An Extensible Query Model and Its Languages for a Uniform Behavioral Object Management System

Randal J. Peters, Anna Lipka, M. Tamer Özsu and Duane Szafron
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

In this paper, we present an extensible, uniform, behavioral query model and its languages for the TIGUKAT object management system [PÖS92]. The TIGUKAT model is purely *behavioral* in nature, supports full encapsulation of objects, defines a clear separation between primitive components such as *types, classes, collections, behaviors, functions, etc.*, and incorporates a *uniform* semantics over objects which makes it a favorable basis for a query model. Queries are modeled as type and behavior extensions to the base object model, thus incorporating queries as an extensible part of the model itself. We present the framework of the complete query model definition that includes the extended types and behaviors, a formal object calculus with safety based on the evaluable class of queries, an equivalent object algebra, an SQL-like *ad hoc* query language for user-level querying and proof of its completeness.

Keywords: object-oriented systems, object query model, object calculus, object algebra, query language, object model

1 Introduction

To meet data management requirements of new complex applications, object management systems[1] are emerging as the most likely candidates. The general acceptance of this new technology depends on the increased functionality it can provide, and one measurement is the power of its query model. Users of these systems must have a declarative language to formulate queries by focusing on "what" information is required, and leaving it to the system to determine "how" to efficiently retrieve the information. Therefore, the formal query model of these systems should define a declarative calculus that can be used to formulate queries to the objectbase and an equivalent procedural or functional algebra to execute them efficiently.

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

Several research efforts have addressed the problem of formalizing a query model in an object-oriented environment. These have led to the development of design frameworks for object algebras [YO91], object calculi [AB93], and query languages [Bla91, ÖSP94]. Furthermore, some work is ongoing to extend SQL with object-oriented features [Gal92]. Straube and Özsu [SÖ90] have investigated query processing issues in the domain of object-oriented databases. They specify both an object algebra and calculus definition, and the two are linked with a calculus to algebra translation. However, the reduction is only partial and the algebra lacks object creating operators. The algebra of Shaw and Zdonik [SZ90] consistently extends the relational algebra with both preserving and creating operators forming a complex object algebra. Osborn [Osb88] defines an algebra for an object-oriented model based on atomic objects, strongly typed aggregates (tuples) and both homogeneous and heterogeneous sets. The algebra provides a full range of algebraic operations including the *combine* operator which is equivalent to cartesian product. However, the integration of the result of a combine with the existing lattice is not specified. Kim [Kim89] defines an algebra that includes object-creating operators for performing joins on objects, but the integration of these new objects into the existing lattice is not known and so they hang off the root of the lattice.

In this paper, we report our work on a complete object query model and its languages. The query model is developed in a uniform, extensible fashion (using object-oriented techniques) as type and behavior extensions to the base object model. Thus, queries are uniformly modeled as objects and are accessible through behaviors like any other object. This work is conducted within the framework of the TIGUKAT[2] project. TIGUKAT is an extensible uniform, behavioral, object management system. Its object model [PÖS92] is characterized by an abstract behavioral definition and a uniform approach to objects. Everything, including types, classes, behaviors, functions, meta-information and so on, is modeled as a first class object.

The identifying characteristics of the TIGUKAT query model which differentiates it from other proposals are as follows:

1. It incorporates a formal object calculus and object algebra specification with a proven equivalence (see [PLÖS93] for theorems and proofs).

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

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

2. Its safety criterion is based on the *evaluable* class of queries [GT91] which is arguably the largest decidable subclass of domain independent queries [Mak81].

3. It exploits object-oriented features to extend the *evaluable* class by introducing notions of *object generation* which relaxes *range* specification requirements.

4. It incorporates a complete SQL-like user language called TQL (TIGUKAT Query Language), an object definition language called TDL (TIGUKAT Definition Language) and a control language called TCL (TIGUKAT Control Language) [Lip93]. TQL is proven equivalent to the formal languages making it easy to perform logical transformations and argue about its safety.

5. It uniformly models queries as first-class objects by defining type and behavior extensions to the object model. This makes for an extensible query model with a consistent uniform underlying semantics.

Although our work is within the context of the TIGUKAT project, the results reported here extend to any system based on a uniform behavioral object model where behaviors are implemented as functions.

The remainder of the paper is organized as follows. In Section 2, we give a brief overview of the TIGUKAT object model. This outlines the fundamental features of the model and gives a short specification of the primitive type lattice. In Section 3, an overview of the TIGUKAT query model as an extension to the object model is presented and the concept of queries as objects is described. In Section 4, we present the formal object calculus and discuss its safety. In Section 5, the syntax and semantics of the TIGUKAT Query Language are given. In Section 6, we present the operators of the formal object algebra. In Section 7, we define a Geographic Information System (GIS) as an example objectbase and present several example queries expressed in their equivalent TQL, object calculus and object algebra forms. The reader may want to refer to the examples in this section while reading the earlier parts of the paper. Finally, Section 8 contains concluding remarks and a brief discussion of the ongoing work.

2 Object Model Overview

The TIGUKAT object model [PÖS92] is defined *behaviorally* with a *uniform* object semantics. The model is *behavioral* in the sense that all access and manipulation of objects is based on the application of behaviors to objects, and the model is *uniform* in that every component of information, including its semantics, is uniformly modeled by objects and has the status of a *first-class object*. Thus, an *object* is a fundamental concept in TIGUKAT, meaning every expressible element incorporates at least the semantics of the primitive notion for "object."

The primitive objects of the model include: *atomic entities* (reals, integers, strings, etc.); *types* for defining common features of objects; *behaviors* for specifying the semantics of operations that may be performed on objects; *functions* for specifying implementations of behaviors over types[3]; *classes* for automatic classification of objects based on type[4]; and *collections* for supporting general heterogeneous groupings of objects. In the remainder of the paper, the prefix `T_` refers to a type, **C_** refers to a class, **L_** refers to a collection, and *B_* refers to a behavior. For example, `T_person` is a type reference, **C_person** a class reference, **L_seniors** a collection reference, *B_age* a behavior reference, and a reference such as **David** without any prefix represents some other application specific reference. Some primitive types and behaviors are elaborated on in this paper. For the complete model definition see [PÖS92].

