TIGUKAT:
An Object Model for Query and View Support
in
Object Database Systems

Randal J. Peters
M. Tamer Özsu
Duane Szafron
{randal,ozsu,duane}@cs.ualberta.ca

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

Technical Report TR 92-14
October 1992

Abstract

Object-oriented computing is influencing many areas of computer science including software engineering, user interfaces, operating systems, programming languages and database systems. The appeal of object-orientation is attributed to its higher levels of abstraction for modeling real world concepts, its support for incremental development and its potential for interoperability. Despite many advances, object-oriented computing is still in its infancy and a universally acceptable definition of an object-oriented data model is virtually nonexistent, although some standardization efforts are underway. This report presents the TIGUKAT[1] object model definition that is the result of an investigation of object-oriented modeling features which are common among earlier proposals, along with some distinctive qualities that extend the power and expressibility of this model beyond others. The literature recognizes two perspectives of an object model: the structural view and the behavioral view. Most object-oriented formalisms have concentrated on one or the other of these two perceptions. The TIGUKAT object model (i) favors formal specifications for both the behavioral and structural components, and (ii) shows an integration of the two components into a complete model definition. Furthermore, the model emphasizes (iii) a clean separation and precise definition of object modeling features which are usually bundled and only intuitively defined in other models, and (iv) a uniform approach to object modeling which encompasses all information, including meta-data, as objects within the primitive definitions.

The establishment of a formal object model provides a theoretical foundation from which investigations of other object database features will be pursued. The first of these involves the definition of an object query model with complete calculus and algebra specifications, including an equivalence transformation proof between the two. Based on the query model, object views and view management with update semantics will be examined. The intention is to incorporate very powerful and dynamic object definition and manipulation language features into the model. In parallel with this, a dynamic schema evolution policy will be developed for integrating new types into the model. We are currently in the process of implementing the model presented in this report to establish a workable software base with the long range goal of developing a full-featured object database management system.

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Object-oriented database systems are seen as the next evolutionary step towards advancing database technology. In addition to subsuming the modeling power and expressibility of the first-generation (i.e., hierarchical and network) and second-generation (i.e., relational) systems, an object-oriented database management system (OODBMS) is well suited for overcoming their deficiencies in handling complex data with complex relationships. Furthermore, an OODBMS is suitable for integrating the semantically rich, and computationally complete, discipline of programming languages with the persistence, performance and consistency features offered by traditional database systems.

Recent work on OODBMSs has resulted in a number of object model proposals. Most properties of these models have emerged from prototypes built for specific application domains that object-oriented technology is intended to serve. Consequently, these models vary considerably in the set of features they support. Most of them incorporate a set of common core concepts, but the semantics of these concepts lack precise formal definitions. There are a number of recent efforts aiming to formalize the characteristics of "object-orientation," but the field is generally suffering from the absence of a universally accepted object model, along the lines of the relational model [Cod70], whose features are formally and unambiguously defined. This void makes it difficult to reason about the internal consistency of these models, investigate database features such as query models, views, transaction management, etc., and to generalize the results of various studies all of which require a precise data model definition.

One result of our research is the development of an advanced object model through the identification and formalization of object-oriented characteristics with sufficient power and flexibility for supporting the advanced functionality demanded by OODBMSs. In this paper we propose a formalization of an object model which includes many of the core concepts introduced by former data models, along with additional features which extend its modeling power and expressibility[1]. The establishment of the model reported here forms a basis from which investigations of other object database features have commenced. The first of these involves the definition of an object query model with complete calculus and algebra specifications, including an equivalence transformation proof. Second, object views and view management facilities with update semantics are being pursued. The intentions of these two steps are to incorporate very powerful and dynamic object definition and manipulation language features into the model. In parallel with these steps,

[1] In this proposal, the term "object model" is preferred over "object-oriented data model" because a model of "objects" is one goal of the research. This perspective complies with the proposal of the ANSI Object-Oriented Database Task Group [FKMT91].

a dynamic schema evolution policy is being developed. Finally, we are attempting to integrate the performance and expressibility of the object query model (including views) with the computational completeness of object-oriented programming languages, resulting in a methodology for developing a coherent and type consistent object database programming language.

A fundamental feature of object models, which differentiate them from others, is their richer semantics. This enables closer modeling of real world concepts, making object models more powerful. On the other hand, it is more difficult to specify a clean, well-defined, universally accepted model. In fact, the power and expressibility of a general object model may prove too difficult to formalize because many important properties become intractable as the model becomes more general [Mai89]. However, certain precautions may be taken to avoid the pitfalls while developing a sound and complete object model definition [KW89, Bee90]. The resulting definition may be more restrictive than a "general" model, but we trade some power and expressibility (which may not be needed anyway) for tractability.

A model for objects involves the specification of two basic components. One part consists of the behavioral aspects which define a universal conceptual abstraction of objects and the relationships among them. The other is the structural definition which specifies the internal organization of objects and how their relationships are structured. Subtleties, like the differences between objects and values, which are hidden by the abstraction of the behavioral model are exposed at the structural level. King [Kin89] states that there is a similarity between a structural object model definition and the semantic data modeling approach [HM78, HM81, AH84, HK87, PM88, JGF+88] in the sense that both are concerned with the representation of data and knowledge. He equates a behavioral model definition to object-oriented modeling which involves access and manipulation of objects from general programming languages and query languages. In this sense, the behavioral model is like a calculus definition which specifies the "what" of object existence and manipulation, while the structural model is similar to an algebra which defines "how" the objects exist and are represented. Thus, a natural conclusion of this perspective is that the two specifications are independent and an equivalence relation between them should be expressible.

These two issues have traditionally been treated separately with object model formalization studies emphasizing one or the other. A notable exception is [Bee90] which attempts to establish a link between the two, even though the behavioral and structural definitions of that model are not fully developed.

The structural and behavioral aspects are both important to the development of an object model, but the two are independent, which accounts for the orthogonality of directions taken by recent studies. However, these definitions must eventually be reconciled in order to form a complete, unambiguous and operational model. This report presents the results of our research in formal specification and integration of behavioral and structural object concepts leading towards the development of the TIGUKAT object model.

The behavioral features of our model extend those of an earlier model [SÖ90a], making it more complete and significantly more powerful. The emphasis of the earlier work was query processing methodology in object-oriented database systems [SÖ90b, SÖ91, Str91], and the object model presented in [SÖ90a] was restricted to include only those features which are essential for that purpose. A result of that work was the identification of necessary design requirements for an object model definition with the appropriate primitives for object query model support [ÖS91, ÖSP92]. TIGUKAT evolved from these requirements and has the power to meet the demands of advanced application domains that object-oriented technology is intended to serve. Furthermore, the model

is more complete in that both the behavioral and structural aspects are presented; the previous work only considered behavioral notions.

The behavioral aspects are fundamental in developing a theory of objects [Ken90a]. However, in order to fully understand the semantics of an object model it is imperative that the formal behavioral model be coupled with a formal structural counterpart. Beeri's formal structural model [Bee90] has been chosen as a basis for the structural component that we integrate with our behavioral model. Several modifications have been incorporated into Beeri's model in order to extend its capabilities to match the enhanced functionality provided by the behavioral component. The integration of these two definitions results in a complete uniform object model specification which will serve as a favorable platform for the implementation of TIGUKAT.

The fundamental novel contributions of the TIGUKAT object model are as follows:

- a formal specification and integration of both the behavioral and structural aspects of an object model with the necessary power for handling advanced database functionality such as object creating query languages, schema evolution, updatable views, rules, etc.;
- a clean separation and a precise formal definition of many object model features which are usually bundled and only intuitively defined in other studies; and
- a uniform approach to objects which includes meta-information as primitive objects in the model resulting in a very powerful and flexible self-contained model.

The remainder of this report is organized into five chapters. In Chapter 2 we present an overview of TIGUKAT's features and define the scope of its functionality. A geographic information system (GIS) is also described and will be used as a running example throughout the report. In Chapter 3 we present, in detail, the behavioral notions of the TIGUKAT object model. In Chapter 4 we supplement the behavioral model with a structural definition which specifies an implementation independent representation for the conceptual objects of the behavioral model, including their relationships through behaviors. This is done in parallel to defining the integration with the behavioral model. An object graph representation which pictorially illustrates the structural model is also presented. In Chapter 5 we survey earlier work in the area of object modeling. In particular, we examine some models which have influenced the design of TIGUKAT. Furthermore, we compare the characteristics of our model to the object-oriented manifestos [ABD+89, SRL+90] and the NIST standards report [FKMT91]. Finally, in Chapter 6 we reexamine the contributions of the TIGUKAT object model and outline the subsequent research we will be pursuing as part of the TIGUKAT project.

Chapter 2

Object Model Overview

The object model proposed here is founded on a high-level behavioral specification with object uniformity being an integral part of the definitions. The semantics of TIGUKAT is specified by a complete set of definitions and is integrated with an example structural model to further clarify its functionality. The model is *behavioral* in the sense that all access and manipulation of objects occurs through behaviors (operations) which are defined on types, and it is *uniform* in that everything in the model is a *first-class object*. The integration with a structural counterpart illustrates how the behavioral concepts can be organized at a structural level. This gives indication of a basis for a clean interface to an object storage manager subsystem. It is important to stress that the choice of a structural counterpart is orthogonal to the behavioral specification of TIGUKAT. The only requirement is that the structural component support the full functionality outlined by the behavioral model.

The *uniformity* aspects of TIGUKAT are similar to the approaches of DAPLEX [Shi81], its object-oriented counterpart OODAPLEX [Day89] and FROOM [MB90]. As mentioned earlier, we adopt another significant facet of these models: their functional approach to defining behaviors. However, we go further by including enhanced functionality along with a full set of precise specifications and an integration with an example structural counterpart. *Uniformity* is paramount in TIGUKAT and is carried through to both the behavioral and structural specifications.

The behavioral model evolves from the definitions of several primitives. The primitives form a foundation that supplies the necessary tools from which other constructs such as user-defined and system objects may be created and extended. These operations have a single, clean semantics because of the uniformity incorporated into the primitives. The primitive components of TIGUKAT include *types, classes, collections, behaviors, functions atomic entities* and, of course, *objects*. In fact, at a conceptual level, all elements of the model have the uniform perspective of an object. Thus, the primitive elements of our model consist of *type objects, class objects, collection objects, behavior objects, function objects* and *atomic objects*. The primitive object system evolves with other application specific objects by applying the primitive behaviors we provide. These include behaviors for object creation, manipulation and management along with several others which are defined in the following sections.

An *object* is an abstraction for encapsulating information into a single entity which may be operated upon as an individual. An object is accessible only through a set of publically defined behaviors which constitutes the *interface* of the object; this is known as the *encapsulation property*.

Furthermore, TIGUKAT supports *strong object identity* [KC86], meaning that every object has a unique, immutable identifier associated with it which differentiates the object from all others.

Object accessibility in TIGUKAT is achieved through the notion of *object reference* which is the only expressible form of an object. A reference serves as a handle or locator for the object. References are associated with a particular *scope* and their meanings may vary over different scopes. Unlike object identities, references need not be unique. That is, there may be many references to a particular object. The exact specification of scope and reference is outside the domain of TIGUKAT. These are left to be precisely defined by application domains based on the model. For example, different object programming languages may have varying levels of scoping which may differ from scoping in query languages and graphical user interfaces.

Throughout this paper, we assume a functional programming environment as a global scope and adopt the following prefix notation and font variations to denote object references of the various primitive kinds within this scope.

`T_name` is a type object reference,

C_name is a class object reference,

L_name is a collection object reference,

B_name is a behavior object reference,

name is some other application specific reference.

In this notation, the prefixes `T_`, **C_**, **L_**, and *B_* distinguish between the various primitive object types where the "name" part is an object specific reference name. The last notation, which does not include any specific prefix, refers to other system and user defined objects which are not of a previously mentioned primitive kind. They may include any sequence of characters but should not normally begin with one of the established prefixes. For example, `T_person` is a type object reference, **C_person** a class reference, **L_seniors** a collection reference, *B_age* a behavior object reference, and a reference such as sherry without any specific prefix represents some other application specific object reference. In some cases, we prefer the use of mathematical symbols over named references. This is done for both convenience and brevity. A full representation using named references is always given as an alternative to the symbolic notations.

We separate the means for defining the characteristics of objects (i.e., a *type*) from the mechanism for grouping instances of a particular type (i.e., a *class*). A *type* is used to specify the structure and behavior of objects of that type. The type serves as an information repository (template) of characteristics common among all objects of that particular type. Types are organized into a lattice structure using the notion of *subtyping* which promotes software reuse and incremental type extensibility.

A *class* ties together the notions of *type* and *object instance*. A *class* is a supplemental construct to a type responsible for managing all instances created using that type (known as the *extent* of the type). Objects of a particular type cannot exist without an associated class and every class is uniquely associated with a single type. In other words, there is a total (into) mapping *classof* which maps objects to classes and a total injective (one-to-one and into) mapping *typeof* which maps classes to types. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes* which imply *types* (i.e., *object* $\Longrightarrow$ *class* $\Longrightarrow$ *type*).

Object creation occurs only through a class using the associated type as a template for its objects. In this sense, our model is similar to others such as EXODUS [CDV88], but we automatically maintain the extent of a type through its class. Defining object, type and class in this manner introduces a clear separation of these concepts.

In addition to classes, we define a *collection* as a more general grouping construct. A *collection* is similar to a *class* in that it groups objects, but it differs in the following respects. Firstly, no object creation may occur through a collection; object creation occurs only through classes. This means that collections only form groupings of existing objects. Secondly, an object may exist in any number of collections, but its participation in classes is restricted by the lattice structure on types. Finally, whereas a class groups the entire extension of a single type and its subtypes (i.e. homogeneous objects), a collection may be heterogeneous in the sense that it can contain objects which may be of any type. The targets of queries are classes and collections. Subsequently, the results of queries are also classes and 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 [Bla91, YO91].

Other fundamental notions of TIGUKAT are those of *behavior* and the objects which implement them, which we call *functions* (also known as *methods*). In the same way as object specifications (types) are separated from the groupings of their instances (classes and collections), we separate the definition of a behavior from its possible implementations (functions/methods). Behaviors provide the only means of operating upon objects. Behaviors define a semantics which describe their functionality. All objects supporting the functionality of a particular behavior incorporate that behavior object into the interface of their type. Functions implement the semantics of behaviors; we say they provide the *operational* semantics of the behavior. The implementation of a particular behavior may vary over the types which support it. Nonetheless, the semantics of the behavior remains constant and unique over all behaviors. The implementation of a behavior may consist of runtime calls to executable code which is known as a *computed* function. Alternatively, it may simply be a reference to an existing object in the database in which case it is called a *stored* function. The uniformity of TIGUKAT considers each behavioral application as the invocation of a function, regardless of how the function is implemented (i.e., stored, computed, etc.).

A semantic description of a behavior may be a complicated expression. One approach is to define the functionality of behaviors using *denotational semantics* [Sto77, All86, Sch88, CP89]. A common approach in other models is to define a *signature* for the behavior which gives the behavior a name (reference) used for behavioral application along with the type specification of its arguments and the type of its result. Although somewhat simplistic and semantically incomplete, a signature expression does offer a useful and necessary tool for defining the structure and interface of behaviors. We incorporate signatures into our behavioral descriptions and are currently investigating other approaches such as the denotational semantics for supplementing these definitions. In the remainder of the paper we assume the existence of a mechanism for defining and distinguishing the semantic expression of behaviors.

