> : T_object <i>B_primitiveExecute</i> : T_object \rightarrow T_object <i>B_basicExecute</i> : T_list \rightarrow T_object <i>B_execute</i> : T_list \rightarrow T_object <i>B_compile</i> : T_object <i>B_executable</i> : T_object

Table 2: Behavior signatures of the non-atomic types of the primitive type system.

Type	Signatures
T_collection	<i>B_memberType</i> : T_type <i>B_union</i> : T_collection → T_collection <i>B_diff</i> : T_collection → T_collection <i>B_intersect</i> : T_collection → T_collection <i>B_collapse</i> : T_collection <i>B_select</i> : T_string → T_list(T_collection) → T_collection <i>B_project</i> : T_set(T_behavior) → T_collection <i>B_map</i> : T_string → T_list(T_collection) → T_collection <i>B_product</i> : T_set(T_collection) → T_collection <i>B_join</i> : T_string → T_list(T_collection) → T_collection <i>B_genjoin</i> : T_string → T_list(T_collection) → T_collection <i>B_setEqual</i> : T_collection → T_boolean <i>B_containedBy</i> : T_collection → T_boolean <i>B_cardinality</i> : T_natural <i>B_elementOf</i> : T_object → T_boolean <i>B_insert</i> : T_object → T_collection <i>B_delete</i> : T_object → T_collection
T_bag	<i>B_occurrences</i> : T_object → T_natural <i>B_count</i> : T_natural Inherited behaviors refined to preserve duplicates
T_poset	<i>B_ordered</i> : T_object → T_object → T_boolean <i>B_ordering</i> : T_behavior Inherited behaviors refined to preserve ordering
T_list	<i>B_first</i> : T_object <i>B_last</i> : T_object <i>B_next</i> : T_object <i>B_previous</i> : T_object Inherited Behaviors refined to preserve duplicates and ordering
T_class	<i>B_deepExtent</i> : T_collection <i>B_new</i> : T_object
T_class-class	<i>B_new</i> : T_type → T_class
T_type-class	<i>B_new</i> : T_set(T_type) → T_set(T_behavior) → T_type
T_collection-class	<i>B_new</i> : T_type → T_collection
T_product-class	<i>B_new</i> : T_list(T_object) → T_object

Table 3: Behavior signatures of the container types of the primitive type system.

Type	Signatures
<code>T_atomic</code>	
<code>T_boolean</code>	$B_not: T_boolean$ $B_or: T_boolean \rightarrow T_boolean$ $B_if: T_object \rightarrow T_object \rightarrow T_object$ $B_and: T_boolean \rightarrow T_boolean$ $B_xor: T_boolean \rightarrow T_boolean$
<code>T_character</code>	$B_ord: T_natural$
<code>T_string</code>	$B_car: T_character$ $B_cdr: T_string$ $B_concat: T_string \rightarrow T_string$
<code>T_real</code>	$B_succ: T_real$ $B_pred: T_real$ $B_add: T_real \rightarrow T_real$ $B_subtract: T_real \rightarrow T_real$ $B_multiply: T_real \rightarrow T_real$ $B_divide: T_real \rightarrow T_real$ $B_trunc: T_integer$ $B_round: T_integer$ $B_lessThan: T_real \rightarrow T_boolean$ $B_lessThanEQ: T_real \rightarrow T_boolean$ $B_greaterThan: T_real \rightarrow T_boolean$ $B_greaterThanEQ: T_real \rightarrow T_boolean$
<code>T_integer</code>	Behaviors from <code>T_real</code> refined to work on integers
<code>T_naturals</code>	Behaviors from <code>T_integer</code> refined to work on naturals

Table 4: Behavior signatures of the atomic types of the primitive type system.