Objects are defined as *(identity, state)* pairs where *identity* represents a unique, immutable object identity and *state* represents the information carried by the object. Thus, the model supports *strong object identity* [KC86]. This does not preclude application environments such as object programming languages from having many *references* (or *denotations*) to objects which need not be unique. The *state* of an object *encapsulates* the information carried by that object. More specifically, the state encapsulates the *denotations* of objects and hides the structure and implementation of the information carried by that object. Conceptually, every object is a *composite* object, meaning every object has references (not necessarily implemented as pointers) to other objects. For example, integers have behaviors which return objects, but obviously they are not implemented as a series of pointers. This illustrates a strong point of the model in the separation of behaviors from their implementations.

The access and manipulation of an object's state occurs exclusively through the application of behaviors. An important primitive behavior defined on objects is *identity equality* that compares two object references based solely on the identities of the objects they denote. This is the only kind of equality defined in the primitive type system.

The model separates the definition of object characteristics (a *type*) from the mechanism for maintaining instances of a particular type (a *class*). A *type* specifies behaviors and encapsulates behavior implementations and state for objects created using that type as a template. The behaviors defined by a type describe the *interface* to the objects of that type. Types are organized into a lattice structure using the notion of *subtyping* which promotes software reuse and incremental development. Since TIGUKAT supports *multiple subtyping*, the type structure is potentially a directed acyclic graph (DAG). However, this DAG is converted to a lattice by *lifting* with the base type `T_null`.

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

Objects of a particular type cannot exist without an associated class and every class is uniquely associated with a single type. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes* which imply *types*. Another unique feature of classes is that object creation occurs only through a class. 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.

We define a *collection* as a general user-definable grouping construct. A *collection* is similar to a *class* in that

[3] Behaviors and functions form the support mechanism for *overloading* and *late binding* of behaviors.

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

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

it groups objects, but it differs in the following respects. First, no object creation may occur through a collection; object creation occurs only through classes. Second, an object may exist in any number of collections, but is a member of the shallow extent of only one class. Third, the management of classes is *implicit* in that the system automatically maintains classes based on the subtype lattice whereas the management of collections is *explicit*, meaning the user is responsible for their extents. Finally, the elements of a class are homogeneous up to inclusion polymorphism while a collection may be heterogeneous in the sense that it can contain objects which may be of different types. There is no equivalent of shallow extent for collections.

We define *class* as a subtype of *collection*. This introduces a clean semantics between the two and allows the model to utilize both constructs in an effective manner. For example, the targets and results of queries are typed collections of objects. This means targets also include classes because of the specialization of classes on collections. This approach provides great flexibility and expressiveness in formulating queries and gives *closure* to the query model which is often regarded as an important feature [YO91].

Two other fundamental notions are *behaviors* and the *functions* (known as *methods* in other models) that implement them. We clearly separate the definition of a behavior from its possible implementations (functions/methods). The benefit of this approach is that common behaviors over different types can have a different implementation for each of the types. This is in direct support for behavior *overloading* and *late binding* of implementations to behaviors. These are recognized as major advantages of object-oriented computing.

The semantics of every operation on an object is specified by a behavior defined on its type. A function implements the semantics of a behavior. The implementation of a particular behavior may vary over the types which support it. However, the semantics of the behavior remain consistent over all types supporting that behavior. There are two kinds of implementations for behaviors. A *computed function* consists of runtime calls to executable code and a *stored function* is a reference to an existing object in the objectbase. The uniformity of TIGUKAT considers each behavior application as the invocation of a function, regardless of whether the function is stored or computed. Functions are examined more closely in Section 3. We show that queries are specialized functions and therefore carry all the semantics of function objects, meaning they can be used as implementations of behaviors.

3 Query Model Overview

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 as illustrated in Figure 1. 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.

Functions have source code associated with them and the source code for a query is a TQL statement as defined in Section 5. Functions have a behavior *B_compile* which compiles the code. For a query, this involves translating the query statement into an algebra tree, optimizing it and

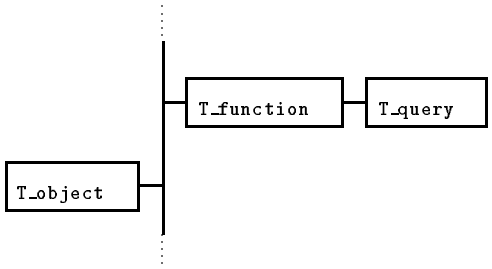

Figure 1: Query type extension to primitive type system.

generating an execution plan. Functions have a behavior *B_execute* which executes the compiled code. In general, for a query this means submitting the execution plan to the storage manager for processing. Furthermore, queries have specialized behaviors such as *B_result* which is a reference to the materialized query result (i.e., the actual result collection itself). If this result is made persistent, then the query is said to be *stored* and in some cases need not be re-evaluated the next time it is called upon to *B_execute* itself. Other behaviors relating to the extensible query optimizer include *B_initialPT* and *B_optimizedPT* for accessing the initial and optimized processing trees; *B_search-strategy* for accessing the search strategy used for optimization; *B_transformations* for accessing the list of transformation rules used during optimization; *B_input-types* for accessing the types of the operand collections; *B_output-type* for accessing the type of the result collection; and several other behaviors for keeping various statistics about queries.

Incorporating queries as a specialization of functions is a very 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 upon 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 is extensible in a uniform way since the type `T_query` can be further specialized by subtyping. This can be used to dichotomize the class of queries into additional subclasses, each with its own unique characteristics, and to incrementally develop the characteristics of new kinds of queries as they are discovered. For example, we can subtype `T_query` into `T_adhocQuery` and `T_productionQuery` and then define different evaluation strategies for both. *Ad hoc* queries may be interpreted without incurring high compile-time optimization strategies while production queries are usually compiled once and then executed many times.

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

The languages for the query model include a complete object calculus, and equivalent object algebra and an SQL-like user language.

The user query language (TQL) has syntax based on the SQL *select-from-where* structure, and formal semantics defined by the object calculus. Thus, it combines the power of the relational query languages with object-oriented features.