Functions are objects which may be broken down into constituent parts that include source and implementation components. A source component would resemble a human readable definition of the function's operation (behavior) usually written in some object-oriented programming language, but could additionally include English commentary and further semantic descriptions. The implementation component would consist of executable code (in the form of a computed function) or a reference to a particular result object (in the case of a stored function). The functional approach adopted by TIGUKAT benefits from the significant amount of research which has been done in

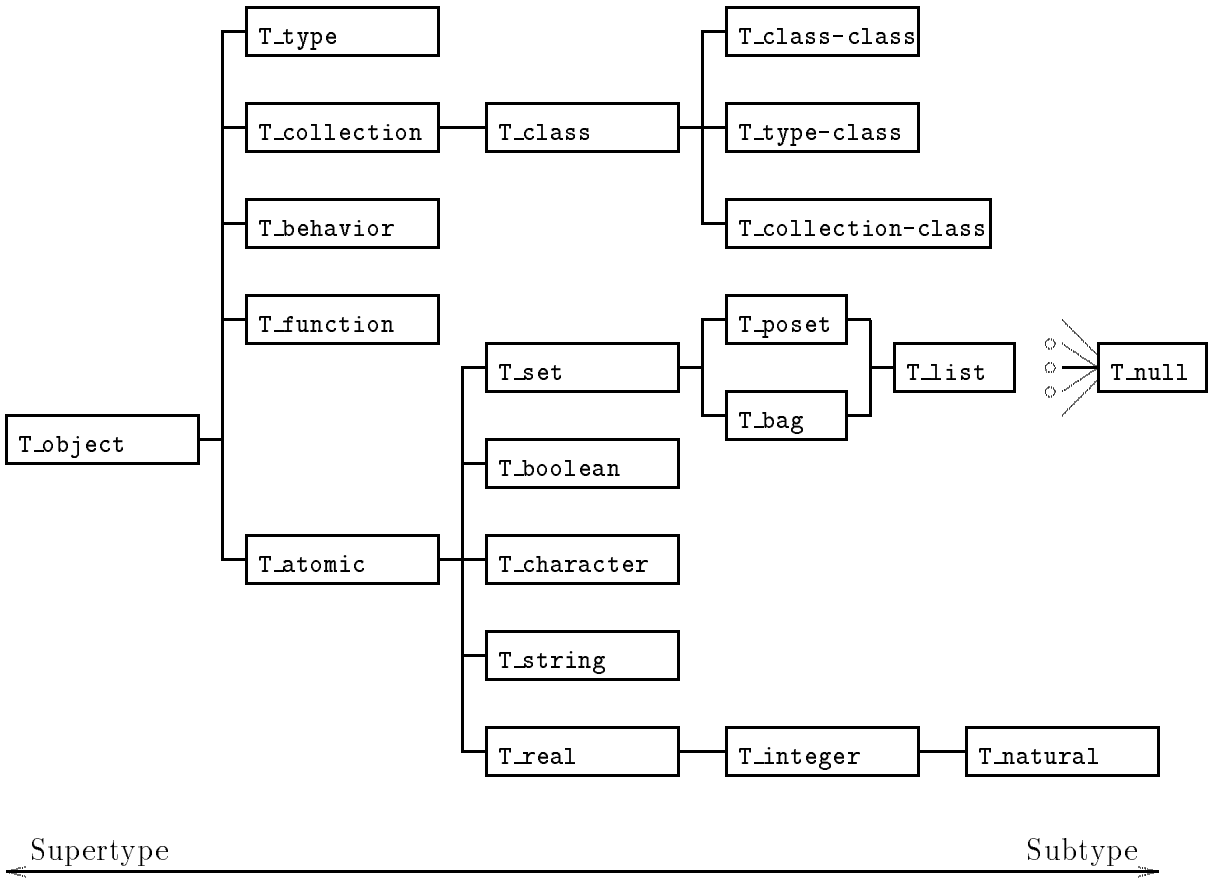


Figure 2.1: Primitive type system $\mathcal{T}$.

the areas of functional programming languages and functional theory such as the lambda calculus [Bar81, Rev89] and category theory [Pie88, LS86].

The primitive type lattice of TIGUKAT is depicted in Figure 2.1. Uniformity dictates that everything in the model be an object; types, classes, collections, behaviors, functions, and so on, are all defined and managed as objects. The introduction of uniformity eliminates the need for externally maintained meta-information since all information, including the meta-data, is self-contained within the model as objects. An additional benefit is that the limitless hierarchy of meta, meta-meta, etc. information is eliminated by incorporating these levels into a single self-contained structure.

Figure 2.1 illustrates the subtyping relationships of the primitive type system. Each box in the figure represents a primitive type and the edges between the boxes denote the well-known notion of *subtyping* which is depicted from left to right (i.e., the type `T_type` is a subtype of type `T_object` and so on). Types are identified by an appropriate reference given within each box. The types of Figure 2.1 are formally addressed in the following sections and a complete specification of the fundamental behaviors of each type are given in the design document for the implementation of the model.

The type structure of Figure 2.1 is referred to as the *primitive type system* $\mathcal{T}$. Every type in $\mathcal{T}$ is associated with a corresponding primitive class object which may possibly contain some other primitive objects. Furthermore, types define primitive behaviors which are associated with

functions that implement the functionality defined by the behaviors. We define the union of the types in $\mathcal{T}$ with the set of all primitive classes, behaviors, functions and other instance objects as the *primitive object system* $\mathcal{O}$.

The type `T_null` in $\mathcal{T}$ is a primitive type defined to be a subtype of all other types. It is a system managed type and included to allow the use of nulls in a consistent fashion. The type `T_null` binds the type lattice from the bottom while the type `T_object` binds it from the top.

From the type structure of Figure 2.1, we can begin to understand the uniformity of TIGUKAT and the relevance of the statement "*everything is an object.*" Our model restricts dynamic type creation in that all types must be in a subtype relationship with `T_object`. Therefore, due to the semantics of subtyping, all behaviors defined on the type `T_object` are applicable to all objects of the system, including the type object `T_object` itself. This structured type lattice is important in maintaining the uniformity of the TIGUKAT object model.

A structural model maps the behavioral definitions into a representation that is consistent with a storage manager level interface. The structural level makes a cleaner distinction between atomic entities of the system and the structured objects which are constructed from them. At this level, the domains of the atomic types may be mapped into the semantics of *values*, which serve as identity, state and reference all at once and give these kinds of objects the properties of immutability.

Sets, bags, partially ordered sets and *lists* are atomic types which impose various levels of structuring on other domains. *Sets* are containers whose elements define the identity of the set. *Bags* are sets which allow duplication of elements. *Partially ordered sets* (i.e., *posets*) are sets with an ordering relation defined between pairs of its elements; total orderings are those posets which define the ordering relation between all pairs of elements. Finally, *lists* combine the properties of bags and posets allowing both duplication and ordering of its elements. These types are all maintained by the system and operations are provided for establishing references to objects of these types. The instances of *set, bag, poset* and *list* structured types are referred to as *structured values* in the structural model and are considered to be atomic and immutable. From the user's perspective, the entire domains of atomic types can be assumed to exist and can be manipulated using the operations we define. Exactly how this abstraction is maintained is implementation dependent. For example, referenced objects could be automatically created when they are accessed for the first time.

Abstract objects include the user-definable objects of the system (e.g., application specific objects, executable functions, etc.) along with some of the primitive system objects (e.g., primitive types, classes, behaviors, etc.). An abstract object, as a whole, envelopes the properties of immutability (and in this sense is atomic), but abstract objects incorporate a separate *state* which may change over time. There are two main reasons for considering abstract objects to be atomic. The first is related to the notion of strong object identity. Changing the state of an abstract object does not transform the object into some other object (i.e., the identity of the object does not change). Rather, it is still the same object it was before, only now it carries different information. In other words, abstract objects are atomic in the sense of their existence. The second reason deals with the representation of (possibly complex) objects in mathematical logic. In certain forums it is beneficial to consider abstract objects as atomic in this sense, because this perspective relates them to the first-order semantics of logic which is well-defined [Bee90].

The structural aspects of the model are clarified by the introduction of an object graph representation which is defined in Chapter 4. An object graph is used to portray the structure and

contents of an object database which uniformly stores application specific and primitive system objects alike. The nodes of the graph correspond to the atomic values, abstract objects and structured values of the objects in an object database while the edges represent the relationships (defined as behaviors) among the objects.

Each concept introduced in this section, although related, has a separate role in the model and each has a distinct semantics. The chapters which follow discuss these concepts in more detail and formalize their semantics, but first we describe a geographic information system (GIS) which will be used as a running example throughout the remainder of the report.

2.1 Example Database

Object-orientation is intended to serve many application areas requiring advanced data representation and manipulation. A geographic information system (GIS) [Aro89, Tom90] is selected as an example to illustrate the practicality of the concepts introduced and to assist in clarifying their semantics. A GIS was chosen because it is among the application domains which can potentially benefit from the advanced features offered by object-oriented technology. Specifically, a GIS requires the following capabilities:

1. management of persistent and transient data,
2. management of large quantities of diverse data types and dynamic evolution of types,
3. a seamless integration of sophisticated computer graphic images with complex structured attribute data,
4. handling of large volumes of data and performing extensive numerical tabulations on data,
5. management of differing views of data, and
6. the ability to efficiently answer a variety of ad hoc queries.

A GIS can be defined as an application "designed for the collection, storage and analysis of objects and phenomena where geographic location is an important characteristic or critical for analysis...In each case, what it is and where it is must be taken into account." [Aro89]. Some examples of this include displaying the effective range of a police force, illustrating how logging activities affect wildlife populations, or depicting the severity of soil erosion.

GIS technology is being applied to many areas. Some common ones include agriculture and land use planning, forestry and wildlife management, geology, archaeology, municipal facilities management, and more global scale applications such as ecology. Each of these areas rely on statistical data, historical information, aerial photographs, and satellite images for analyzing and presenting empirical data, for drawing conclusions about certain phenomena, or for predicting future events through sophisticated computer simulations using the information at hand. GISs require advanced information management and analysis features in order to be effective. Object-oriented databases have the potential to provide this advanced functionality.

A type lattice for a simplified GIS is given in Figure 2.2. The example is sufficiently complex to illustrate the advanced functionality of the model we present, yet simple enough to be

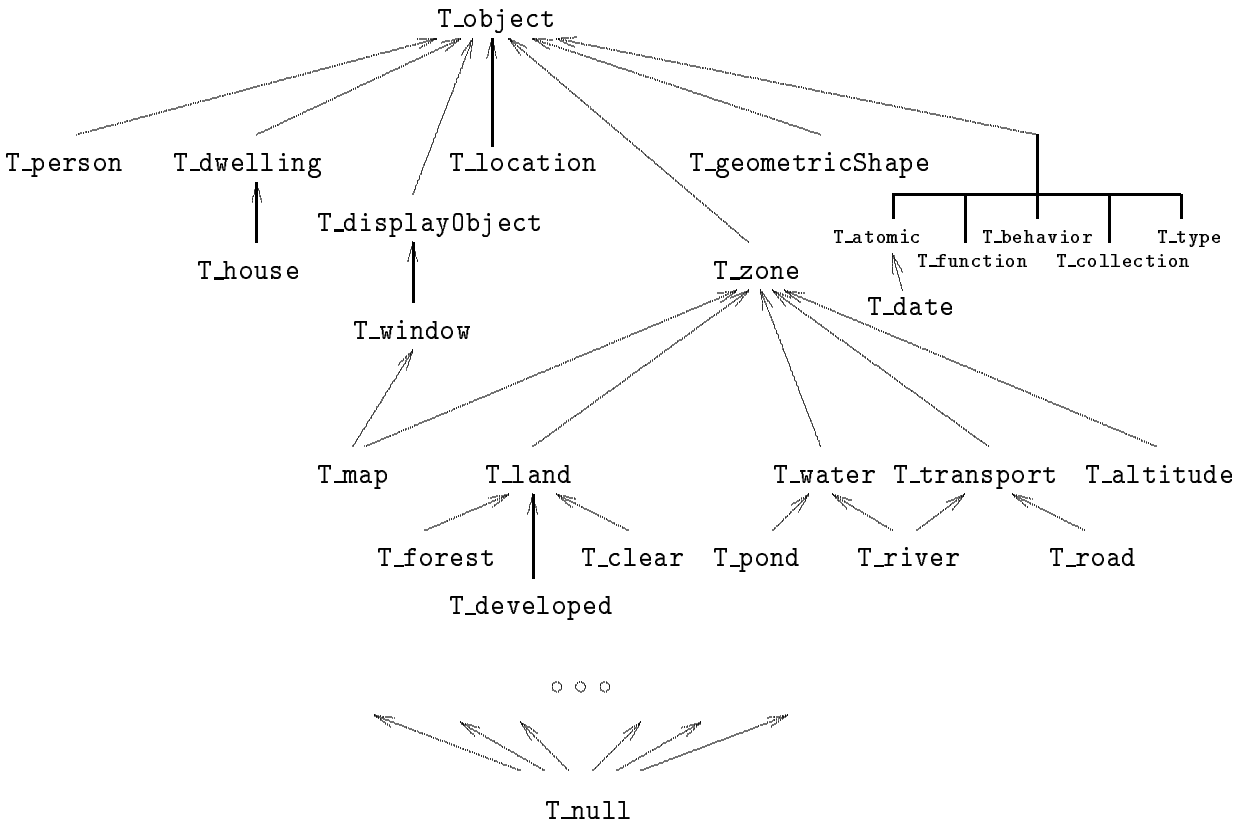


Figure 2.2: Type lattice for a simple geographic information system.

understandable without an elaborate discussion. The example includes the root types of the various sub-lattices of the primitive type system $\mathcal{T}$ to illustrate their relative position in an extended application lattice. The additional types defined by the GIS example include:

1. Abstract types for representing information on people and their dwellings. These include the types `T_person`, `T_date`, `T_dwelling` and `T_house`. Note that `T_date` is a new atomic type introduced by the application which is used to represent dates in a form acceptable to the application.

2. Geographic types to store information about the locations of dwellings and their surrounding areas. 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.

3. Displayable types for presenting information on a graphical device. These include the types `T_displayObject` and `T_window` which are application independent and the type `T_map` which is the only GIS application specific object that can be displayed.

4. A type `T_geometricShape` which 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, etc.

Table 2.1 defines the signatures of the GIS specific types in the lattice of Figure 2.2. The semantics of these behaviors will be illustrated in the chapters which follow. Furthermore, the signatures for the types of the primitive type system $\mathcal{T}$ will also be developed.

Type	Signatures	
T_location	*B_latitude*:	T_real
	B_longitude:	T_real
T_displayObject	*B_display*:	T_displayObject
T_window	*B_resize*:	T_window
	B_drag:	T_window
T_geometricShape		
T_zone	*B_title*:	T_string
	B_origin:	T_location
	B_region:	T_geometricShape
	B_proximity:	T_zone → T_real
T_map	*B_resolution*:	T_real
	B_orientation:	T_real
	B_zones:	T_set⟨T_zone⟩
T_land	*B_value*:	T_real
T_water	*B_volume*:	T_real
T_transport	*B_efficiency*:	T_real
T_altitude	*B_low*:	T_integer
	B_high:	T_integer
T_person	*B_name*:	T_string
	B_birthDate:	T_date
	B_age:	T_natural
	B_residence:	T_dwelling
	B_spouse:	T_person
	B_children:	T_person → T_set⟨T_person⟩
T_dwelling	*B_address*:	T_string
	B_inZone:	T_land
T_house	*B_inZone*:	T_developed[a]
	B_mortgage:	T_real

[a] Behavior was refined from supertype T_dwelling.

Table 2.1: Behavior signatures pertaining to example specific types of Figure 2.2.

Chapter 3

The TIGUKAT Object Model

In this chapter we concentrate on the behavioral issues of the TIGUKAT object model. We follow a formal presentation of the high-level abstract functionality of the model and at times address structural components to clarify certain points that are raised. However, a full integration of the behavioral model with an example structural counterpart is delayed until Chapter 4.

3.1 *Atomic* Types, Classes and Objects

Most data models include a set of basic primitive types referred to as *atomic types*. We include the common types `T_boolean`, `T_character`, `T_string`, `T_real`, `T_integer` and `T_natural` as part of our primitive model definitions. Furthermore, we include several primitive types for aggregating objects such as `T_set` for managing sets of objects, `T_bag` for defining sets of objects with possible duplication of elements, `T_poset` for defining ordering relations on sets of elements, and `T_list` for specifying both ordering and duplication on sets. We refer to the collection of atomic types as the *atomic type pool*. Other types may be easily added to this collection through the operation known as *subtyping*[1]. For example, the GIS application schema of Section 2.1 extends the atomic types with the type `T_date`.

Atomic types define the behaviors applicable to atomic objects of that type. Atomic objects are equated to the notion of *literals* defined in [FKMT91]. They are never explicitly created by the user. Instead, they are assumed to exist and users manipulate system maintained references to these objects; or create and use their own references derived from the primitive ones. For each atomic type, there exists a corresponding *atomic class* which groups the instances of that atomic type. Thus, we include an atomic class for each one of the atomic types.

Atomic types and classes are objects that are related to other types and classes in the model. For example, the atomic types are all objects of the primitive type `T_type` and are managed as instances of the primitive class **C_type**. The atomic classes are of type `T_class` and belong to class **C_class**. This structure follows from the uniformity aspects of the model.

TIGUKAT defines the usual standard behaviors for atomic types (i.e., behaviors which are commonly associated with objects of these types), and provides conventional syntactic representations of atomic objects to serve as references. Only brief descriptions are given here since these types are

[1] Subtyping is formally defined in Section 3.4.

universally known abstractions. The full behavioral specification of these types and their objects is defined in the implementation of the model.

Objects of the type `T_real` are represented as floating point numbers (e.g., -23.456 or 3.9E-3) with behaviors for the usual arithmetic operations such as *addition*, *subtraction*, *multiplication* and *division* along with relational operators ($<, >, \leq, \geq$). Equality is excluded from this list because it is defined as a behavior of the more general *object* type. *Integer* objects have the usual syntactic denotation as a string of digits (e.g., 12345) with an optional sign while *naturals* represent the subset of positive integers exclusively. The type `T_boolean` only includes the two instance objects true and false which have the usual logical operations. Objects of type `T_character` are represented as a single character in single quotes (e.g., 'x') that correspond to a particular collating sequence. Characters support comparison operators through their ordinal values. Strings are of type `T_string` and are represented by a sequence of characters in double quotes (e.g., "A string"). Strings support comparison operators by examining the ordinal values of their component characters and also include a variety of string manipulation behaviors.

Some atomic types (e.g., `T_integer`, `T_real`, `T_string`) represent an infinite domain of atomic objects. For all the atomic types, it is assumed that the instances of these types exist and that atomic objects serve as state, identity and reference all at once. For example, the atomic type `T_integer` has an infinite domain of objects whose elements serve as the reference, identity and state of their existence. The integer reference 5 refers to an integer object whose identity and state is the universally known abstraction of the integer 5. There is only one 5, there always has been and there always will be. Note that this does not restrict us from establishing additional references to the integer 5 such as five or V. The same argument holds for all types in the atomic type pool.