The calculus has a logical foundation and its expressive power is outlined by the following characteristics. It defines predicates on collections (essentially sets) of objects and returns collections of objects as results which gives the language *closure*. It incorporates the behavioral paradigm of the object model and allows the retrieval of objects using nested behavior applications, sometimes referred to as *path expressions* or *implicit joins*. It supports both *existential* and *universal* quantification over collections. It has rigorous definitions of safety (based on the evaluable class of queries) and typing which are compile time checkable. It supports controlled creation and integration of new collections, types and objects into the existing schema.

Like the calculus, the algebra is *closed* on collections. Algebraic operators are modeled as behaviors on the primitive type `T_collection`. They operate on collections and return a collection as a result. Thus, the algebra has a behavioral/functional basis as opposed to the logical foundation of the calculus. The composition over these behaviors brings closure to the algebra.

A desirable property of an object query model is that the algebra and calculus be equivalent in expressive power, meaning that all queries expressed in one language can also be expressed in the other. Space limitations do not allow us to include them here, but in [PLÖS93] we prove the equivalence of our object calculus and algebra in both directions and present the reduction of the user query language to the calculus. Moreover, the safety of our languages is proven in that report as well.

4 The Object Calculus

It is well recognized that a declarative query facility is an essential component of any database management system; object-oriented systems are no exception. In this section, we present a high-level object calculus with first-order semantics. In order to maintain the uniformity of the behavioral object model within the query model, the behavioral abstraction paradigm is carried through into the calculus. The logical foundation of the calculus includes a function symbol to incorporate the behavioral nature of the object model. This allows the use of very general path expressions in the calculus. The safety of our 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 *object generators* in queries which alleviates the need for explicit *range* expressions for each variable. For a complete discussion of safety in TIGUKAT see [PLÖS93].

The alphabet of the calculus consists of object constants (a, b, c, d), object variables (o, p, q, u, v, x, y, z), monadic predicates (C, P, Q), dyadic predicates $(=, \neq, \in, \notin)$, an n-ary predicate $(Eval)$, logical connectives $(\exists, \forall, \wedge, \vee, \neg)$, a function symbol (β) and delimiters (& ( ) ,). The object constants, object variables, monadic predicates and function symbol may be subscripted (e.g., a_3, o_i, C_n, β_1,etc.). In addition, we adopt a vector notation $\vec{s}$ to denote a countably infinite list of symbols $s_1, s_2, \ldots, s_n$ where $n \geq 0$.

From object constants and object variables we develop the syntax and semantics of the function symbol β called a *behavioral specification* (Bspec). A *term* is an object constant, an object variable or a Bspec. A Bspec is an n+2-ary function $\beta(s, b, \vec{t})$ where s and each t_i denote terms and where b is an object constant. For $n = 0$ we use $\beta(s, b)$ without loss of generality. A *ground term* is a term consisting of an object constant or any term composed from ground terms. From now on, any symbol defined as denoting an object constant, including symbols a, b, c, d, denote ground terms as well. Any term that is not a ground term is called a *variable term*.

The ordered list of terms $s, b, \vec{t}$ is considered to be *behaviorally consistent* if and only if b is an object constant denoting a behavior, the type of the object denoted by s defines behavior b as part of its interface, $\vec{t}$ is compatible with the arity of behavior b, and the types of the objects denoted by $\vec{t}$ are compatible with the argument types of behavior b. A Bspec $\beta(s, b, \vec{t})$ is *consistent* if and only if $s, b, \vec{t}$ is *behaviorally consistent*.

The "evaluation" of a consistent Bspec involves applying the behavior b to the object denoted by term s using objects denoted by terms $\vec{t}$ as arguments. The "result" of Bspec evaluation denotes an object in the objectbase. Since Bspecs denote objects, they have a type (and a class) that are in the objectbase as well.

Bspec evaluation has the following logical foundation. We introduce the n+3-ary predicate $Eval(R, s, b, \vec{t})$ as an axiom in the language such that $Eval(R, s, b, \vec{t})$ is true if and only if R denotes the "result" of applying behavior b to the object denoted by term s using terms $\vec{t}$ as arguments. The function symbol $\beta(s, b, \vec{t})$ is a logical representation of R. The $Eval$ predicate also serves as an enforcement of the consistency property of Bspecs. From now on we consider only those Bspecs that are consistent.

Bspecs may be composed. This provides the capability of building *path expressions* in queries. For example, given object constants **David**, *B_worksFor* and *B_budget* where **David** is an employee, *B_worksFor* is defined for employees and *B_budget* is defined for departments, we can compose the Bspec $\beta(\beta(\textbf{David}, B\_worksFor), B\_budget)$ which denotes the object representing the annual budget of the department that **David** works for.

For brevity, we recast the syntax of Bspecs into the dot notation as $s.b(\vec{t})$ which we intend as being semantically equivalent to the original specification. If behavior b does not require any arguments, then the notation simplifies to $s.b$. The previous example can then be represented as $\textbf{David}.B\_worksFor.B\_budget$ assuming left-associativity of behavior applications. Parenthesis may be used to change the order of precedence. Some other equivalent syntax, such as function application $b(s, \vec{t})$ which is popular in other languages, could have been chosen instead.

As shown by the above example, many path expression formations often include a sequence of behaviors with the semantics that the result of the first behavior be used as the input to the second and so on. We call such a sequence of **m**ultiple **op**erations a **mop** [SÖ90] which is a Bspec. We introduce the multi-operation dot notation $<\vec{s}>.b_1.b_2...b_m$ to denote a multi-operation resulting in the application of behavior object constants $b_1.b_2...b_m$ using objects denoted by terms $\vec{s}$ as arguments. Furthermore, $<\vec{s}>.\vec{b}$ is used as a shorthand to denote a multi-operation where the number and ordering of the behaviors are immaterial.

The *atoms* of the calculus consist of the following:

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

Range Atom: $C(o)$ is a *range atom* for o where C corresponds to a unary predicate representing a collection and o denotes an object variable. A range atom asserts true if and only if o denotes an object in collection C. When C defines a class, it denotes the deep extent of the class and we extend the notation to include $C^+(o)$ which asserts true if and only if o denotes an object in the *shallow extent* of the class. One may think of C^+ as a separate monadic predicate for specifying the shallow range of o.