Structured object aggregates in TIGUKAT are provided by the primitive `T_set`, `T_bag`, `T_poset` and `T_list` types. These types are used to build typed groupings of objects with varying levels of duplication and ordering imposed upon the elements.

Sets have the standard mathematical notion of set where duplicates and ordering within a set are irrelevant and where the axiom of extensionality defines set equality. A set of objects is denoted as $\{o_1, o_2, \ldots, o_m\}$ where each of the o_i is an object reference. A *bag* is an object container which may hold duplicates, but where ordering is irrelevant. The type `T_bag` defines a behavior δ which, given a bag and an object reference, returns the number of times that object appears in the bag. A bag of objects is denoted as $(o_1, o_2, \ldots, o_m)$ where each of the o_i is an object reference. A partial order (*poset*) is a set with a reflexive, transitive and antisymmetric relation $\leq$ defined on pairs of elements from the set. Duplication within a poset is irrelevant. If every pair of elements in a poset is comparable (i.e., either $a \leq b$ or $b \leq a$ for all a, b in the poset), we say the set is *totally* ordered. A poset of objects is denoted as $< o_1, o_2, \ldots, o_m >$ where each of the o_i is an object reference. *Lists* are sets with both duplication and ordering properties imposed upon its elements. A list of objects is denoted as $[o_1, o_2, \ldots, o_m]$ where each of the o_i is an object reference.

These aggregate types may be specialized by subtyping the general types. One form of specialization is to define a subtype which restricts the elements of its instances to be of a particular type. We use parameterization to denote this form of refinement. The syntax for this is given as `T_set`$\langle$`T_X`$\rangle$, `T_bag`$\langle$`T_X`$\rangle$, `T_poset`$\langle$`T_X`$\rangle$ and `T_list`$\langle$`T_X`$\rangle$ where `T_X` represents some other type specification. This restricts the members of the aggregate type to be compatible with the type `T_X`.[2] The notion of type compatibility is described in Section 3.4. For example, consider the signature ex-

[2]The notations `T_set`, `T_bag`, `T_poset` and `T_list` are really only abbreviations for the parameterized notations `T_set`$\langle$`T_object`$\rangle$, `T_bag`$\langle$`T_object`$\rangle$, `T_poset`$\langle$`T_object`$\rangle$ and `T_list`$\langle$`T_object`$\rangle$ respectively.

pression *B_zones* : T_set⟨T_zone⟩ defined on type T_map of the example GIS schema. This denotes a behavior *B_zones* whose application to an object of type T_map results in a set whose members are all objects compatible with type T_zone.

The aggregation types encompass the standard semantics and operations of mathematical sets which include behavioral definitions for *union, intersection, difference* and *membership*. The objects of these domains are considered to be atomic, meaning that the contents of a set are immutable and uniquely identify the set. Thus, behaviors which seem to update sets are really mappings from one set to another. For example, one would refer to an integer set object containing the atomic integer objects 5 and 8 as $\{5, 8\}$. The operation of inserting an atomic object 4 into this set is really a mapping from the set object $\{5, 8\}$ to the new and distinct set object $\{4, 5, 8\}$. These semantics apply to all the atomic aggregate types. These types are similar to the *immutable aggregate types* of other models [Day89, MB90].

We do not include an explicit *tuple* type in our model. The notion of *tuple* can be cast into ordinary object definitions. Tuples are entities with *attributes* which define the *value* of the tuple. Objects are entities with *behaviors* which define the *state* of the object. A tuple can be mapped directly into the representation we propose for an object by mapping *attributes* to *behaviors* and *values* to *state*. Whenever a tuple definition is required, one may create a type where the attributes of the tuple are defined as the behaviors of the type. The values of the tuple attributes are accessed and manipulated by applying the behaviors to objects complying with the given type. Tuples and objects have an inherent uniform representation, and defining tuples in this way makes for cleaner and more concise semantics.

In addition to the atomic types, our model defines other primitive types for *objects, types, collections, classes, behaviors*, and so on. The following sections elaborate on each one of these primitives in turn.

3.2 *Behaviors* and *Functions*

Behaviors are objects which perform operations on objects and produce objects as results. Behaviors are defined on types and are applicable to the object instances which are compatible with that type. Each behavior includes a semantic expression of its functionality. It is this semantic expression which determines the uniqueness of a behavior. Types wishing to provide a particular semantics (behavior) must define the behavior object as part of their interface or have the behavior inherited. Equality for behaviors is refined to incorporate equality of semantic expression.

Different types may supply common behaviors (i.e., semantics). However, the implementation of these shared behaviors over the types may vary. Each type can define its own implementation for each of its behaviors as long as the implementations provide the specified semantics of the behavior which is unique. In this way, we separate the notion of a *behavioral specification* from its *implementation(s)* much in the same way as [MB90]. A type's behavior represents the only means of accessing and manipulating the properties of objects. Functions provide the necessary implementation forum for realizing behaviors. The TIGUKAT object model is strictly concerned with the behavioral aspects of operations and does not model implementation specific details. Together, behaviors and functions represent the operational semantics of the system.

Behaviors are all instances of the type T_behavior and the implementations of behaviors are instances of the type T_function. We use the standard mathematical arrow ($\rightarrow$) notation as a

syntactic representation for functions and curry multiple argument functions. In this way, a wide variety of other representations are supportable. A general function specification 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 type expressions may consist of any other type specifications (including function specifications).

We assume that functions, defined as implementations of behaviors, are unary in the sense that they have an argument expression $\mathcal{A}$ which consists of a single type that is compatible with the type the function is being defined on and a result type $\mathcal{R}$ giving the result type of its execution.

Types have an *extent* of objects which are grouped by a corresponding *class*. Types define a set of behaviors which are applicable to the objects in its extent (i.e., its class)[3]. Behaviors represent the only means of accessing and manipulating objects in a class and functions are the objects which implement these behaviors.

The semantic definition of a behavior can be specified in many ways. Some examples include using the code which implements the function as a specification, or using an informal English description, or possibly a more formal denotational specification. A simple method which is common among other models is the use of a *signature* expression for representing the meaning of a behavior. Signatures are useful and necessary for describing behaviors, but they are inadequate for characterizing the full semantics of behaviors. For now, we assume that a proper semantic specification mechanism for behaviors exists and that equality testing on behavioral semantics operates reliably. We further assume the existence of a behavior *B_semantics* (denoted $[\![\ ]\!]$) defined on the type `T_behavior` which returns the semantic specification of the behavior to which it is applied. For example, given a behavior b, the behavioral application *B_semantics*(b) (denoted $[\![b]\!]$) returns the semantic specification of b. Currently, we only define signatures for behaviors to give some indication of their semantics. In subsequent research we will define a more complete mechanism for specifying behaviors.

A *signature* specification consists of two elements; the *name* used to invoke the behavior and the *result type* of its application. We assume that the name for invoking the behavior has a standard string representation and the result type is one of the types available to the user. Since behaviors are always defined on a particular type, a behavioral specification may be thought of as a function with a single argument (an object of the type it is defined on) and a single result (an object of the type specified as the result; which may be a function). We formally define the representation of signatures as follows:

Definition 3.1 *Signature* ($f : R$)*:* A *signature* is a specification of behavior. It is denoted as $f : R$ which consists of a *name* (f) that is used to apply the behavior to an object and a *result type* (R) which specifies the type of the object resulting from the application of the behavior. □

In order to extract these two components of a behavior, we define primitive behaviors on the type `T_behavior`. These behaviors include *B_name* : `T_string` to access the name of the signature and *B_result* : `T_type` → `T_type` to return the result type of the behavior for a particular type. The names of behaviors cannot change over types which are in a subtype relationship with one another. However, the result type of a behavior may vary as long as it is compatible with the result type of the behavior at the supertype from where it was inherited. Type compatibility and subtyping is discussed in Section 3.4. Table 2.1 defines the signatures of the GIS example database.

[3]The relationships between *type, class* and *extent* are formally defined in Sections 3.4 and 3.5.

Behaviors are applied to objects. The object receiving the behavior is always explicit. This is similar to the classical or message-based object model outlined in [FKMT91]. The application of a behavior *B_i* to an object *o* is denoted as a function call *B_i*(o). The result of this application is an object which is in the extent of the result type specified by the signature of *B_i*.

For example, consider the signature *B_residence* : `T_dwelling` defined on the type `T_person` in Table 2.1. Applying *B_residence* to an object of type `T_person` results in the execution of the function object associated with this behavior which returns an object that is compatible with the type `T_dwelling`. The behavior then returns an object reference to the applicant who interprets this object as being of the type `T_dwelling`. If we used a full signature specification as is done in [SÖ90a], the signature would be written as *B_residence* : `T_person` → `T_dwelling`. Since a behavior is specified on a particular type, we omit the "`T_person` →" part of the specification and choose to derive this portion from the type the behavior is defined on. Now consider an object sherry which is an instance of type `T_person`. The application of *B_residence* to object sherry is denoted as *B_residence*(sherry) which results in an object reference that is in the extent of type `T_dwelling`.

An optional representation for behavioral application is the dot notation ($o.f$). The general form of this application is given as object.*B_behavior*(arg) which denotes the application of the behavior *B_behavior* to an object object using the object reference *arg* as an argument. This representation is equivalent to the function application (*B_behavior*(object))(arg) or simply *B_behavior*(object)(arg) assuming left associativity of function application. If no argument is required by the result type, then the application is simplified to object.*B_behavior* which is equivalent to *B_behavior*(object). Referring back to a previous example, applying the behavior *B_residence* to the object sherry using the dot notation would be specified as sherry.*B_residence*. The result of behavior application is an object reference which may additionally have other behaviors applied to it. Thus, the behavioral application itself may be thought of as an object reference. The dot notation presents an optional uniform specification for behavioral application which has a direct translation to the standard function application notation.

Functions have additional attributes such as source and implementation components. The source component would be a human readable form of the function written in some language including possible semantic descriptions or natural language explanations. The implementation component would be the executable form of the function that is invoked when the function object is called upon to perform its task. Some models [MB90] consider functions to be either *stored* or *computed*. A *stored* function is one whose implementation is simply a reference (pointer) to a object that is in the extent of the result type specified in the associated behavior's signature. A *computed* function is one which executes some code in order to produce the result object. Regardless of whether a function is stored or computed, a semantic description (i.e., behavior) of that function always exists. This description may be as simple as the signature specification, or a complex set of documentation together with a denotational semantics of its operation. The separation of behaviors from functions in TIGUKAT allows it to support any kind of function, including stored and computed, by abstracting the implementation specific details of functions with the semantic consistency of behaviors. A syntax for specifying functions and developing their code is one of the requirements of a database programming language for the system.

In Section 3.4 we define *subtyping* (also referred to as *behavioral inheritance*) as a reuse mechanism on the behavioral specifications of types. However, this has no implications on the reuse of implementations. Just because a type `T_`τ is a subtype of some other type `T_`σ does not mean that the behaviors of `T_`τ borrow any implementation from the behaviors defined on `T_`σ (although they may, or this may be the default). Because of this, we define a separate reuse mechanism

for implementations which we call *implementation inheritance*. Every behavior must relate each type it is defined on with an implementation of the behavior for that type. These implementations may be either the same function object in the subtypes, in which case we say the implementations are *inherited*, or the implementations are completely different objects, in which case we say the implementation is *redefined* or *overridden*. When overriding a function, the newly defined function must provide the semantics specified by the behavior to which it is related.

Implementation inheritance on functions could be defined as a default in accordance with subtyping on types. That is, a subtype *inherits* all the behaviors and implementations of its supertypes unless otherwise specified. We have separated the inheritance of behaviors from the inheritance of their implementations. One of the problems with inheritance arises from TIGUKAT's support of *multiple inheritance*; also referred to as *multiple subtyping*. Multiple subtyping means that a type can be a direct subtype of several other types. This complicates matters since a conflict resolution policy is needed for choosing an implementation when inheriting semantically common behaviors (with different implementations) from several types. Note that conflict resolution is not a problem with behavioral inheritance because it deals with semantics of behaviors which are preserved over type boundaries while the implementation of these semantics may differ over conflicting types. A simple, but effective, solution for conflict resolution is to have the user resolve the conflict by choosing one of the implementations or by redefining the implementation completely. Note that this resolution process need only occur at type definition time, and after an implementation has been established, the system can operate unambiguously.

The inheritance mechanism as well as the conflict resolution policy is implementation dependent and not part of the formal model definition. We have described some solutions which would be sufficient in resolving the problems, but delay an exact solution until the implementation of the model.

3.3 The *Object* Primitive

An *object* is a fundamental primitive in TIGUKAT because the conceptual level of the model deals uniformly with objects. In Chapter 2 we describe an object as an abstraction for encapsulating information and behavior into a single entity. The encapsulated portion of an object is referred to as its *state* which is accessible only through a set of behaviors defined on the type for that object. The state carries the information content of the object and each object has an associated *identity* which serves as an immutable internal identifier for the object throughout its existence. Thus, the model views an object as a pair consisting of an identity and a state.

Definition 3.2 *Object:* An *object* is defined as the pair (*identity, state*) where *identity* is the immutable internal identifier of the object and where *state* is an assortment of information carried by the object which is separate from the object's identity. □

An *object identifier* (or *oid*) is associated with an object upon its creation and persists with the object throughout its lifetime. In TIGUKAT, objects may be composed of other objects because the semantics of objects are defined by behaviors which are objects that operate on other objects and return objects as results. Therefore, the model automatically supports the notion of *complex objects*. The state of an object may be naively thought of as a composition of references to other

objects which defines that particular state. In this case, each object includes an *oid* along with some private state which holds the referential information for that object.

If we consider the domain of all objects as the collection of pairs consisting of all possible combinations of identity and state, then two unwanted inconsistencies arise. First, this domain will contain objects with the same identity, each associated with different states. This is obviously inconsistent because there is a single identity attempting to identify several semantically separate states. Second, this domain will contain objects with different identifiers, each associated with the same state. The inconsistency of this arrangement may not be as apparent. However, we consider a state to be an integral part of an object which is bound to, and inseparable from, its identity. If there are two or more identities associated with a single state, it is equivalent to saying there exists an object which is identified by more than one identity since the different identities share the single common state in its entirety. We overcome this potential source of confusion by considering a state as a unique component of an object. Note that we are simply enforcing the formation of states with identities to be unique and in no way are implying that the information carried by these states are unique. For example, if we consider the information within a state as a bit stream in memory, then the states of different objects would occupy different areas of memory, but the content (i.e., bit streams) of these states may be identical.

To eliminate these inconsistencies, we form the following definition of a consistent set of objects which gives a basis for object database construction. The definition assumes the existence of two internal functions which take an object as input and return an internal component (identity or state) of the object as output. The first function is denoted $oid(o)$ which returns the object identifier of an argument object o. The second is called $state(o)$ which returns the state formation associated with the object o. It is important to note that since these functions return internal components of objects, they are not normal functions with respect to the object functions defined in Section 3.2. These two functions are strictly internal to the system and in no way accessible to the user.

Definition 3.3 *Consistent Object Set (coset):* A set of objects O is consistent if and only if for all objects $o_i, o_j \in O$, $oid(o_i) \neq oid(o_j)$ and $state(o_i) \neq state(o_j)$ when $i \neq j$. Alternatively, if OID represents the domain of object identifiers from O and $STATE$ represents the domain of states. Then a function $stateof : OID \rightarrow STATE$ which maps *oids* to *states* is a bijective mapping. □

The definition of a consistent object set adheres to the notion of strong object identity [KC86]. That is, every object in a *coset* has an internal identifier which is unique from all others in the *coset*. This feature gives each object a unique existence within a *coset* and provides an unambiguous association with the state of the object. We define the primitive object system $\mathcal{O}$ as a *coset* of objects.

Axiom 3.1 The primitive object system $\mathcal{O}$ is a *coset*. □

Some argue [SRL+90, Bee90] that object identities should have the option of being either system or user assigned. In our model, all object identities are maintained automatically by the system without any user involvement. This is in keeping with the notion of strong object identity and has additional benefits when it comes to reconciling the components of distributed object bases and the variable interpretations that may exist among them. Nevertheless, user defined identities are supportable in our model even in the presence of strong object identity as we define it. They are possible through application specific interpretations. For example, a user may choose

to recognize one of the behaviors of an object (e.g., *B_social_insurance_number*) as an identifier for that object and all other objects like it. Our model places no restrictions on this kind of customized interpretation.

Object existence, access, and manipulation in TIGUKAT is based on the notions of *reference*, *scope* and *lifetime*. This is similar to other model proposals [Sny90, Ken90b, FKMT91] in that the only user expressible form of an object is a *reference* within a particular *scope*. A *scope* defines the visibility, access paths and lifetime of object references. A *reference* may be thought of as a *pointer* (or *handle*) to an object, which in turn leads to the object's *identity* and *state*. We use the notation $R_i@S_i$ to denote an object reference R_i in scope S_i. We shorten this to simply R_i when the scope is obvious or immaterial. The R_i component is a reference name adhering to the prefix notation outlined in Chapter 2. The *lifetime* of an object is different from the *lifetime* of a reference in that an object's lifetime is not necessarily dependent upon a particular reference within a certain scope. That is, when a reference or a scope disappears, the objects being referenced do not necessarily disappear along with it, but may persist past the lifetime of the scope or reference (i.e., lifetime of an object $\neq$ lifetime of a reference). However, if an object no longer has any references (system or user) maintaining its existence, then the object should be selected as a candidate for garbage collection. From the database perspective there is also the issue of explicit deletions. Deleting an object within a particular scope should guarantee that the object is no longer visible in that scope, but how this affects its visibility within other scopes that are concurrently referencing the object is part of a concurrency control mechanism and is not addressed in the primitive model definitions. The exact semantics of object deletion and garbage collection is outside the scope of this paper. Figure 3.1 is an example of an object reference model which illustrates the relationships among scope, reference, identity and state.