Equality Atom: $s = t$ is a built-in predicate called an *equality atom* where s and t are terms. The predicate asserts that the object denoted by s is object identity equal to the object denoted by t. As a syntactical convenience, we simplify the equality atom specification $s = \mathbf{true}$ where s is boolean, to just s and $s = \mathbf{false}$ to $\neg s$. The built-in predicate $s \neq t$ is the complement of equality.

Membership Atom: $s \in t$ is a built-in predicate called a *membership atom* where s and t are terms and t is a term denoting a collection. The predicate asserts true if and only if the object denoted by s is an element of the collection denoted by t. The built-in predicate $s \notin t$ is the complement of membership.

Generating Atom: An equality atom of the form $o = t$ or a membership atom $o \in t$, where o is an object variable and t is an appropriate term for the atom in which o does not appear, are called *generating atoms* for o. They are so named because the object denotations for o can be *generated* from t.

A *ground atom* is an atom that contains only ground terms. A *literal* is either an atom or a negated atom. A *ground literal* is a literal whose atom is a ground atom.

The choice of atoms may seem restrictive when compared to other calculi such as the tuple relational calculus which allows a greater variety of comparison predicates including $=, <, \leq, >$, and $\geq$. An identifying characteristic of our calculus is that it is strictly behavioral and does not allow explicit value based comparisons of objects or its subcomponents. Thus, operations such as $<, >, \geq, \leq$ must be defined as behaviors on the respective types of objects that are to be compared. The only comparison predicate defined is that of object identity equality. However, type implementors can specialize this behavior for their types in order to define a form of equality (including value based comparisons) that is of most utility to them. For example, we define a form of "structural equality" on cartesian product types that compares two product objects based on the isomorphic mappings of their respective component objects.

From atoms, *first-order well-formed-formula* or simply *formula* (abbreviated WFF) are defined in the usual way using logical connectives $\exists, \forall, \wedge, \vee$ and $\neg$. The concepts of *free* and *bound* object variables are also defined in the usual way.

Several variations of queries may be formed. One class of queries deals only with behaviors that are without *side-effects*. A behavior is said to be *side-effect free* if it does not modify the state of any object or create new objects during its execution. This property is too restrictive in the context of our model since all operations (including the algebraic operators) are uniformly managed as behaviors. At minimum, a query will always return a new collection as a result and in certain cases will generate a new type for the collection as well. Thus, a small set of predefined behaviors, labelled as algebraic operators, will manage the controlled creation of collection and type objects as their side effects. It is a requirement of our calculus that all user-defined behaviors be side-effect free.

A distinction is commonly made [SS90] between *object preserving* and *object creating* operations. An *object preserving* operator produces results containing only existing objects from an objectbase. That is, it does not create or modify objects in any way. The query formalism of Straube and Özsu [SÖ90] only considered operations of the object-preserving kind. On the other hand, *object creating* operators allow for the "taking apart" and "putting together" of objects into various new structures, with new identity. The objects created (especially persistent objects) must be integrated into the underlying type system, including any derived types or classes necessary for the consistent existence of these new objects.

The terms *object-preserving* and *object-creating* require further clarification in the context of a uniform object model like TIGUKAT. Queries (at minimum) always create and return a new collection of objects. Furthermore, a query may create a new type object to go along with the collection if a proper type does not already exist. Therefore, all queries are object-creating in one sense. We make the following distinction between queries: if the result collection is formed from existing objects in the objectbase, the query is called *target-preserving*; otherwise the query is called *target-creating*.

A target-preserving query is an *object calculus expression* of the form $\{t \mid \psi\}$ where t is a target term consisting of a single variable, say o, possibly indexed by a set of behaviors, ψ is a WFF with o as the only free variable, and all behavior denotations in the expression are side-effect free.

Indexed variables are of the form $o[\mathcal{B}]$ where $\mathcal{B}$ is a set of object constants denoting behaviors defined on the type of o. The semantics of indexed terms is to *project* over the behaviors in $\mathcal{B}$ for o (possibly) creating a new type. Following a projection, only the behaviors given in $\mathcal{B}$ are applicable to objects in the result collection of the query.

Target-preserving queries may seem, at first glance, to be somewhat simplistic and too restrictive, but this form supports a wide variety of useful queries. For example, assume finite classes **C_dept** and **C_emp** where **C_emp** objects have behaviors *B_dept* and *B_age* defined on them. The following target-preserving query returns a collection of department objects that have senior citizens working for them:

$$\{\ o \mid \mathbf{C\_dept}(o) \wedge \exists p(\mathbf{C\_emp}(p) \wedge o = p.B\_dept \wedge <p, 65>.B\_age.B\_greaterThan)\ \}$$

A target-creating query is an object calculus expression of the form $\{t_1, \ldots, t_k \mid \psi\}$ where the set of variables appearing in (possibly indexed) target terms $t_1, \ldots, t_k$ is precisely the set of free variables, say $\vec{o}$, in the WFF ψ. This form is a generalization of the target-preserving kind by allowing $k \geq 2$ target terms over $\vec{o}$ distinct object variables. The result of such a query is a collection of product objects. Say in the previous example we wanted to return (department, employee) pairs instead of just departments and that the returned employee objects project over behavior *B_age*. The target-creating query that produces this result is as follows:

$$\{\ o, p[B\_age] \mid \mathbf{C\_dept}(o) \wedge \mathbf{C\_emp}(p) \wedge o = p.B\_dept \wedge <p, 65>.B\_age.B\_greaterThan\ \}$$

Additional examples are presented in Section 7.

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

5 The User Language

The main function of the TIGUKAT language is to support the definition, manipulation and retrieval of objects in an objectbase. The language consists of three parts: the TIGUKAT Definition Language (TDL) which supports the definition of metaobjects (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.). Only TQL is presented in this paper; the complete specification of all languages is given in [Lip93, PLÖS93].