In Figure 3.1 there are the two scopes S_1 and S_2. The scope S_1 could be an application programming environment while S_2 may be an interactive query processor. The exact semantics of the scoping rules is defined by the application accessing the database and may vary over applications. Scoping is not specifically part of the object model itself. Within scope S_1 there are the three object references R_1, R_2 and R_3. References R_1 and R_2 refer to the same object identified by I_1, and R_3 refers to the object identified by I_2. Within scope S_2 there are two object references, R_3 and R_4. In this scope, R_3 refers to the same object as R_1 and R_2 do in scope S_1, and R_4 refers to the object identified by I_3 which is unrelated to scope S_1. This example clearly shows the one-to-one and onto (bijective) mapping *stateof* from object identities to their states, and the various mappings from references over scopes S_1 and S_2 to their associated objects. The heavy dark line around the objects indicates the boundary of the TIGUKAT object model. If, for example, we consider everything within the boundary as being persistent (i.e., we assume a persistent object store), then if the scopes disappear, the objects will persist past the lifetime of the scopes. We emphasize the separation of the object model which provides the object management facility from the reference/scope model which makes use of this facility. When referring to objects we use the terms "object" and "object reference" interchangeably.

All object manipulations are performed through behaviors. In order to maintain uniformity, we introduce the existence of a type object called `T_object` on which we define behaviors that are applicable to all objects in the system. Since object access is given through references, all behaviors defined on objects are always applied to object references within a particular scope; this in turn applies it to the actual objects (i.e. the identity and state) and returns a resulting object reference in that scope.

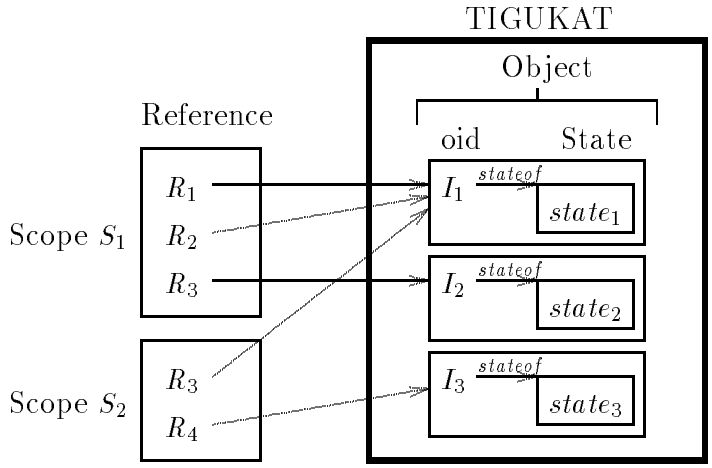


Figure 3.1: An object reference example.

Scope S_1	Scope S_2	Scopes S_1 and S_2
$R_1 = R_1$	$R_3 = R_3$	$R_1@S_1 = R_3@S_2$
$R_2 = R_2$	$R_4 = R_4$	$R_1@S_1 \neq R_4@S_2$
$R_3 = R_3$	$R_3 \neq R_4$	$R_2@S_1 = R_3@S_2$
$R_1 = R_2$		$R_2@S_1 \neq R_4@S_2$
$R_1 \neq R_3$		$R_3@S_1 \neq R_3@S_2$
$R_2 \neq R_3$		$R_3@S_1 \neq R_4@S_2$

Table 3.1: Object equalities of Figure 3.1.

The first behavior we define is that of *equality*. This behavior is necessary since we require a mechanism to determine if two object references are actually referring to the same object. The *equality* behavior is defined on the type object `T_object` which makes it applicable to all objects.

Behavior 3.1 *Object Equality, (B_equal :* `T_object` $\rightarrow$ `T_boolean`*) (=):* For any two object references R_i and R_j and their respective scopes S_i and S_j, the result of $(B\_equal(R_i@S_i))(R_j@S_j)$ is true if and only if $R_i@S_i$ and $R_j@S_j$ map to the same object identity in the domain of object identities (i.e., the object identities are the same). We introduce the infix binary relational operator "=" as a shorthand for *B_equal*, expressing the above behavioral application of *B_equal* as $R_i@S_i = R_j@S_j$. We similarly define the inverse relation $\neq$ to test for inequality. The result of equality is an object reference to one of the atomic boolean objects true or false. Object equality/inequality is reflexive, symmetric and transitive. □

Table 3.1 lists the equalities/inequalities which result in true among the references of Figure 3.1 over the two scopes. The first column shows the equalities in scope S_1, the second in scope S_2 and the third lists the equalities over both scopes.

This is the only kind of equality our model defines at the object level. It is quite strong in that the only way two object references are considered equal is if they actually refer to the same object 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 [SZ89, SZ90]. These notions can be defined as identity 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, the model may provide the classic shallow and deep equivalence through behaviors which evaluate and determine the equivalence of objects based on the identity equivalence of their component behaviors. However, this is strictly a design decision that should be left for the implementation phase of a particular system. Dayal [Day89] also makes this argument by stating that there are many notions of equality and those other than "identity equality" are best left for the "customizers" of the model to define the ones which are of most utility to them. For example, our model specializes equality for behaviors to mean semantic equality and equality for atomic objects to mean value equivalence.

Note that equality testing at the object identity level is transparent to the reference model and is an operation provided by the system through the internal *oid*() function. This is necessary since the identities serve as part of the representation of objects and are not objects themselves. Including identities as objects, in one sense, cleans up the semantics of certain definitions, but poses problems in many other aspects. The deciding argument which suggests that identities should not be treated as objects has to do with the circularity of definitions which arise if identities are objects. If an identity is an object, then by definition it must consist of an identity (and a state), but this new identity must be an object, which must consist of an identity (and a state), etc.. A fix-point for this recursive definition is not obvious and has led to the development of a consistent approach which does not treat identities as objects.

Our model is strongly-typed, meaning each object is associated with a particular type. That is, there is a strong implication from object to type (*object* $\Longrightarrow$ *type*). A type defines all the behaviors applicable to the objects of that type. It is important in type-checking and query processing to know the type of an object [SÖ90b] (or a conformance of types for an object). Therefore, it is necessary to define a behavior on objects which, when applied to an object, returns another object that represents the type of the argument object. We say that every object *maps to* a particular type. The *B_mapsto* behavior is defined on the type `T_object` making it applicable to all objects.

Behavior 3.2 *Maps to (B_mapsto :* `T_type`*)* ($\mapsto$)*:* For an object reference o, the behavioral application *B_mapsto*(o) is defined to be the singleton type object reference `T_`τ that represents the type of object o. We use the notation $o \mapsto$ `T_`τ to denote that object o *maps to* type `T_`τ (i.e. ($o \mapsto$ `T_`τ) $\Longrightarrow$ (*B_mapsto*(o)= `T_`τ)). □

For example, if the object sherry is an instance of the type `T_person`, then *B_mapsto*(sherry) returns the type object `T_person`. Using the symbolic notation, we specify sherry $\mapsto$ `T_person`. Extending this uniformly to types, we have *B_mapsto*(`T_person`) returning the type object `T_type` and *B_mapsto*(`T_type`) returning `T_type` as well. Thus, `T_type` is a fix-point for the *B_mapsto* behavior. Symbolically we have:

$$\begin{aligned} \text{sherry} &\mapsto \texttt{T\_person} \\ \texttt{T\_person} &\mapsto \texttt{T\_type} \\ \texttt{T\_type} &\mapsto \texttt{T\_type} \end{aligned}$$

A model must also supply a mechanism for removing objects from the system. In our model, there can be many references to an object. Therefore, the removal of an object (within a particular

scope) consists of severing the link between the reference and the object itself. This process does not necessarily destroy the object because other references may still be valid and in use (i.e., reference lifetime $\neq$ scope lifetime). When no references to an object exist, we say that the object is *dangling*. A garbage collection policy could be employed to reclaim the storage occupied by dangling objects. Since this is an implementation issue, it is not part of the formal model definition. Concerning the primitive and atomic objects, these are system defined objects and the system always maintains a reference to them. Therefore, these objects are not endangered of becoming dangling objects and being removed by a garbage collection manager.

There are other object behaviors whose development is delayed until the formalization of subsequent primitive concepts because of their dependency on these concepts. We introduce these behaviors after their foundations are established.

In summary, an object in the most general sense consists of an internal identity and a private state which is manipulated through a set of public behaviors defined by the object's type. Our definition of object allowed for several inconsistencies when considering collections of objects and so the definition of a consistent object set (*coset*) was formed as a basis for model development. Objects are accessed via references within particular scopes and have a lifetime independent of scope. We have defined some primitive behaviors on the type `T_object` which include *object equality B_equal* (=) and *B_mapsto* ($\mapsto$). These behaviors are primitives on the type `T_object`, and therefore are universally known and applicable to all objects in the system.

3.4 The *Type* Primitive

Types define the behavioral characteristics of objects. A *type* specifies the set of behaviors which are applicable to objects that have been created, or exist, in accordance with that type specification. Every object is associated with a type which gives the functionality of the object in terms of the behaviors it supports. The idea of associating every object with a type is known as *strong typing*.

The uniformity aspects of TIGUKAT imply that types must also be objects with their own state and identity along with their own type. The state of a type object consists of an internal structural specification of its instances (a template) along with references to encapsulated behaviors which forms the public *interface* of its instances. The type which describes all other type objects is a primitive in the model and is defined as the type `T_type` which is also a type (i.e., `T_type` $\mapsto$ `T_type`). The type `T_type` is a fix-point for the *B_mapsto* type referencing behavior. The type `T_type` must supply the necessary tools to develop other types. These include tools for behavior definition, subtyping, specialization, type/schema evolution, rules, etc. with the stipulation that these tools are provided as behaviors which act upon other type objects so that the uniformity of the model is maintained. Furthermore, the type `T_type` must be accessible in the same manner as any other object. Thus types, in addition to serving as descriptions of objects, are objects themselves and the type `T_type` serves as the description of all other types; this is known as the **type:type** property. The issue of **type:type** is controversial, particularly in the area of programming languages. Fortunately, some functional language specifications where the **type:type** property holds have emerged [Car86].

From the behavioral perspective, a type only defines the semantics for the behaviors of its instances. To ensure encapsulation, the behavioral model only publicizes the interface of a type while keeping the implementations private. Behaviors are either explicitly defined on a particular

type or are inherited from a supertype. Behaviors which are explicitly defined on a type are called *native behaviors* and those which exist due to subtyping are called *inherited behaviors* of the type. We now define two primitive behaviors on the type `T_type` which return the set of *native* and *inherited* behaviors for any particular type.

Behavior 3.3 *Native Behaviors (B_native* : `T_set⟨T_behavior⟩`*):* For a given type `T_`τ, the behavioral application *B_native*(`T_`τ) returns the set of behavior objects which have been explicitly defined as native behaviors of type `T_`τ. □

Behavior 3.4 *Inherited Behaviors (B_inherited* : `T_set⟨T_behavior⟩`*):* For a given type `T_`τ, the behavioral application *B_inherited*(`T_`τ) returns the set of behavior objects which have been inherited by type `T_`τ. □

The entire *public interface* of a type can be derived by forming the union of the *native* and *inherited* behaviors of the type. For convenience, we define a behavior *B_interface* : `T_set⟨T_behavior⟩` which returns the union of of the native and inherited behaviors as described. Additional operations are defined on the interface sets to provide facilities for adding, deleting and updating behaviors. These operations address issues of *schema evolution* which is part of the future research of this project.

Two relationships among types have been identified [ÖSP92]. The first is the concept of a type *specializing* another type in a manner similar to what is described in [MZO89]. The second is the more popular, and stronger, notion of explicitly declaring a type to be a *subtype* of another type [Car84]. We define *specialize* as a binary relation on types which determines whether one type specializes another. A specialization is determined from the semantic characteristics of behaviors.

Behavior 3.5 *Specialize (B_specialize* : `T_type` → `T_boolean`*)* ($\sqsubseteq$)*:* A *specialize* relation $\sqsubseteq$ between pairs of types `T_`τ, `T_`σ is a reflexive, transitive relation such that (*B_specialize*(`T_`τ))(`T_`σ) (denoted `T_`$\tau \sqsubseteq$ `T_`σ) is `true` if and only if *B_interface*(`T_`σ) $\subseteq$ *B_interface*(`T_`τ). This is interpreted as, type `T_`τ *specializes* type `T_`σ if and only if the behavioral specification of `T_`τ subsumes the behavioral specification of `T_`σ. If `T_`$\tau \sqsubseteq$ `T_`σ and `T_`$\sigma \sqsubseteq$ `T_`τ, then either the behavioral specifications of `T_`τ and `T_`σ are identical or `T_`τ and `T_`σ refer to the same type object (i.e., `T_`τ = `T_`σ). □

For every type, there may exist a collection of objects which belong to that type. This is known as the *extent* of the type which is important in the context of *subtyping*. Subtyping, like specializing, is a binary relation on types, but is stronger in the sense that it defines a partial ordering of the type lattice and a subset inclusion relationships on extents.

Behavior 3.6 *Subtype (B_subtype* : `T_type` → `T_boolean`*)* ($\preceq$)*:* A *subtype* relation $\preceq$ between pairs of types `T_`τ, `T_`σ is a reflexive, transitive and antisymmetric relation such that the behavioral application (*B_subtype*(`T_`τ))(`T_`σ) (denoted `T_`$\tau \preceq$ `T_`σ) is `true` if and only if type `T_`τ has been explicitly declared as a subtype of type `T_`σ. The term `T_`$\tau \preceq$ `T_`σ is interpreted as `T_`τ is a subtype of `T_`σ and implies that:

1. `T_`$\tau \sqsubseteq$ `T_`σ,
2. the behaviors of `T_`σ are inherited by `T_`τ (i.e., *B_inherited*(`T_`τ) = *B_interface*(`T_`σ)), and

3. the extent of $\texttt{T\_}\tau$ is a subset of the extent of $\texttt{T\_}\sigma$.

We can equally say that $\texttt{T\_}\sigma$ is the *supertype* of $\texttt{T\_}\tau$. □

Subtyping is a stronger relationship than specialize in several respects. First, the subtype relation ($\preceq$) defines a partial order on types while specialize ($\sqsubseteq$) does not, because specialize is not antisymmetric. That is:

$$\begin{array}{lllll} \texttt{T\_}\tau \preceq \texttt{T\_}\sigma & \text{and} & \texttt{T\_}\sigma \preceq \texttt{T\_}\tau & \Longrightarrow & \texttt{T\_}\tau = \texttt{T\_}\sigma, \text{ but} \\ \texttt{T\_}\tau \sqsubseteq \texttt{T\_}\sigma & \text{and} & \texttt{T\_}\sigma \sqsubseteq \texttt{T\_}\tau & \not\Longrightarrow & \texttt{T\_}\tau = \texttt{T\_}\sigma \end{array}$$

Second, all behaviors of the supertype are automatically inherited by the subtype which implies these behaviors cannot be native. Note that this only refers to the *behavioral inheritance* which is different from *implementation inheritance*; the implementation of inherited behaviors may change in the subtype as long as they provide the semantics specified by the behavior. For types in a specialize relationship only, common behaviors may be redefined as native behaviors in each of the types. Lastly, subtyping defines a subset inclusion relationship on type extents while no such property is enforced for specialize. A type may be declared as a subtype of several other types, meaning that a type can have many supertypes and also many subtypes. This is sometimes referred to as *multiple inheritance* [Car84], but we prefer the term *multiple subtyping*. It follows from this property that a type can also specialize many types and be specialized by many other types. Multiple subtyping requires a conflict resolution scheme to select a proper implementation when a type inherits semantically common behaviors (with different implementations) from different types. We consider the definition of this protocol to be an implementation issue and therefore do not include it as part of the primitive model definition. A simple approach is to enforce the user to resolve the conflict by either choosing one of the possible implementations or redefining the implementation altogether. Note that conflict resolution is only a problem in implementation inheritance and is not required for behavioral inheritance due to our assumption that semantic definitions of behaviors are powerful enough to express uniqueness which persists across type boundaries.

The definition of subtyping leads to the axiom of root type which imposes structure on the schema of types and is important for the maintaining the model's integrity and consistency.

Axiom 3.2 *Root Type:* for all types $\texttt{T\_}\tau$, $\texttt{T\_}\tau \preceq \texttt{T\_object}$. □

The axiom of root type states that all type objects are subtypes of the type object `T_object` which forms the root of the type lattice. This axiom is important in that it forces all types in the system to support the behaviors defined on type `T_object`, meaning the behaviors defined on `T_object` are applicable to all objects in the system. In other words, the axiom forces everything to be an object.