TQL is based on the SQL paradigm. We adopt this paradigm for various reasons. Most importantly, SQL is accepted as a standard query language in relational databases, and current work on SQL3 attempts to extend its syntax and semantics to fulfill requirements of object-oriented systems [Gal92]. 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. 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 are queryable, since queries operate on collections and always return finite collections as results. Query results can also be used in the *from* and *where clauses* of other queries (nested queries). Finally, objects can be queried regardless of whether they are persistent or transient.

We 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.

There are four basic TQL operations: **select, insert, delete**, and **update**, and 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 syntax[5]:

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

The *select clause* in this statement identifies objects that are 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 free variables in object calculus formulas. The *into clause* declares a reference to a new collection which may be made persistent. If the *into clause* is not specified, a new transient collection is created, but there is no reference to it. 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 a result of a subquery where a subquery can either be given as a select statement or as a reference to a query object. Thus, a range variable in the *from clause* is of the form:

<*range variable*>: <*id list*> **in** <*collection reference*> [+]
<*collection reference*>: <*term*> | (<*subquery*>)

Note that the *collection reference* in the range variable definition can be followed by a plus '+' which refers to the shallow extent when the collection reference is a class. The default is deep extent for classes. The *id list* is a list of object variables appearing in the select clause and the *term* is either a constant reference, a variable reference or a path expression.

The *where clause* defines a boolean formula which must be satisfied by 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:

<*atom*>: <*term*> = <*term*>
| <*term*> **in** <*collection reference*> [+]

and *term* is 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* which is of the form:

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 structure:

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 which 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 part of the boolean formula definition is the *boolean path expression* which is equivalent to:

<*path expression*> = **TRUE**|**FALSE**

where the *path expression* is the TQL equivalent of a Bspec. However, to avoid such an artificial construct, we include a boolean path expression in the definition of a TQL formula under two conditions. First, all behaviors in the expression must be *side-effect-free* and second, the result type of the path expression must be boolean.

TQL supports three binary operations: **union, minus**, and **intersect**. The syntax of these statements is as follows:

<*collection reference*> **union** <*collection reference*>
<*collection reference*> **minus** <*collection reference*>
<*collection reference*> **intersect** <*collection reference*>

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 can be found in [Lip93, PLÖS93].

[5] The 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 '>'; a vertical bar '|' separates alternatives; and the square brackets '[', ']' enclose optional material which consists of one or more items separated by vertical bars.

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

6 The Object Algebra

The underlying framework of the object algebra and calculus are essentially the same. However, an important difference is that the algebra has a functional basis as opposed to the logical foundation of the calculus. In the algebra, names are used as placeholders for collections of objects with the appropriate types. The predicates $=, \neq, \in, \notin$ and connectives $\wedge, \vee, \neg$ are handled as boolean functions. There is a small set of well-defined functions (the algebraic operators) that provide meaningful iterations over collections and can be composed to form queries (existential and universal quantification are handled by composing these operators). Thus, a query is a functional expression to be evaluated and the algebra is a functional language.

Operands and results of the object algebra operators are typed collections of objects. Thus, the algebra is *closed* since the result of any operator may be used as the operand of another. Let Φ represent an operator in the algebra. The notation $P\ \Phi\ \langle Q_1, \ldots, Q_n \rangle$ is used for expressions where P and each Q_i are names for typed collections of objects. They represent the arguments to Φ. When $n = 1$ we use $P\ \Phi\ Q$ and when $n = 0$ we use $P\ \Phi$ without loss of generality. The collections represented by P and Q_i may be names for base collections or the result of an algebraic subexpression. Since the model supports substitutability, any specialization of collection, including classes, may be used as an operand. Similar to the range predicates of the calculus, we define P^+ to denote the shallow extent when P is the name for a class.

Some algebraic operators are qualified by a *boolean-valued formula* (i.e., a predicate). A predicate F is a boolean-valued functional expression that is composed from Bspecs, mops and the functions for $=, \neq, \in, \notin, \wedge, \vee, \neg$. The arguments to F are permutations of the members of the operand collections for the algebraic operator that F is defined on. The arguments of predicate F must be type consistent with the operand collections. Predicate qualified operators are written as $P\ \Phi_F\ \langle Q_1, \ldots, Q_n \rangle$ where F is a predicate with arguments, say $p, q_1, \ldots, q_n$, that range over the elements of collections $P, Q_1, \ldots, Q_n$ respectively.

The object algebra defines both *target-preserving* and *target-creating* operators. The target-preserving operators are as follows:

Set Operations The typical set **union**, **difference** and **intersection** operators are defined. In addition, a **collapse** operator is defined for flattening collections of collections.

Select (denoted $P\ \sigma_F\ \langle Q_1, \ldots, Q_n \rangle$): Select is a higher order operation accepting a function, the predicate F, and the n+1-ary collections $P, Q_1, \ldots, Q_n$ as arguments. The result collection contains objects from P corresponding to the p component of each permutation $<p, q_1, \ldots, q_n>$ that satisfies F.

Map (denoted $Q_1 \gg_{mop}\ \langle Q_2, \ldots, Q_n \rangle$): where *mop* is a side-effect free behavior application that is type consistent with the membership types of $Q_1, Q_2, \ldots, Q_n$. Map applies the behaviors of *mop* to each permutation of objects $< q_1, q_2, \ldots, q_n >$. The results of the application are returned in the result collection.

Project (denoted $P\ \Pi_{\mathcal{B}}$): where $\mathcal{B}$ is a collection of behaviors with the restriction that it be a subset of the behaviors defined on the membership type of P. The $\mathcal{B}$ collection is automatically unioned with the behaviors of type `T_object` in order to ensure consistency. The result collection contains objects of P, but with the membership type coinciding with the behaviors of $\mathcal{B}$. The $\mathcal{B}$ collection has no impact on which objects appear in the result collection of the query. It is only important during the final type assignment which occurs at type inferencing time after the extent of the query has been produced. This form of project differs from the traditional one in that it does not project over the structure of objects, but rather over their behavioral semantics.

The full object algebra includes target-creating operators in order to provide necessary object formation operations. The result of these operations is a collection of new objects that are object identity distinguishable from the ones in the argument collections. The primary target-creating operator is *product*:

Product (denoted $Q_1 \times \cdots \times Q_n$): where $n \geq 2$. Product produces a collection containing product objects created from each permutation $< q_1, \ldots, q_n >$ such that component q_i is an object from Q_i. Product may initiate the creation of a new type along with a new class to maintain the product objects.

If the *mop* of a **map** operator is not side-effect free and creates new objects (e.g., it may contain an algebraic operation), then the map operation is target-creating.

The above collection of operators form the *primitive* algebra (some refer to this as a *physical* algebra). They are fundamental in supporting the expressive power of the calculus and other expressions can be defined in terms of them. We add the following operators to the primitive algebra and call it the *extended* algebra (some call this a *logical* algebra). These operators provide useful functionality, generalize the expressive power of the algebra and are important for higher-level optimizations [SÖ90].

Join (denoted $P \bowtie_F \langle Q_1, \ldots, Q_n \rangle$): where $n \geq 1$. Join produces a collection of product objects created from each permutation $<p, q_1, \ldots, q_n>$ that satisfies F.

Generate Join (denoted $Q_1\ \gamma_g^o\ \langle Q_2, \ldots, Q_n \rangle$): where g is a generating atom of the form $o\ \theta\ < \vec{q} > .\vec{b}$ (where θ is one of $=$ or $\in$) over the elements of collections $Q_1, Q_2, \ldots, Q_n$. Generate join produces a collection of product objects created from each permutation of the q_i's and extended by an object o in the following way. If θ is $=$, the result contains product objects of the form $<q_1, q_2, \ldots, q_n, <q_1, q_2, \ldots, q_n>.\vec{b}>$ for each permutation of the q_i's (i.e., each product object is a permutation of the q_i's extended by the result of the mop for that permutation). If θ is $\in$, the result contains product objects $< q_1, q_2, \ldots, q_n, o >$ for each permutation of the q_i's and each $o \in <q_1, q_2, \ldots, q_n>.\vec{b}$ (i.e., for a permutation of the q_i's and for each member o of the collection resulting from the mop $< q_1, q_2, \ldots, q_n > .\vec{b}$, a product object $< q_1, q_2, \ldots, q_n, o >$ is created as a member of the result collection).

Reduce (denoted $P \triangle_{\vec{o}}$): where P is a collection of product objects and $\vec{o}$ is a list representing symbolic references to the component domains of the product. The reduce operator has the effect of discarding the $\vec{o}$ components of the objects in P. That is, product objects of the form $< p_1, \ldots, p_i, \vec{o}, p_{i+1}, \ldots, p_n >$ are mapped

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

to $<p_1, \ldots, p_i, p_{i+1}, \ldots, p_n>$. This is similar to the relational projection operator except that the specified components are removed. If P is not a product object, the empty collection is produced.

7 Example Objectbase

In this section, we present some examples on a geographic information system (GIS) objectbase to demonstrate the power of the query model and its languages. This example is selected because it is among the application domains which can potentially benefit from the advanced features offered by object-oriented technology.

A type lattice for a simplified GIS is given in Figure 2. The example includes the root types of the various sublattices 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_display` 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_shape` defines the geometric shape of the regions representing the various zones. For our purposes we only use this general type. In more practical applications this type would be specialized into subtypes representing polygons, polygons with holes, rectangles and so on. Table 1 lists the signatures of the behaviors defined on the GIS specific types. The specification `T_set`$\langle T \rangle$ where T is a type is used to denote a collection type whose members are of type T.

The following examples illustrate possible queries on the GIS. They are first expressed in TQL (T:), followed by the corresponding object calculus expression (C:) and then the equivalent algebraic expression (A:). In the algebraic expressions, we subscript an operand collection by the variable which ranges over it. This variable is used as a symbolic reference to the elements of the collection. We also use symbols $R1$, $R2$ as temporary results and "$\leftarrow$" for assignment. Furthermore, we use the arithmetic notation for operations $o.B\_greaterThan(p)$, $o.B\_elementOf(p)$, etc., instead of the boolean path expressions.

Example 7.1 Return land zones valued over $100,000 or cover an area over 1000 units.

T: **select** o
from o **in C_land**
where $(o.B\_value() > 100000)$ **or** $(o.B\_area() > 1000)$

C: $\{o \mid \mathbf{C\_land}(o) \wedge (o.B\_value > 100000 \vee o.B\_area > 1000)\}$

A: $\mathbf{C\_land}_o\ \sigma_{[o.B\_value > 100000\ \vee\ o.B\_area > 1000]}$

Example 7.2 Return all zones which have people living in them (the zones are generated from person objects).

T: **select** o
from q **in C_person**
where $(o = q.B\_residence().B\_inZone())$

C: $\{o \mid \exists q(\mathbf{C\_person}(q) \wedge o = q.B\_residence.B\_inZone)\}$

Type	Signatures	
`T_location`	*B_latitude*:	`T_real`
	B_longitude:	`T_real`
`T_display`	*B_display*:	`T_display`
`T_window`	*B_resize*:	`T_window`
	B_drag:	`T_window`
`T_shape`		
`T_zone`	*B_title*:	`T_string`
	B_origin:	`T_location`
	B_region:	`T_shape`
	B_area:	`T_real`
	B_proximity:	`T_zone` $\rightarrow$ `T_real`
`T_map`	*B_resolution*:	`T_real`
	B_orientation:	`T_real`
	B_zones:	`T_set`$\langle$`T_zone`$\rangle$
`T_land`	*B_value*:	`T_real`
`T_water`	*B_volume*:	`T_real`
`T_transport`	*B_efficiency*:	`T_real`
`T_altitude`	*B_low*:	`T_integer`
	B_high:	`T_integer`
`T_person`	*B_name*:	`T_string`
	B_birthDate:	`T_date`
	B_age:	`T_natural`
	B_residence:	`T_dwelling`
	B_spouse:	`T_person`
	B_children:	`T_person` $\rightarrow$ `T_set`$\langle$`T_person`$\rangle$
`T_dwelling`	*B_address*:	`T_string`
	B_inZone:	`T_land`
`T_house`	*B_inZone*:	`T_developed`
	B_mortgage:	`T_real`

Table 1: Signatures of example specific types in Figure 2.