Every type, together with its supertypes, form a strong mathematical structure called a *complete lattice*. We introduce this structure and establish its role in the model through the definition of a supertype lattice behavior on the type `T_type`. The following definitions reference a type system denoted $\mathcal{T}'$ which we define to include the primitive type system $\mathcal{T}$ together with all application specific types supplementing $\mathcal{T}$.

Behavior 3.7 *Super lattice (B_super-lattice :* `T_poset⟨T_type⟩`*)* ($\diamond$)*:* For a given type $\texttt{T\_}\tau$, *B_super-lattice*($\texttt{T\_}\tau$) (denoted as $\diamond_{\texttt{T\_}\tau}$) returns a set of types, partially ordered by $\preceq$ (i.e., a poset),

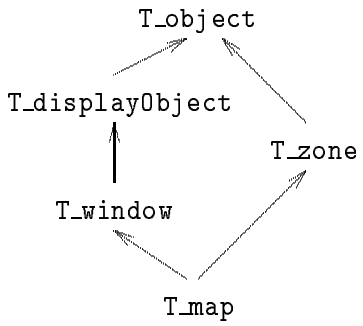


Figure 3.2: Super lattice of type `T_map`.

such that for all types $\texttt{T\_}\sigma \in \Diamond_{\texttt{T\_}\tau}, \texttt{T\_}\tau \preceq \texttt{T\_}\sigma$ and there does not exist a type $\texttt{T\_}\rho \in \mathcal{T}'$ such that $\texttt{T\_}\tau \preceq \texttt{T\_}\rho$ and $\texttt{T\_}\rho \notin \Diamond_{\texttt{T\_}\tau}$. □

From Axiom 3.2, all types are a subtype of the type `T_object`. Therefore, `T_object` must be in $\Diamond_{\texttt{T\_}\tau}$ for all types $\texttt{T\_}\tau$. Thus, $\Diamond_{\texttt{T\_}\tau}$ forms a *complete lattice* of types with $\texttt{T\_}\tau$ being the most defined element in $\Diamond_{\texttt{T\_}\tau}$ and the type `T_object` being the least defined one.

For example, applying the super lattice behavior to the map type `T_map` of Figure 2.2 (denoted as *B_super-lattice*(`T_map`)) would result in the set of types {`T_map`, `T_zone`, `T_window`, `T_displayObject`, `T_object`}, partially ordered by $\preceq$ which is represented graphically by the complete lattice shown in Figure 3.2.

By definition, any object of type $\texttt{T\_}\tau$ must support the behaviors of all types in $\Diamond_{\texttt{T\_}\tau}$. In other words, any behaviors which operate on objects of a type $\texttt{T\_}\sigma \in \Diamond_{\texttt{T\_}\tau}$ must operate on objects of type $\texttt{T\_}\tau$. Some have called this *substitutability* [SZ89] because an object of type $\texttt{T\_}\tau$ can be used (substituted) in any context specifying a supertype of $\texttt{T\_}\tau$. We refine the definition of *conformance* from [Str91] to describe this property. We first define a *conforms to* relation on the type `T_object` as follows:

Behavior 3.8 *Conforms-to (B_conformsTo* : `T_type` → `T_boolean`*)* ($\rightsquigarrow$)*:* Given an object o and a type $\texttt{T\_}\tau$, the behavioral application $(B\_conformsTo(o))(\texttt{T\_}\tau)$ (denoted $o \rightsquigarrow \texttt{T\_}\tau$) is `true` if and only if $(B\_mapsto(o)) \sqsubseteq \texttt{T\_}\tau$. The term $o \rightsquigarrow \texttt{T\_}\tau$ reads object o *conforms to* type $\texttt{T\_}\tau$. □

The truth of the statement $o \rightsquigarrow \texttt{T\_}\tau$ implies that all behaviors defined on type $\texttt{T\_}\tau$ are applicable to the object o. Given an object o which maps to type $\texttt{T\_}\tau$, o must conform to all types which $\texttt{T\_}\tau$ specializes. These include the types in $\Diamond_{\texttt{T\_}\tau}$ (since for all $\texttt{T\_}\sigma \in \Diamond_{\texttt{T\_}\tau}$, $\texttt{T\_}\tau \preceq \texttt{T\_}\sigma$ which implies $\texttt{T\_}\tau \sqsubseteq \texttt{T\_}\sigma$), plus all types that the types in $\Diamond_{\texttt{T\_}\tau}$ specialize. If we consider the possible groupings of these types, each grouping forms what we call a *conformance* for the object o. A conformance is formally defined as follows:

Definition 3.4 *Conformance* ($\approx$)*:* A *conformance* for an object o is a set of types Θ = {`T_1`, `T_2`,..., `T_n`} such that for all types $\texttt{T\_i} \in \Theta, o \rightsquigarrow \texttt{T\_i}$. We use the notation $o \approx \Theta$ to indicate that object o has *conformance* Θ. □

A *conformance* for a particular object gives a typed perspective of that object. The types in a conformance define behaviors which are applicable to the given object. It is possible that some of the

behaviors may be shared among the types in the conformance because of subtyping and specialize relationships which may exist among the types. It is also possible that not all behaviors applicable to the object are represented by the types in the conformance. An object has (possibly) many conformances, which translates directly into the statement that a type can specialize (possibly) many other types. However, for every object there exists a conformance such that adding a type to the conformance does not add any additional type information for the object, and deleting a type from the conformance would lose typing information. We call this conformance the *most specific conformance* for the object.

Definition 3.5 *Most Specific Conformance* ($MSC()$)*:* A conformance Θ for an object o is a *most specific conformance* if and only if there does not exist a type $\mathtt{T\_}\tau \in \mathcal{T}'$ such that $o \rightsquigarrow \mathtt{T\_}\tau$ and $\mathtt{T\_}\tau \sqsubseteq \mathtt{T\_}\sigma$ for some $\mathtt{T\_}\sigma \in \Theta$, where $\mathtt{T\_}\sigma \neq \mathtt{T\_}\tau$. We denote a most specific conformance for an object o as $MSC(o)$. □

The *most specific conformance* for a particular object o is the one and only set of types $MSC(o)$ that most specifically define the behaviors of o. Every object has one and only one *most specific conformance*. In general, for a given object o, the most specific conformance is a set consisting of the single type which the object o *maps to*. In previous work [SÖ90a] we have found that when an object o is a set, there is another form of MSC to consider which is important for typing the results of queries that are sets. This second form of MSC is useful for determining the set of types which most specifically define the common behaviors of the element objects in the set rather than the conformance of the set object itself.

Definition 3.6 *Most Specific Set Conformance* ($MSC_{set}()$)*:* The most specific set conformance for a set of objects O (denoted $MSC_{set}(O)$) is the one and only set of types Θ such that:

(1) $\forall o \in O,\ o \approx \Theta$, and
(2) $\{\ \nexists \mathtt{T\_}\tau \in \mathcal{T}' \mid \forall o \in O,\ o \rightsquigarrow \mathtt{T\_}\tau$ and $\mathtt{T\_}\tau \sqsubseteq \mathtt{T\_}\sigma$ for some $\mathtt{T\_}\sigma \in \Theta$ where $\mathtt{T\_}\sigma \neq \mathtt{T\_}\tau\}$

□

The first statement indicates that Θ is a conformance for every object in O. The second states that there is no type in the type lattice that more specifically defines the behavior of all objects in O other than the types given in Θ. For example, consider the type structure of Figure 3.3, and assume the existence of two objects o_1 and o_2 such that o_1 is in the extent of `T_1` and o_2 is in the extent of `T_2`. Because of subtyping, o_1 and o_2 are also in the extents of `T_3` and `T_4`. The $MSC(o_1)$ is {`T_1`} and the $MSC(o_2)$ is {`T_2`}. Using this schema, a query could generate and return the generic set object $\{o_1, o_2\}$. The $MSC(\{o_1, o_2\})$ could be given as the generic set type {`T_set`} because of the lack of further type information. In contrast, the $MSC_{set}(\{o_1, o_2\})$ is the set of types which most specifically define the behaviors of the elements of $\{o_1, o_2\}$ (i.e., objects o_1 and o_2 respectively). The result of this conformance is the set of types {`T_3`,`T_4`} because both o_1 and o_2 inherit the behaviors of `T_3` and `T_4` and there is no other type which more specifically defines both objects. The result could not have been {`T_1`} because o_2 does not conform to `T_1` and it could not have been {`T_2`} because o_1 does not conform to `T_2`; it also couldn't have been {`T_1`,`T_2`} for the same reason. Furthermore, {`T_3`} and {`T_4`} are also incorrect because in these cases some typing information is lost for the member objects; namely, behaviors *B_4* or *B_3* respectively.

$MSC_{set}()$ will be used extensively in the query model to perform type checking and type inferencing on the results of queries. The result of a query has the form of a set which may contain

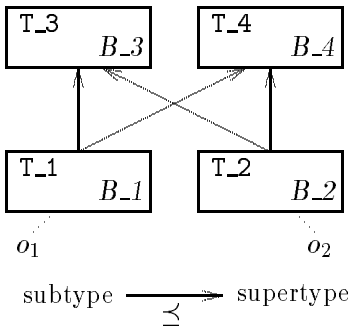


Figure 3.3: An example type schema.

objects of several heterogeneous types. We can use $MSC_{set}()$ on these query sets to determine the most typing information (i.e., behaviors) for the result objects and then can dynamically create a subtype which consistently incorporates these behaviors. The usefulness of $MSC_{set}()$ and an algorithm for determining the most specific set conformance for a set of objects is presented in [Str91].

A final behavior we require on types is for determining 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 the instances of that type. However, types do not require an associated class if there are no instances of that type. For example, 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 this case, there may be no reason to manage classes for abstract types, because there are no instances of these types. However, a class may be formed if there is a need to categorize the objects of the subtypes by a common class. Thus, the model enforces the one way implication: *class* $\Longrightarrow$ *type*. The definition of *B_classof* is given as follows:

Behavior 3.9 *Class of (B_classof :* `T_class`*)* ($\mathcal{C}$)*:* Given a type `T_`τ, the behavior *B_classof*(`T_`τ) (denoted as $\mathcal{C}_{\texttt{T\_}\tau}$) returns the class object (if it exists) **C_**τ associated with `T_`τ which manages the extent of type `T_`τ. □

For example, if we assume that a class **C_map** has been created and associated with type `T_map`, then the application *B_classof*(`T_map`) returns the class object **C_map**. The notation $\mathcal{C}_{\texttt{T\_map}}$ represents an object reference which is equivalent to the references **C_map** and *B_classof*(`T_map`) (i.e., $\mathcal{C}_{\texttt{T\_map}}$ = **C_map** = *B_classof*(`T_map`)).

In summary, types represent the fundamental building blocks of objects. A type is a specification of both structure and behavior which is imposed upon objects created using the type as a template. In this section we have concentrated on the behavioral aspects of objects. A type defines the full behavior of all objects which map to the type. Some of the primitive behaviors we defined on the type `T_type` include interface related behaviors (*B_native*, *B_inherited*, *B_interface*), *super-lattice* ($\diamond$) and *class-of* ($\mathcal{C}$) all of which are supported by type objects. To characterize the properties of types, we defined the *specialize* ($\sqsubseteq$), *subtype* ($\preceq$), and *conforms to* ($\rightsquigarrow$) relationships and introduced the notion of a *conformance* (Θ), the more restrictive definition of *most specific conformance* ($MSC()$) and the set related MSC denoted $MSC_{set}()$.

3.5 *Collections* and *Classes*

The support of efficient query processing and storage management requires mechanisms to group related objects so that they may be managed, referenced and processed collectively. The *collection* and *class* objects serve this purpose in TIGUKAT. The relative advantages and disadvantages of providing a system-managed class as the only grouping mechanism for the extent of a type versus supporting user defined and managed collections as clusters of instances has been debated [YO91, ÖSP92]. Beeri [Bee90] shows, at a structural level, that both can be supported. Our model defines both classes and collections for grouping objects.

A *collection* is a general grouping mechanism. The set of all objects managed by a collection is called its *extent*. It is common to equate the term "collection" with "extent." There are two ways in which objects are included in the extent of a collection; objects may be explicitly declared to be part of a collection's extent, or one may additionally specify a predicate on a collection which enforces automatic inclusion of appropriate objects into the extent. The objects in the extent of a collection all support a set of common behaviors; they must minimally support the behaviors of `T_object`. These common behaviors are defined by a type in the type lattice that is associated with the collection when it is created. Every collection knows the type which defines the common behaviors of the objects in its extent. The semantics of collection objects are given by behaviors defined on the primitive type `T_collection`. The following behavior returns the extent type which defines the common behaviors of the objects in a collection's extent. This extent type may be specified by the user or the system may automatically derive the type.

Behavior 3.10 *Type of (B_typeof :* `T_type`*)* (Λ)*:* Given a collection $\mathbf{L}\_\tau$, the behavioral application *B_typeof*($\mathbf{L}\_\tau$) (denoted $\Lambda_{\mathbf{L}\_\tau}$) returns the singleton type object which has been defined to be the type of the objects in the extent of $\mathbf{L}\_\tau$. □

The following behavior returns the entire extent of a collection. The result type of this behavior is a set of objects and the behaviors of the elements in the set are defined by the *B_typeof* type associated with the receiver collection object. In the following behavioral definitions we use the notation $\Lambda_{receiver}$ to denote the type resulting from applying the *B_typeof* behavior on a *receiver* collection or class object.

Behavior 3.11 *Extent (B_extent :* `T_set`$\langle\Lambda_{\texttt{receiver}}\rangle$*)* (*)*:* For a collection $\mathbf{L}\_\tau$, the behavioral application *B_extent*($\mathbf{L}\_\tau$) (denoted $\mathbf{L}\_\tau^*$) results in a set of objects explicitly declared to be part of $\mathbf{L}\_\tau$, together with the set of objects satisfying the inclusion predicate of $\mathbf{L}\_\tau$, if defined. This set of objects is called the *extent* of $\mathbf{L}\_\tau$. The *receiver* of the *B_extent* behavior (in this case $\mathbf{L}\_\tau$) determines the type of the member objects through the *B_typeof* behavior (in this case $\Lambda_{\mathbf{L}\_\tau}$), so the result type of *B_extent*($\mathbf{L}\_\tau$) will be `T_set`$\langle\Lambda_{\mathbf{L}\_\tau}\rangle$. □

Collections may be heterogeneous in the sense that their extent may contain objects which map to different types that are not in a subtype relationship. This feature is essential for proper handling of queries which may return objects of various types [SÖ90a]. A collection always has an associated type which specifies the behaviors supported by all objects in the extent of the collection. This may require the automatic derivation of new types (during projections and joins for example) in order to provide as much type information as possible for the objects in the collection. Dynamic schema evolution is used extensively by the object query model we define.

The specialized, better known, form of a collection is that of a *class*. We define the type `T_class` as a subtype of the type `T_collection`. Therefore, classes must support the *B_typeof* and *B_extent* behaviors defined on collections. We specialize (refine) these behavior for classes as shown below. Every class is required to be uniquely associated with a single type. This association occurs at class creation time and persists with the class throughout its lifetime. The *B_typeof* behavior for classes returns the type which has been associated with the receiver class object.

Behavior 3.12 *Type of (B_typeof* : `T_type`*)* (Λ)*:* Given a class $\mathbf{C\_}\tau$, the behavioral application *B_typeof*($\mathbf{C\_}\tau$) (denoted $\Lambda_{\mathbf{C\_}\tau}$) returns the singleton type object which has been explicitly associated with the class object $\mathbf{C\_}\tau$. *B_typeof* on classes is the inverse behavior of *B_classof* on types. □

The *B_extent* behavior for classes is separated into two forms. The first form is similar to the extent of a collection but is referred to as the *deep extent* in the context of a class. The second form results in a subset of the deep extent and is referred to as the *shallow extent*. Shallow and deep extents are well know concepts which have been discussed in other models [KC86, BCG+87, SÖ90a]. The result types of these behaviors are sets of objects whose members all support the behaviors defined on the *B_typeof* type of the receiver class object.

Behavior 3.13 *Deep Extent, (B_extent* : `T_set`$\langle\Lambda_{\texttt{receiver}}\rangle$*)* (*)*:* The *deep extent* of a class $\mathbf{C\_}\tau$ (written $\mathbf{C\_}\tau^\star$) is a set consisting of all objects o satisfying the property that if $B\_mapsto(o) \preceq \Lambda_{\mathbf{C\_}\tau}$ then $o \in \mathbf{C\_}\tau^\star$. □

Behavior 3.14 *Shallow Extent, (B_shallow* : `T_set`*)* (†)*:* The *shallow extent* of a class $\mathbf{C\_}\tau$ (written $\mathbf{C\_}\tau^\dagger$) is the subset of objects from $\mathbf{C\_}\tau^\star$ ($\mathbf{C\_}\tau^\dagger \subseteq \mathbf{C\_}\tau^\star$) satisfying the property that for all objects $o \in \mathbf{C\_}\tau^\star$, if $o \mapsto \Lambda_{\mathbf{C\_}\tau}$ then $o \in \mathbf{C\_}\tau^\dagger$. □

The shallow extent of a class includes only those objects created using the class's *B_typeof* type as a template. The deep extent includes the objects of the shallow extent unioned with the shallow extents of all subtypes of the class's *B_typeof* type. The shallow extents of classes are disjoint groupings of objects. That is, for all classes $\mathbf{C\_i}$, $\mathbf{C\_j}$, $\mathbf{C\_i}^\dagger \cap \mathbf{C\_j}^\dagger$ is empty when $i \neq j$. The definition of deep extent imposes a subset inclusion relationship on the *extents* of classes. We refer to this as *subclassing* which has a direct relationship to subtyping and is in keeping with the conformance properties on types.