A: $\left(\mathbf{C\_person}_q\ \gamma^{o}_{o=q.B\_residence.B\_inZone}\right) \triangle_q$

Example 7.3 Return the maps with areas where senior citizens live.

T: **select** o
from o **in C_map**
where exists (**select** p
from p **in C_person**, q **in C_dwelling**
where $(p.B\_age() \geq 65$ **and** $q = p.B\_residence()$
and $q.B\_inZone() \in o.B\_zones()))$

C: $\{\ o \mid \mathbf{C\_map}(o) \wedge \exists p(\mathbf{C\_person}(p)$
$\wedge\ \exists q(\mathbf{C\_dwelling}(q) \wedge p.B\_age \geq 65$
$\wedge\ q = p.B\_residence \wedge q.B\_inZone \in o.B\_zones))\}$

A: $R1 \leftarrow \mathbf{C\_person}_p\ \sigma_{p.B\_age \geq 65}$
$\left(\mathbf{C\_map}_o \bowtie_F \langle \mathbf{C\_dwelling}_q,\ R1_p \rangle\right) \triangle_{p,q}$
where F is the predicate:
$q = p.B\_residence \wedge q.B\_inZone \in o.B\_zones$

Example 7.4 Return all maps that describe areas strictly above 5000 feet.

T: **select** o
from o **in C_map**
where forAll p **in** (**select** q
from q **in C_altitude**
where $q \in o.B\_zones())$
$p.B\_low() > 5000$

C: $\{\ o \mid \mathbf{C\_map}(o) \wedge \forall p(\neg\mathbf{C\_altitude}(p)$
$\vee\ \neg(p \in o.B\_zones) \vee p.B\_low > 5000)\}.$

A: $R1 \leftarrow \mathbf{C\_altitude}_p\ \sigma_{\neg(p.B\_low > 5000)}$
$\mathbf{C\_map} - \left(\left(\mathbf{C\_map}_o \bowtie_{p \in o.B\_zones} R1_p\right) \triangle_p\right)$

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

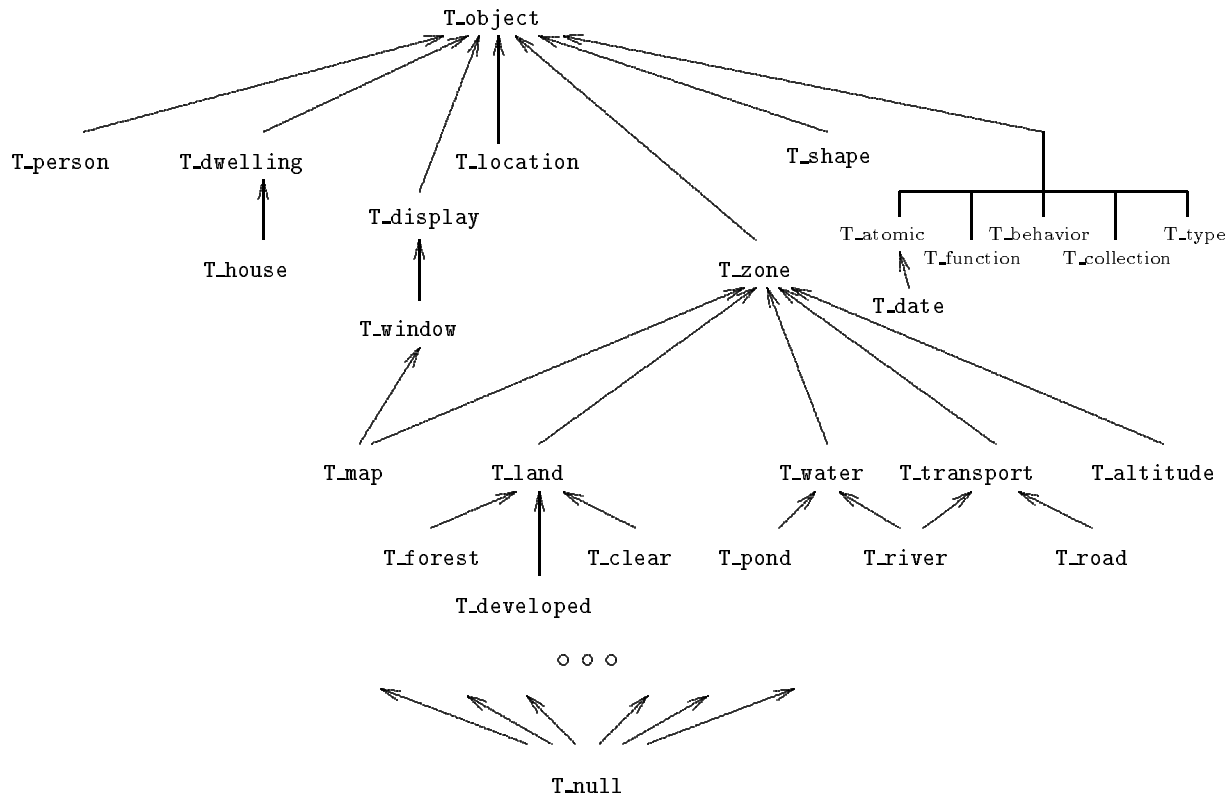

Figure 2: Type lattice for a simple geographic information system.

Example 7.5 Return the dollar values of the zones that people live in.

T: **select** $p.B\_residence().B\_inZone().B\_value()$
from p **in C_person**
C: $\{ o \mid \exists p(\mathbf{C\_person}(p)$
$\wedge\ o = p.B\_residence.B\_inZone.B\_value)\}$.
A: $\left(\mathbf{C\_person}_p\ \gamma^{o}_{o=p.B\_residence.B\_inZone.B\_value}\right) \triangle_p$
This has a simplier form using map as follows:
$\mathbf{C\_person}_o \gg_{o.B\_residence.B\_inZone.B\_value}$

Example 7.6 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*.