Definition 3.7 *Subclass:* A class $\mathbf{C\_}\tau$ is a *subclass* of a class $\mathbf{C\_}\sigma$, meaning $\mathbf{C\_}\tau^\star \subseteq \mathbf{C\_}\sigma^\star$, if and only if $\Lambda_{\mathbf{C\_}\tau} \preceq \Lambda_{\mathbf{C\_}\sigma}$. We can equally say that $\mathbf{C\_}\sigma$ is the *superclass* of $\mathbf{C\_}\tau$. □

In our definitions, we separate a *type* from the declaration of its *class* and subsequent *collections*. This design issue is a controversial one. Many former model proposals bundled these two concepts calling it either a "type" or a "class" [GR85, LRV88, BBB+88, Str90]. In our model we take special care to separate the two notions and attach individual semantics to each one. We believe that a type is simply a specification mechanism whose semantics should remain as such. Types are used to describe the structure and behavior of objects which should be separated from the grouping of objects in order to provide flexibility in defining exact grouping semantics. In our model, classes

group the shallow and deep extents of types which has its basis on subtyping. In other models, this definition may vary. The introduction of collections supplement classes by providing a very general grouping mechanism which has a consistent semantics with the concept of a class. We feel the inclusion and separation of these notions provide greater modeling flexibility and expressibility than if they were bundled into a single concept.

The final behavior we define on the type `T_class` is that of object creation. All objects are created through a particular class using that class's *B_typeof* type as a template. This has the side effect of automatically placing the object in the shallow extent of the class which implies that it is in the deep extent as well.

Behavior 3.15 *New (B_new : $\Lambda_{receiver}$):* Given a class $\mathbf{C\_}\tau$, the application of the behavior *B_new*($\mathbf{C\_}\tau$) has the result of creating a new object o such that o is consistent, $o \mapsto \Lambda_{\mathbf{C\_}\tau}$ and $o \in \mathbf{C\_}\tau^{\dagger}$ (which implies $o \in \mathbf{C\_}\tau^{\star}$). The application *B_new*($\mathbf{C\_}\tau$) denotes an object reference to the newly created object o whose type is $\Lambda_{\mathbf{C\_}\tau}$ which is derived from the receiver class object $\mathbf{C\_}\tau$. □

The result type of *B_new* is refined for each class to reflect the type of that class (i.e., *B_typeof*). This ensures that objects created by *B_new* have the proper type. For example, the behavioral application *B_new*(**C_person**) creates a new object of type $\Lambda_{\mathbf{C\_person}}$ = `T_person` and places it in the extent of class **C_person**. The returned result of the application is an object reference to the newly created `T_person` object. The *B_new* behavior on classes gives our model the necessary ability to create new objects and to have them automatically placed into their respective class extents.

In summary, we define collections and classes for grouping objects with similar characteristics. Collections are a more general, user declared, mechanism for grouping while classes are more restrictive and automatic. We define the type `T_class` as a subtype of type `T_collection` which is intuitive since a class is a more specialized grouping mechanism than a collection. The behaviors *B_typeof* and *B_extent* are defined on the type `T_collection` and refined for its subtype `T_class`. The `T_class` subtype adds behavior *B_shallow* to return a restricted subset of its deep extent and behavior *B_new* for creating objects through a class.

3.6 Higher Level Constructs

Several of the primitives introduced in the previous sections are sometimes referred to as *meta-information* because they are objects which provide support for other objects. For example, the type `T_type` provides support for types by defining the structure and behaviors of type objects and the class **C_class** supports classes by managing class objects in the system. These meta-objects are objects themselves and therefore must be uniformly managed by means of the primitives. This is possible through the introduction of higher level constructs we call *meta-meta-objects*. Our model defines a three tiered structure for managing objects as depicted in Figure 3.4. Each box in the figure represents a class and the text within the box is the common reference name of that class. The arrows represent instance relationships with the head of the arrow being the instance and the tail being the class it belongs to.

The lowest level of our structure consists of the "normal" objects which depict real world entities such as *integers*, *persons*, *maps*, *behaviors* and so on, plus most of the primitive object system $\mathcal{O}$

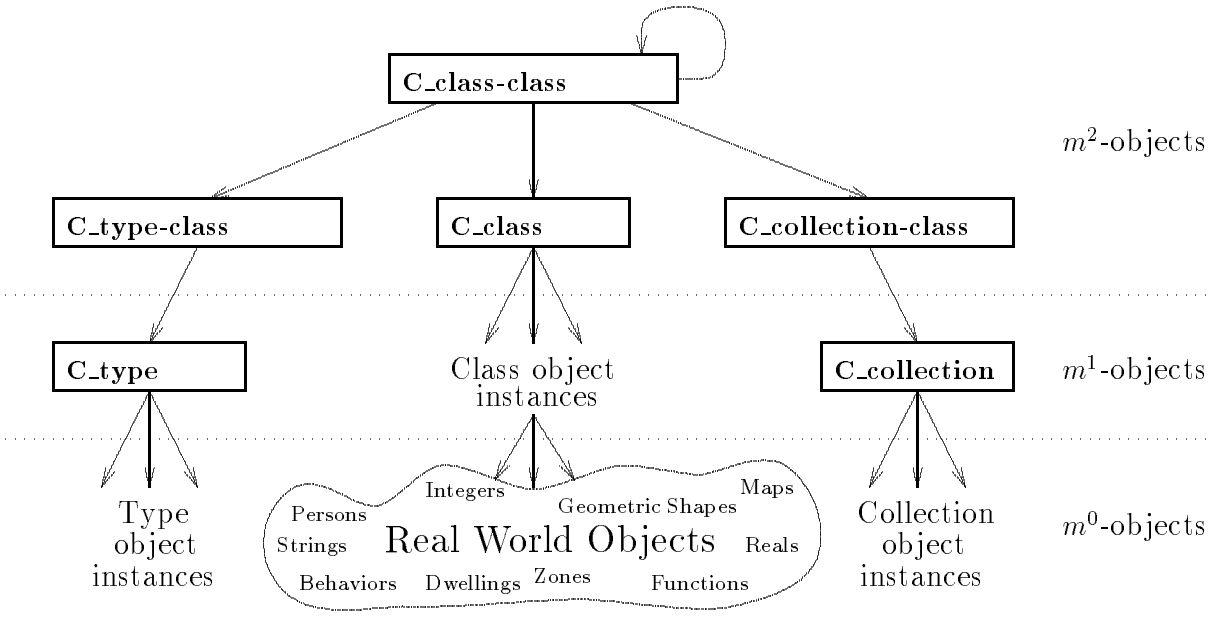


Figure 3.4: Three tiered instance structure of TIGUKAT objects.

resides at this level, including the type and collection objects 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 which 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 as 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, $object \Longrightarrow class$) and classes are associated with types which define the schema information of their instances (classes cannot exist without types, $class \Longrightarrow type$). Thus, $object \Longrightarrow class \Longrightarrow type$ and classes represent the binding management between objects and the operations that can be performed on them as defined by some 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 the 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. We refer the reader back to the primitive type lattice in Figure 2.1 on page 7 and a portion of its companion primitive subclass lattice shown in Figure 3.5. Each **C_x** class in Figure 3.5 is associated with the corresponding `T_x` type in Figure 2.1.

The model must have a way of consistently creating new types. The *B_new* behavior on the class **C_type** is inadequate for this purpose because it simply creates new empty objects and a type must always be created as a subtype of some other type(s); minimally a subtype of `T_object`. *B_new* cannot handle these semantics because it is a generic behavior for creating any kind of object and only new type objects need supertype information; it would be inappropriate to place these semantics on *B_new*. Therefore, we must somehow specialize the *B_new* behavior for types to allow for the addition of arguments that specify the supertype(s) of the new type along with other arguments such as its native behaviors. To accomplish this, we subtype the type `T_class` with a type `T_type-class` (see Figure 2.1) and define a behavior *B_newtype* on this type. Then, in the primitive class system, the class **C_type** is created as an instance of the class **C_type-class** (as

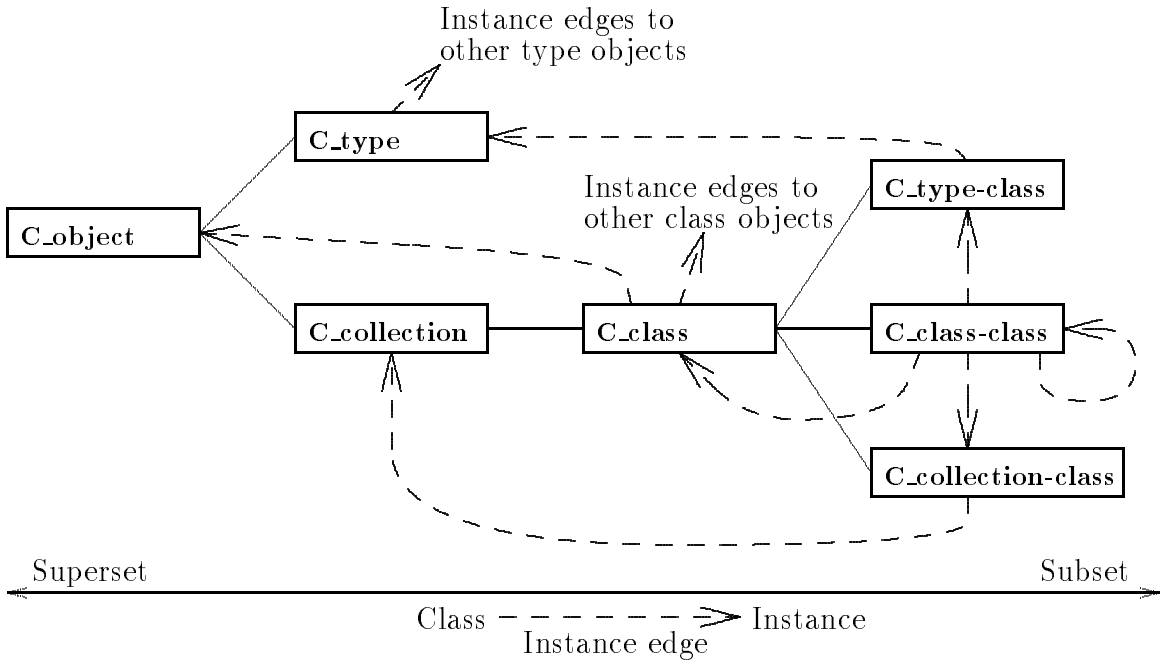


Figure 3.5: Subclass and instance structure of m^1 and m^2 objects.

shown in Figure 3.5) and new types are created by applying the *B_newtype* behavior to **C_type**. We again use the notation $\Lambda_{receiver}$ to refer to the type that the receiver class is associated with.

Behavior 3.16 *New Type (B_newtype* : `T_set`$\langle$`T_type`$\rangle \rightarrow$ `T_set`$\langle$`T_behavior`$\rangle \rightarrow \Lambda_{receiver}$*):* Given the class **C_type** which is an instance of type `T_type-class`, a set of types s such that `T_`$\tau \preceq$ `T_object` for some `T_`$\tau \in s$, and a set of behaviors b, the behavioral application *B_newtype*(**C_type**)(s)(b) has the result of creating a new type object o such that o is a subtype of all types in s and the behaviors of b are incorporated into the interface of o as native behaviors. □

A class must be associated with a type in order to create objects of that type. Classes must be associated with only a single type and no class may exist without an associated type. In order to consistently support this, we subtype the type `T_class` with the type `T_class-class` (see Figure 2.1) and define its native behavior *B_newclass* for creating and associating new classes with a type. The m^2-class **C_class** is structured as an instance of **C_class-class** (see Figure 3.5) and new classes are created by applying *B_newclass* to **C_class**.

Behavior 3.17 *New Class (B_newclass* : `T_type` $\rightarrow \Lambda_{receiver}$*):* Given class **C_class** and type `T_`σ, the behavioral application *B_newclass*(**C_class**)(`T_`σ) has the result of creating a new class object **C_**σ such that **C_**σ is in the shallow extent of **C_class** and **C_**σ is associated with type `T_`σ. If type `T_`σ does not exist, or is already associated with some other class, an error condition is raised because a type may be associated with at most one class. □

In the same way as types are associated with classes, types are also associated with collections; but a type may be associated with any number of collections. Therefore, we define the type `T_collection-class` as a subtype of `T_class` (see Figure 2.1) and provide the native behavior

B_newcollection for creating new collections which is similar to the behavior for creating new classes. The class **C_collection** is created as an instance of **C_collection-class** (see Figure 3.5) and new collections are created by applying *B_newcollection* to **C_collection**.

Behavior 3.18 *New Collection (B_newcollection* : `T_type` $\rightarrow \Lambda_{receiver}$*):* Given class **C_collection** and type `T_`σ, the behavioral application *B_newcollection*(**C_collection**)(`T_`σ) has the result of creating a new collection object **L_**σ such that **L_**σ is in the shallow extent of **C_collection** and **L_**σ is associated with type `T_`σ. The type `T_`σ may be omitted in which case the associated type of the collection is maintained by the system and derived according to the members in the extent of the collection. If type `T_`σ is given and does not exist, an error condition is raised. Types may be associated with any number of collections. □

The introduction of the m^2-objects complicates the type lattice and instance structures. However, the benefit of this approach is that the entire model is now consistently and uniformly defined within itself. Figure 3.5 shows the subset inclusion and instance structure of some of the m^0, m^1 and m^2-objects in relation to one another.

The class **C_object** is an m^1-object which maintains all the objects in the system (i.e., every object is in the deep extent of class **C_object**). Thus, **C_object** represents the entire database (i.e., the deep extent of **C_object** is a *coset*). Two other m^1-objects, which are subclasses of **C_object**, are **C_type** and **C_collection**. These two classes maintain all the instances of types and collections respectively. Class **C_collection** is further subclassed by the m^2-object **C_class** because every class object is also a collection of objects. The class **C_class** maintains the instances of all classes in the system. Furthermore, **C_class** is subclassed by m^2-objects **C_type-class**, **C_class-class** and **C_collection-class**. These three objects are class objects which maintain instances of other collection style m^1-objects and allow for these m^1-objects to be specialized with refined object management behaviors.

The class **C_type-class** has the single instance object **C_type**. Any behaviors defined on type `T_type-class` are applicable to object **C_type**. The type `T_type-class` defines the behavior *B_newtype* for creating new type objects. Since **C_type** is an instance of `T_type-class`, the *B_newtype* behavior is applicable to it and has the result of creating a new type object as an instance of **C_type**, thereby adding it to the type lattice.

The class **C_collection-class** has the single instance object **C_collection**. Any behaviors defined by `T_collection-class` are applicable to **C_collection**. The type `T_collection-class` defines the behavior *B_newcollection* for creating new collection objects. Since **C_collection** is an instance of `T_collection-class`, the *B_newcollection* behavior is applicable to it and has the result of creating a new collection object as an instance of **C_collection**.

The class **C_class-class** maintains all the m^2-classes. Its instances include itself, **C_type-class**, **C_collection-class** and **C_class**; because **C_class** maintains instances of other classes. The type `T_class-class` defines the behavior *B_newclass* for creating new instances of this m^2-class or one of its subordinate m^2-classes. This can be useful for defining a m^2-class which includes behaviors for creating default objects of a particular type. For example, consider the GIS database of Section 2.1 and assume that type `T_person` and class **C_person** are defined. The only way to create a `T_person` object is to apply *B_new* to class **C_person** and this has the effect of creating a new empty person object. Now, let's say we require a behavior called *B_defaultPerson* for creating new `T_person` objects with some of the `T_person` behaviors of the new object receiving default information. We

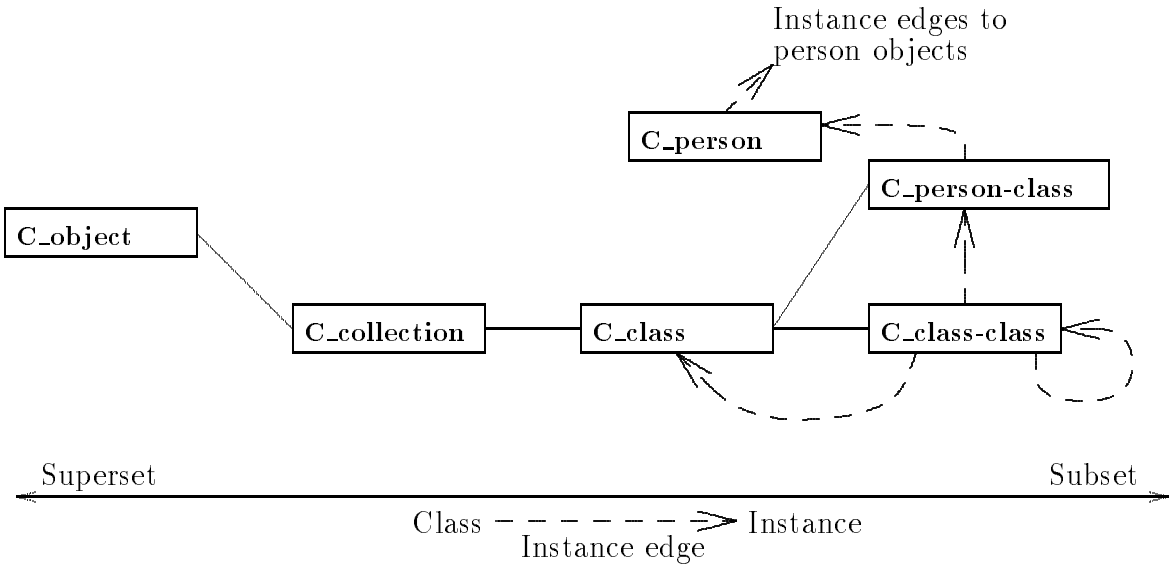


Figure 3.6: Instance structure of default person creation example.

cannot define this behavior on the type `T_class` because then it would be applicable to all classes and we only want it to apply to person objects. The solution lies in the m^2-objects.

Returning back to the GIS example database, let us assume that `T_person` is defined as a subtype of `T_object` and behavior *B_defaultPerson* is defined as an instance of type `T_behavior`. We first create a new type called `T_person-class` as a subtype of `T_class` which will incorporate *B_defaultPerson* as a native behavior of its interface. The following behavioral application does this (the $\leftarrow$ represents assignment):

$$\texttt{T\_person-class} \leftarrow \textit{B\_newtype}(\textbf{C\_type})(\{\texttt{T\_class}\})(\{\textit{B\_defaultPerson}\})$$

Following this, we need to define an implementation of *B_defaultPerson* and associate it with the type `T_person-class`. We'll assume that this is done. Next, we create a companion m^2-class for `T_person-class` which we call **C_person-class** as follows:

$$\textbf{C\_person-class} \leftarrow \textit{B\_newclass}(\textbf{C\_class-class})(\texttt{T\_person-class})$$

Now, it is semantically consistent for all instances of **C_person-class** to have the *B_defaultPerson* behavior applied to them. Thus, the final step is to create a class **C_person** as an instance of **C_person-class** and associate it with the type `T_person`:

$$\textbf{C\_person} \leftarrow \textit{B\_newclass}(\textbf{C\_person-class})(\texttt{T\_person})$$

This series of behavioral applications results in the subclass and instance relationships shown in Figure 3.6. Now, the class **C_person** is in the deep extent of **C_class** because **C_person** is an instance of **C_person-class** and **C_person-class** is a subclass of **C_class**. Therefore, the behavior *B_new* can be applied to **C_person** to create new empty person objects (i.e., *B_new*(**C_person**) creates a new empty person). Furthermore, **C_person** is also in the extent of **C_person-class** and thus the *B_defaultPerson* behavior may be applied to it to create a new person with defaults

as defined (i.e., *B_defaultPerson*(**C_person**) creates a new person with defaults as dictated by the particular implementation). This example illustrates the very powerful type definition and structuring facilities which are available from the primitives of the TIGUKAT object model.

3.7 The *Null* Primitive

Nulls in our model have a simplistic semantics. The model defines a primitive type `T_null` along with its corresponding class **C_null**. This class is defined to have a single primitive instance object reference called null. The type `T_null` is defined to be the subtype of all other types in the system. This gives it the exact opposite semantics of the type `T_object` which is defined to be the supertype of all other types. We say that type `T_null` *lifts* the domain of types and creates a *complete lattice* of types. We define a companion axiom for the axiom of root type (Axiom 3.2) which describes the type constraint of the null type.

Axiom 3.3 *Null Type:* for all types $\texttt{T\_}\tau$, $\texttt{T\_null} \preceq \texttt{T\_}\tau$. □

As a subtype of all other types, the `T_null` type refines the implementations of all application specific inherited behaviors (i.e., all behaviors except those of the primitive type system) in such a way that applying a given behavior to its only instance object null, always returns the instance object null. In this way, *nulls* represent a fix-point for non-primitive behavioral application over the domain of objects. It is always safe to allow a function to return the object null because null will always conform to all the types in the type lattice and therefore must conform to the result type in the inherited signature. Nulls can be used as results of functions when more meaningful results are not known.

For example, `T_null` is a subtype of the type `T_person` in the GIS example type lattice of Figure 2.2 which is an application specific type. Therefore, `T_null` refines the behaviors of `T_person` to return the object null when applied to its instance object null. Now, if for a specific instance of `T_person`, say sherry, the result of a certain behavior, say *B_age*, is not known, we can assign it the object null. Then, the application *B_age*(sherry) returns the object null, and all subsequent behavioral applications (except for those of the primitive type system) also return null.

3.8 *Database* Definition

With the modeling primitives established, we now define an object database.

Definition 3.8 *Database* ($\mathcal{D}$)*:* An object database $\mathcal{D}$ is a consistent set of objects (*coset*) such that:

1. $\mathcal{O} \subseteq \mathcal{D}$.

   The elements of the *primitive object system* $\mathcal{O}$ (which is a *coset*, Section 3.3) are part of $\mathcal{D}$.

2. for all objects $o \in \mathcal{D}$, for all behaviors $B\_i \in \mathcal{D}$, $B\_i(o) \in \mathcal{D}$ and is consistent.

> For all general objects and behavior objects in $\mathcal{D}$, applying a behavior from $\mathcal{D}$ to an object in $\mathcal{D}$ generates an object which is also in $\mathcal{D}$ and keeps $\mathcal{D}$ consistent.

A database defines a restricted enclosure of objects which facilitates a consistent, systematic investigation of database features such as query processing, views and transaction management. A database does not define the relationships of its consistent object set with external objects outside the domain of the database. For now, these relationships should be considered ill-defined and inconsistent, although may prove useful in a distributed environments.

Chapter 4

The Structural Model

Beeri's work on formal structural object models [Bee90] has been chosen as a foundation for our structural model definition. This chapter follows Beeri's framework in defining a structural model and shows its integration with the formal behavioral definition of TIGUKAT.

4.1 Objects and Values

Our model considers an object-oriented database (OODB) to be a collection of objects. Each object, in order to exist, must be associated with at least one reference which gives access to the object in the database. Thus, every object has the universal perception of a reference and our model has a single uniform representation for objects. In this way, the model resembles the general naming facility of O_2 [LR89b] or the "Name" operation of [Osb88] which allow names (references) to be attached to individual objects, but our model applies a more uniform semantics to these features by servicing all access to objects through references.

Beeri makes a strong case in distinguishing between the notions of "object" and "value" at the structural level. However, he does point out that in the general intuitive sense, objects and values should have the universal perception of objects. The latter perspective is defined by the TIGUKAT object model presented in Chapter 3. The structural model presented here introduces a separation of these two notions because there is an inherently different representation and semantics for values at this lower level. These differences need to be resolved eventually, and the structural model seems to be the appropriate place for this.

Beeri outlines several arguments which support the distinction of "values" from "objects." The reasons which seem most influential in this separation are:

1. the perception that values represent universally known abstractions (such as the integers), while objects denote application specific abstractions,
2. the notion that values are built into the system and are assumed to exist, while objects need to be defined and introduced into the system,
3. the information carried by a value is itself and is immutable, while an object consists of a separate mutable state that represents the information carried by the object.

Using these distinctions, the following definition of a value is formed. We qualify these as *atomic values* because they are formed from the atomic types and they are immutable. Atomic values are entirely under the management of the system.

Definition 4.1 *Atomic Value:* An *atomic value* is any object from the domains of the atomic types. Atomic values are predefined by the atomic types and are assumed to exist. Atomic values are immutable.

Each atomic type has a standard representation for references to the atomic values of their respective domains. These references are also assumed to exist and are maintained by the system. The form of these standard references is purely syntactic and one interpretation is discussed in Section 3.1. Since these references are system maintained, they will never be released and will persist throughout the lifetime of the database, thereby making them immutable.

Recall the definition of an object as an (*identity, state*) pair (Section 3.3). For atomic values, the value itself serves as identity and state all at once. This property is what makes values immutable to change. The distinguishing factor between objects and values seems to be that objects have an immutable identity separate from a mutable state, while values represent identity and state all at once, both of which are immutable. Beeri makes the distinction that values are used to describe other things, while objects are the things being described. From a mathematical perspective, one may consider values to be elements of the *built-in* domains, while objects are elements of the *uninterpreted* domains.

4.2 Structured Values

The behavioral specification of our model defines *set, bag, poset* and *list* types for developing structured container objects. The objects which form the instances of these structured types have the semantics of values and are qualified as *structured values*. These are equivalent to the set structured values defined by Beeri. These structures may be subtyped to customize their semantics. One example is the use of parameterization to define containers whose elements are restricted to a particular type. Beeri also defines tuple structured values which we do not explicitly include in our model. We have cast the notion of tuple into the uniform concept of behaviors on types. A tuple in our model is just a type definition with the behaviors representing the named slots (or attributes) of the tuple.

The contents of a set object uniquely identifies itself. This is in keeping with the mathematical notion of set (i.e., two sets with the same members are the same set.). Sets are *structured values* which carry their own information in the same way as atomic values do. Thus, a set value defines both its identity and state all at once and is therefore immutable. The act of updating the contents of a set does not change the set object itself, but rather is a mapping from one set to another. For example, consider the signature B_zones : `T_set`⟨`T_zone`⟩ defined on the type `T_map`. For an object o of type `T_map`, $B\_zones(o)$ returns a set whose members are references to `T_zone` objects which represent the zones contained within the map object o. Adding a zone to this set does not change the original set of references, but rather modifies the mapping of B_zones for object o from the previous set to reference the new set which reflects the addition of the new zone. These properties are inherited by the *bag, poset* and *list* primitive types.

Structured values are recursive in the sense that multi-leveled structured values (i.e., sets of sets) may be formed. These nested structures are considered to be structural values and updates to them require the modification of a function mapping to a new nested structural value which reflects the changes.

4.3 Abstract Objects

An *abstract object* is defined as an object which has the semantics of an immutable identity separate from a mutable state. Application specific objects and the primitive *type, class, collection* and *function* objects all fit into this category.

For a given abstract object, the values of its behaviors are given as signature specifications with the result type of each signature replaced by the actual resulting object for that signature. For example, we could specify the name behavior for an object o of type `T_person` as *B_name*: "joe", or if the object context was not explicit, this could be qualified as *B_name*(o): "joe".

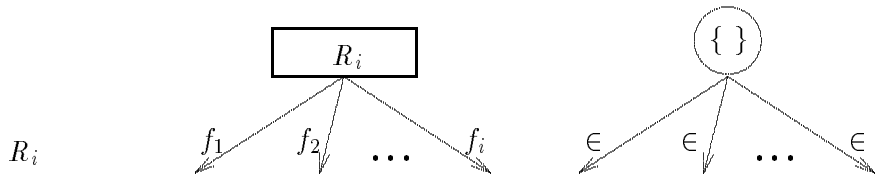
Beeri uses the semantics of atomic values in the treatment of *abstract objects*, meaning that an abstract object is also immutable in a sense. It is true that abstract objects incorporate a state which may change over time. However, modifying the state does not change the object as far as its existence in relation to other objects is concerned. For example, given two objects o_1 and o_2 where $o_1 \neq o_2$, no matter how the state of any of these two objects is modified, the object o_1 will never be identity equal to the object o_2. They are two unique objects within the system and will remain that way throughout their lifetime. In this respect, abstract objects are also atomic in the structural model. From a mathematical perspective, attributing abstract object with atomic properties is very useful since it allows first order semantics to be applied to them. This will be useful when defining a query language for the model.

In the context of our model, there is a commonality between values and objects which captures their atomicity. When talking about atomic values, structured values or abstract objects, we are essentially referring to the identities of objects which are separate from their state. The difference between values and abstract objects is that the state of the former is immutable while the latter has a state which may change over time.

4.4 Object Graph

An object database can be structurally represented as a directed graph. The nodes of the graph represent the atomic forms of objects: atomic values, structured values and abstract objects. Directed edges between nodes illustrate relationships (defined as behaviors) from one object to another.

A graph representation is important in several respects. First, it allows for a pictorial representation of the attributes and relationships of objects. This can assist in clarifying the contents and structure of a database. Second, a graph representation has the advantage that graph theoretic algorithms and proofs may be applied to extract and derive properties of the graph. There are many examples of graph related applications that can assist in solving query processing [Yan90] and object data management problems such as type inferencing, optimization strategies for object distribution and dynamic schema evolution.



(a) Atomic value. (b) Abstract object. (c) Set structured value.

Figure 4.1: Graphical representations of nodes in an object graph.

Our graph representation defines several kinds of nodes that may be used in an object graph. Figure 4.1 illustrates the graphical representation of these nodes and the semantics of each is defined as follows:

4.1 (a) Atomic value nodes consist of a label which represents a standard reference defining their value. Atomic values are terminal nodes of the graph which cannot have any outgoing edges.

4.1 (b) Abstract objects consist of a box labeled with an explicit reference for identifying the object. This label can be thought of as a structural model reference and has no implications of the other scope specific object references that may exist. Abstract objects have an outgoing edge for each behavior applicable to the object which is labeled with the name of the behavior and which leads to a node resulting from the application of the behavior to the given abstract object.

4.1 (c) Set structured values consist of a node labeled with the symbols { } which has outgoing edges labeled with "$\in$" to each member object of the set. The only differences between sets and its subtypes are that bags define a duplication relation for the elements and posets define an oredering relationship between pairs of elements.

As with Beeri's model, each object occurs only once in the graph, meaning each node represents a unique immutable object in terms of its existence. The nodes of the graph can be thought of as the object identities of the database and the edges leading to them can be thought of as object references. Objects and values (nodes) can be shared by having multiple edges leading to them.

4.5 Structural Example

Consider the object definitions of Figure 4.2. Each box represents a separate abstract object where the header specifies a reference for the object along with the *maps to* type for that object. Following this, the behaviors for each object is listed and their associated values are given.

Figure 4.3 illustrates an object graph for the geographic objects SCounty, Notingham and Forest3 of Figure 4.2. The map object SCounty is an abstract object with several outgoing behavioral edges as shown. The *B_proximity* behavior is not defined for the object and therefore points to the abstract object null. The behaviors *B_resolution*, *B_orientation* and *B_title* point to the atomic valued objects 0.5, 0 and "Sherwood County" respectively. The *B_region* behavior points to a `T_geometricShape` object which defines the geometric structure of the SCounty object. The *B_origin* behavior points to the `T_location` object loc_0 which has *B_latitude* and *B_longitude* behaviors to the appropriate

Person4 ↦ T_person	
B_name:	"Robin Hood"
B_birthDate:	null
B_age:	null
B_residence:	SForest
B_spouse:	Person7
B_children:	*B_children*(Person4)
B_children(Person7):	{Person9}

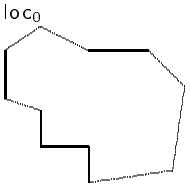
Person9 ↦ T_person	
B_name:	"Robin Jr."
B_birthDate:	null
B_age:	null
B_residence:	SForest
B_spouse:	null
B_children:	null

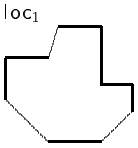
Person7 ↦ T_person	
B_name:	"Maid Marion"
B_birthDate:	null
B_age:	null
B_residence:	SForest
B_spouse:	Person4
B_children:	*B_children*(Person7)
B_children(Person4):	{Person9}

Person1 ↦ T_person	
B_name:	"Sheriff of Notingham"
B_birthDate:	null
B_age:	null
B_residence:	NCastle
B_spouse:	null
B_children:	null

SForest ↦ T_dwelling	
B_address:	"Top Secret"
B_inZone:	Forest3

NCastle ↦ T_house	
B_address:	"#1 Main Notingham Road"
B_inZone:	Notingham
B_mortgage:	0.00

Forest3 ↦ T_forest	
B_title:	"Sherwood Forest"
B_origin:	loc_0
B_region:	
B_proximity:	*B_proximity*(Forest3)
B_proximity(Notingham):	25.34

Notingham ↦ T_developed	
B_title:	"City of Notingham"
B_origin:	loc_1
B_region:	
B_proximity:	*B_proximity*(Notingham)
B_proximity(Forest3):	25.34

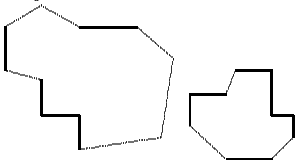
SCounty ↦ T_map	
B_title:	"Sherwood County"
B_origin:	loc_0
B_region:	
B_proximity:	null
B_resolution:	0.5
B_orientation:	0
B_zones:	{Forest3,Notingham}

Figure 4.2: Objects of Sherwood County.