T: **select** $o[B\_title, B\_area]$
from p **in C_map**, o **in** $p.B\_zones$, q **in C_water**
where $o.B\_proximity(q) < 10$
C: $\{ o[B\_title, B\_area] \mid \exists p \exists q(\mathbf{C\_map}(p) \wedge \mathbf{C\_water}(q)$
$\wedge\ o \in p.B\_zones \wedge o.B\_proximity(q) < 10)\}$.
A: $R1 \leftarrow \mathbf{C\_map}_p\ \gamma^{o}_{o \in p.B\_zones}$
$R2 \leftarrow \left(R1_{p,o} \bowtie_{o.B\_proximity_{(q)<10}} \mathbf{C\_water}_q\right) \triangle_{p,q}$
$R2\ \Pi_{B\_title,B\_name}$

Example 7.7 Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map.

T: **select** $p,\ q.B\_title()$
from p **in C_person**, q **in C_map**
where $p.B\_residence().B\_inZone() \in q.B\_zones()$
C: $\{p, o \mid \exists q(\mathbf{C\_person}(p) \wedge \mathbf{C\_map}(q) \wedge o = q.B\_title$
$\wedge\ p.B\_residence.B\_inZone \in q.B\_zones)\}$
A: $\left(\mathbf{C\_person}_p \bowtie_F \left(\mathbf{C\_map}_q\ \gamma^{o}_{o=q.B\_title}\right)_{q,o}\right) \triangle_q$
where F is the predicate:
$p.B\_residence.B\_inZone \in q.B\_zones$

8 Conclusions and Future Work

The development of an efficient and effective query processor for an object management system requires a formal specification of all its components including the user level language, the object calculus, the object algebra and the completeness of the languages. In this paper, we present the framework of the formalization of the TIGUKAT query model [PLÖS93]. This specification is being used as a foundation for implementing the query model.

The query model is defined in a consistent way as type and behavior extensions to the base object model. This is a uniform object-oriented approach to developing an extensible query model that is seamlessly integrated with the object model. This kind of natural extension is possible due to the uniformity built into the TIGUKAT object model which treats everything as a first-class object and allows the consistent abstraction of an object's "attributes" into the uniform semantics of behaviors.

The TIGUKAT Query Language (TQL) is a user-level language with similarities to the SQL3 standardization efforts [Gal92]. The formal object calculus is a powerful declarative object creating language that incorporates the behavioral paradigm of the object model. Safety is based on the evaluable class of queries [GT91] which is arguably the largest decidable subclass of the domain independent class

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

[Mak81]. The calculus includes atoms for object restriction and object generation which extend the evaluable class. The object algebra includes a complete set of functional operators that fully support the object-creating nature of the calculus. The novel operators are the behavioral projection which is a form of type generalization and the generate join which extends a join with a mapping of behaviors on the operand collections. The calculus and algebra are equivalent in expressive power and the reduction from TQL to the calculus is complete. Object creating languages require the ability to perform schema evolution because the new objects may not correspond to any type in the lattice. As part of the algebra, we define how the operators relate to the schema in terms of the creation and integration of new types [PLÖS93].

There are a number of ongoing research activities related to this project. A main memory version of the TIGUKAT object model has been implemented. We are coupling this implementation with the EXODUS storage manager [CDV88] to provide persistence. A compiler for the query model and user language presented in this paper is being implemented on top of the object model implementation. Furthermore, we have completed the design of an extensible query optimizer for the algebra and it too is in the implementation stage. The optimizer has been developed as a uniform extension to the object model and will therefore be integrated just like the query model.

Another issue we are addressing is the definition of update semantics for the model. We have defined the syntax and semantics of a TIGUKAT Definition Language (TDL) which allows for the consistent creation of schema such as types, classes and collections. Furthermore, we have also defined TQL statements **insert, update** and **delete** to perform updates on the objectbase and are currently working out the semantics of these constructs. A related issue involves handling behaviors with side effects and we are hopeful in developing some rules for dealing with these in our languages.

References

[AB93] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. Technical report, INRIA, France, 1993.

[Bla91] J.A. Blakeley. DARPA Open Object-Oriented Database Preliminary Module Specification: Object Query Module. Technical report, Texas Instruments, December 1991.

[CDV88] M. Carey, D.J. DeWitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 413–423, September 1988.

[Gal92] L.J. Gallagher. Object SQL: Language Extensions for Object Data Management. In *Proc. of the 1st International Conference on Information and Knowledge Management*, pages 17–26, November 1992.

[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.

[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 *Proc. of the Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 406–416, September 1986.

[Kim89] W. Kim. A Model of Queries for Object-Oriented Databases. In *Proc. of the 15th Int'l Conf. on Very Large Databases*, pages 423–432, August 1989.

[Lip93] A. 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.

[Mak81] J.A. Makowsky. Characterizing Data Base Dependencies. In *Proc. of the* 8^{th} *Colloquium on Automata, Languages and Programming*, pages 86–97. Springer Verlag, 1981.

[Osb88] S.L. Osborn. Identity, Equality and Query Optimization. In *Proc. of the 2nd Int'l Workshop on Object-Oriented Database Systems*, pages 346–351. Springer Verlag, September 1988.

[ÖSP94] M.T. Özsu, D.D. Straube, and R.J. 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.

[PLÖS93] 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, Edmonton, Alberta, Canada, June 1993.

[PÖS92] R.J. Peters, M.T. Özsu, and D. Szafron. TIGUKAT: An Object Model for Query and View Support in Object Database Systems. Technical Report TR92-14, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, October 1992.

[SÖ90] 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.

[SS90] M. Scholl and H. Schek. A Relational Object Model. In *Proc. of the 3rd Int'l Conf. on Database Theory*, pages 89–105, December 1990.

[SZ90] G. Shaw and S. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proc. of the 6th Int'l. Conf. on Data Engineering*, pages 154–162, February 1990.

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.

[YO91] L. Yu and S.L. Osborn. An Evaluation Framework for Algebraic Object-Oriented Query Models. In *Proc. of the 7th Int'l. Conf. on Data Engineering*, pages 670–677, April 1991.

Appears in *Proc. of the Second Int'l Conf. on Information and Knowledge Management* (CIKM'93), pages 403-412, November 1993.