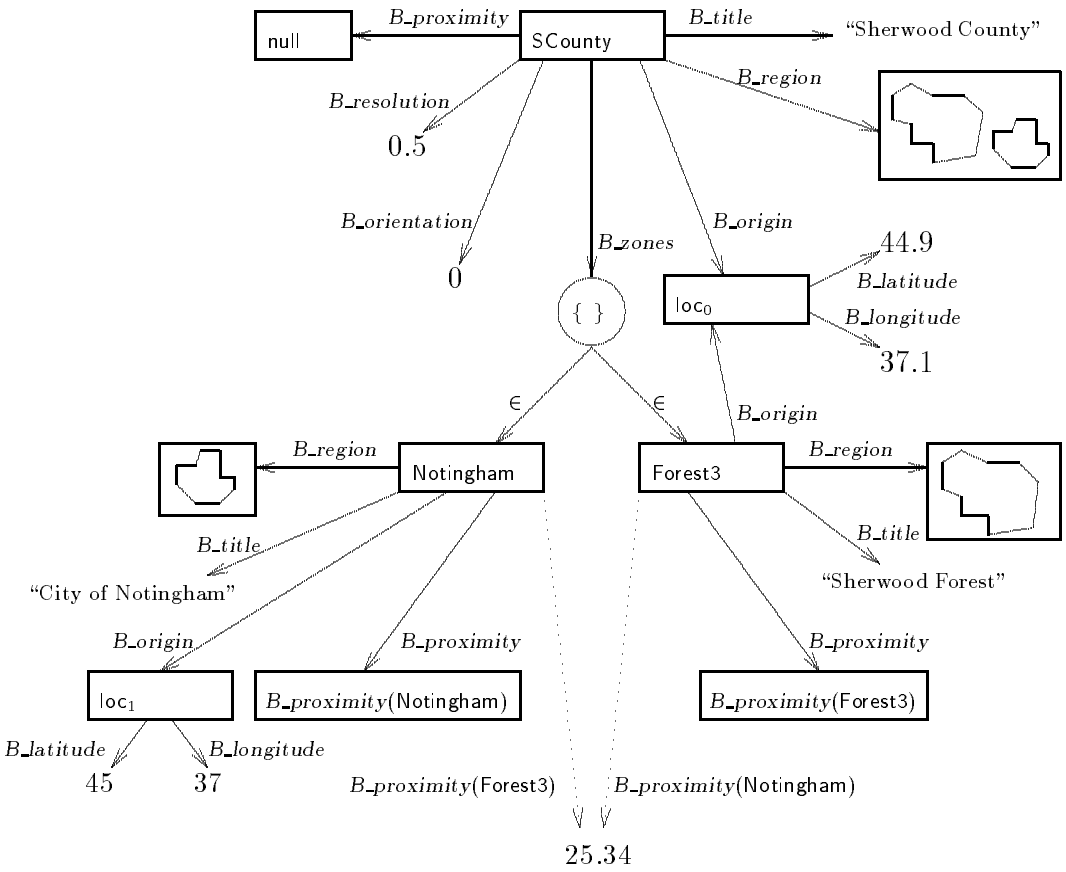


Figure 4.3: Object graph of SCounty, Notingham and Forest3 objects in Figure 4.2.

atomic valued objects 44.9 and 37.1 respectively. Finally the *B_zones* behavior points to a set structured value which contains the two `T_zone` element abstract objects Notingham and Forest3.

There are a few anomalies to note for the zone objects Notingham and Forest3. First, the *B_origin* behavior for Forest3 and SCounty share the same `T_location` object loc_0 which is indicated by its two incoming edges. Second, the *B_proximity* behaviors for the two zone objects are defined and point to function abstract objects which, when given another zone object as an argument, produce the desired distance measurement representing the proximity of the argument zone to the zone on which the function is defined. For example, *B_proximity* applied to Forest3 results in the function abstract object *B_proximity*(Forest3). This abstraction can be maintained by returning the implementation function object associated with *B_proximity* with the first argument fixed to Forest3. The graph further indicates that an invocation of this function when passed the argument zone Notingham will produce the atomic valued object 25.34. This function execution is represented by the dotted line attached to Notingham in Figure 4.3. A similar application is shown on Forest3 for the *B_proximity* behavior of Notingham which shares the same object as the previous execution.

The function executions shown by the dotted lines are not behavioral applications defined on the type `T_zone`, although they could be. Instead, they represent the result of executing a function defined object and are included in this example to illustrate the power and flexibility that the functional approach provides.

4.6 Schema Objects

Our model differs from Beeri's in that he makes a clear separation between the data of a database and its schema whereas we carry through the uniformity aspects of the behavioral model into the structural model. This means that schema objects are represented using the same graphical structures as other objects and may be integrated into a single object graph representing all information. In this way, the schema objects become part of the database which allows all database operations to be performed on them in a consistent manner. The schema is distinguishable from the application specific data, but in our model one could define applications that operate on the schema data transparently.

The uniformity of the schema is illustrated in the structural model by means of object graph links (relations) between objects. From the definition of type `T_object`, all objects inherit a *B_mapsto* outgoing edge to the type object which represents the declared type of that object. Furthermore, all objects support the equality behavior between all other objects although this behavior is specialized for some of the subtypes. Finally, all objects have a *B_conformsTo* edge to a function which, when executed with a type object argument, results in a true or false object depending on whether or not the object conforms to the type object argument.

Objects of type `T_type` have *B_native*, *B_inherited* and *B_interface* behavior edges pointing to sets of behaviors representing their various interface components that define the functionality of the type's instances. There is a *B_super-lattice* edge to a partially ordered set containing the union of all supertypes for that object up to and including the root type `T_object`. A type has a *B_classof* edge which points to the class object that maintains the instance objects of the type. Finally, there are *B_subtype* and *B_specialize* edges to function objects which, when executed with another type object argument, result in a true or false object depending on whether or not the original type is in the given relationship with the second argument type.

Collection objects have a *B_typeof* edge to the type object which is the declared type of the objects in the collection. Collections also have an *B_extent* edge to a set structured value whose member edges point to the instance objects in that collection.

A class object has the same outgoing edges as collections do, plus an extra edge for its shallow extent behavior (*B_shallow*) to a set node which has an $\in$ edge to each object that is in the shallow extent of the class. Recall that the shallow extent set is a subset of the deep extent (*B_extent*) for classes. Finally, there is an edge for the *B_new* behavior to the last newly created object of the appropriate type. The side effect of applying *B_new* is to update itself to create a new object and add the object to the class extents of the argument class (i.e., the one at the tail of the edge).

Putting all these components together results in a fairly complex directed graph with cycles. The advantage of this approach is that the schema has become part of the object graph. This means that a query model based on the graph can query the schema objects in a uniform manner. Furthermore, any graph-theoretic proofs or algorithms applicable to the object graph in general may be consistently applied to the schema objects as well.

For example, consider the partial schema representation of the type `T_zone` as an object graph shown in Figure 4.4. The `T_zone` object indicates a *B_mapsto* behavior to the type object `T_type` of which it is also an instance. There is a *B_classof* edge to the class **C_zone** which maintains the instances of `T_zone`. The *B_conformsTo*, *B_subtype* and *B_specialize* behaviors result in functions that are applied to other `T_type` objects and determine the truth or falsity of the relationship.

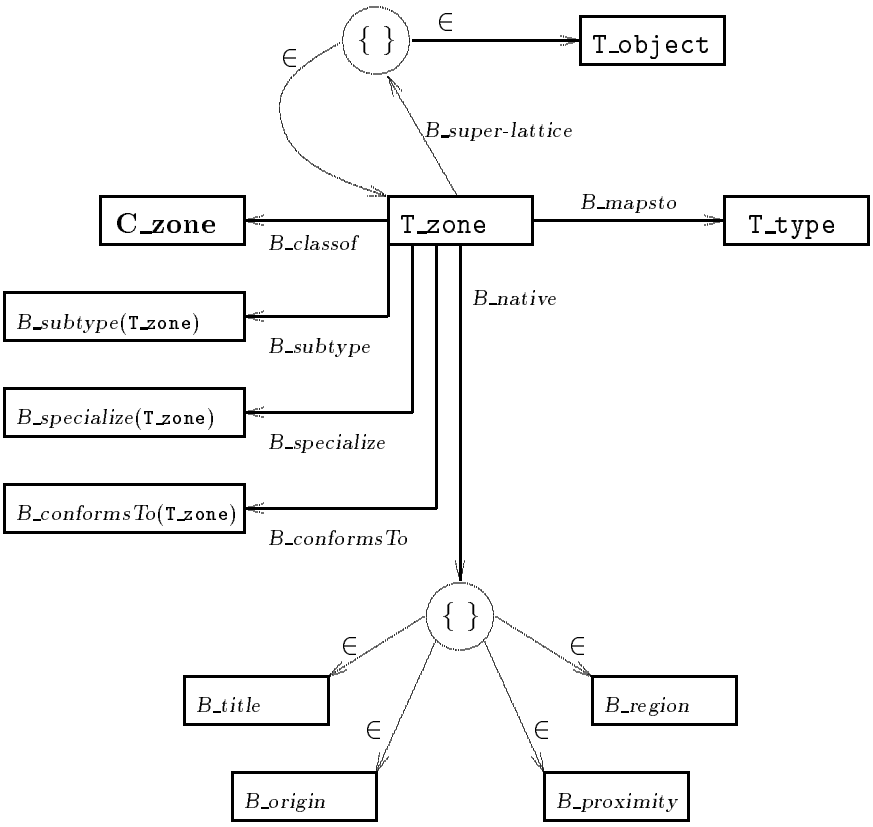


Figure 4.4: Object graph of partial schema for type `T_zone`.

There is a *B_super-lattice* set which has element edges to the two supertypes of `T_zone` (one of which is itself). Finally, the *B_native* set of behaviors for `T_zone` is shown to contain four functions which are defined locally by `T_zone`. The sets for *B_inherited* and *B_interface* are not shown. The set for *B_inherited* would contain behaviors *B_mapsto*, *B_equality* and *B_conformsTo* which are inherited from `T_object` and *B_interface* would simply be the union of these two sets.

Due to the complexity of these graphs, many of the relationships are not shown. However, we hope that through the previous examples the reader has a flavor of how these links are managed and realizes the inherent uniformity in the definitions we provide.

Chapter 5

Comparison with Previous Work

In this chapter we survey some of the earlier work on object modeling. We first examine the concepts presented in these models and indicate how they have influenced the design of TIGUKAT. We follow this investigation with a brief discussion of TIGUKAT's conformance with the guidelines outlined in the two manifesto papers [ABD+89] and [SRL+90]. Futhermore, we consider TIGUKAT's compliance with the recommendations given in [FKMT91]. These references are slightly outdated in terms of current object technology, but nevertheless they contain many of the core concepts important to the development of an object model.

5.1 Related Work

Codd's landmark paper in 1970 [Cod70] defined the relational model which provided a simple, but powerful, method of organizing data. The main advantages of this approach are that it offers a high degree of *data independence*, *data consistency* and language facilities based on the *first-order predicate calculus*. The success of the relational model can be partially attributed to its precise formal specification which facilitates a systematic investigation of database management system (DBMS) functions such as query processing, views and transaction management. However, it is well recognized that the flat record based representation of the relational model results in a semantic mismatch between the entities being modeled and the underlying DBMS [Ken79].

Several approaches have been followed to incorporate more meaning into a data model. One approach proposes modifications to the relational model in order to supply it with more power [Cod79]. Others have extended the relational model with data abstraction by including semantics for specifying user defined types [OH86, Sto88, WSSH88]. Some prototype systems employing this approach include STARBURST [Haa90] and POSTGRES [SR86, RS87, SRH90, SK91]. Another approach allows for non-first normal form relations which facilitates the modeling of nested relations [OY87, RK87, SS86, DKA+86]. This extension takes the language features outside the domain of first-order predicate calculus, thus higher-order languages for these nested relational models have also been developed [AB84, JS82, Sch85]. Some more recent relational model extensions have carefully incorporated properties of the object-oriented paradigm (discussed below) designating them relational object models [RK89, SS90].

An orthogonal approach to relational model extensions has been to develop a completely new data model with advanced modeling power and expressibility. One class of such models are the

semantic data models whose key features are based on the abstraction mechanisms of *classification*, *aggregation* and *generalization* [SS77] which allow for complex information to be categorized and accessed in meaningful ways. The pioneering models which fall into this category are the Entity-Relationship (ER) model [Che76] and SDM [HM78, HM81]. An overview of the entire field can be found in [HK87, PM88].

Some particular semantic data models with which TIGUKAT has similarities include:

- The functional data model and the data language DAPLEX [Shi81] which defines *entities* and *functions* as primitive modeling constructs. In DAPLEX, properties of entities and the relationships among them are all modeled as functions. This places the computational power of functional languages on properties and relationships in a uniform manner which facilitates a greater semantic expression of them; TIGUKAT adopts this uniform functional approach.
- SIM [JGF+88] is a commercially available DBMS based on the semantic data model SDM. Entities are defined in terms of simple *data-valued* attributes and more complex *entity-valued* attributes which represent a binary relationship between two classes of entities. Entities are organized into meaningful collections called *classes* each of which is either a *base class* (a class which is defined independently of other classes) or a *subclass* (a class which is defined in terms of other classes). This gives an inheritance hierarchy for entity classes. TIGUKAT separates the notions of type and class and extends the basic notion of class by supplementing classes with heterogeneous user-defined collections.
- The IFO data model [AH84], which formalizes the characteristics of semantic data models, was developed to serve as a theoretical foundation for investigations of advanced data representation. Our approach in defining TIGUKAT is proposed to establish a similar foundation for our investigations of object-orientation.

Object-oriented models were developed to further enhance the expressiveness and abstractions of semantic data models. Despite the wide variety of object-oriented models which have been proposed, no universally agreed upon definition exists. The reason for this being that object-oriented models have followed the same informal route of development as semantic models. However, there currently are several efforts to standardize the features of object-orientation. Two recent manifestos [ABD+89, SRL+90] have proposed various features of next generation DBMSs. A side-effect of these manifestos is to outline some object-oriented concepts which have sifted through the various model proposals over the years. In addition to these, Zdonik and Maier [ZM90] define a reference model which specifies the common features that should exist in an OODBMS, Wegner [Weg90] examines the goals, concepts and paradigms of object-oriented technology in the forum of object-oriented programming, Bancilhon and Kim [BK90, Kim90b, Kim90a] discuss the issues which will be driving object-oriented research in the next few years, Kent [Ken90a] defines a framework which emphasizes behaviors and their invocations as a means of comparing the "objectness" of models, the X3/SPARC/DBSSG/OODBTG report [FKMT91] defines an open object model architecture and recommends some standards for object data management (ODM), and several other classifications of object-oriented concepts have also appeared [CW85, SB85, AC86, KC86, Ull87, Weg87, Kin89, Mai89, Nie89, Str90]. These papers serve as useful guidelines to measure the "objectness" of various models. The formal model proposed here draws from all these reports and incorporates several of their core concepts. Several other models that have also influenced our design are discussed below.

Kent [Ken90b] defines a model which specifies a rigorous semantics for the existence of objects through identities and has separated this from the access of objects which is achieved through object references. Our model of object identity and reference is based on the concepts presented by Kent.

Snyder [Sny90] defines a generalized abstract object model which includes a set of core concepts and terminology that is meant to represent the essence of object models. These concepts intend to be abstract enough so that any specific object model may be built from them by refining and populating the general model. Our definitions are open and extensible because of the uniform treatment of objects. Extensions are made through subtyping and refinement of behaviors which may uniformly be applied to the schema objects of the model.

Beeri's model [Bee90] analyses and classifies the formal aspects of common features found in most current OODBMSs. He presents his ideas in a framework consisting of both the structural and behavioral model components. The structural model deals with the concepts of structured objects vs. values, object identity, inheritance graphs and declarative languages. The behavioral component explores the higher-order concepts of object-orientation such as model uniformity, method semantics and inheritance. The paper is a sketch of ideas and is meant as a motivator for object-oriented theorists and designers. Emphasis on logic-oriented modeling is evident throughout the paper. The structural model presented in Chapter 4 evolved directly from the concepts presented by Beeri.

Maier, Zhu and Ohkawa [MZO89] outline the structural object model TEDM which encompasses prominent features of the object-oriented and logic programming worlds. From the object-oriented side, the model incorporates support for object identities, complex objects, type structures and property inheritance. Types in the model have both an intensional and extensional aspect. The intensional view consists of the structural specification of the type, while the extension represents the collection of objects adhering to the structure of the type. Thus, the model separates the notion of a type from its extent. However, the entire extension of a type is not automatically maintained by the model (i.e., there is no notion of a class) and in this respect resembles the structural model of Beeri [Bee90]. Our model supports the separation of type and extent, but additionally maintains the extent of a type through a class which may have additional user-collections defined over them to customize object structuring. In this way, we have a single accessible object (a class) which gives the entire extent of a type without losing any of the flexibility of general collections. From the separation of types and extents, the notions of specialize and subtyping evolve and are defined by TEDM. The applicability of these relationships to type inferencing is the reason we have included them in the design of TIGUKAT.

The PROBE Data Model (PDM) [MD86] draws from Shipman's DAPLEX functional data model. PDM defines *entities* which denote individual things such as PERSON or MATERIAL, and *functions* to represent properties of entities and the relationships among them. PDM generalizes functions of DAPLEX by defining a function as a relationship between collections of entities and scalar values. This generalization allows functions with zero or more inputs and multiple outputs, or arguments that serve as both input and output. PDM handles both stored and computed functions but syntactically treats all function references as computed functions which is the approach taken by our model.

OODAPLEX [Day89] extends DAPLEX into an object-oriented model by directly building on the PROBE model. The extensions to DAPLEX include abstraction, encapsulation of behavior, closure, and enhancement of the declarative language features by allowing for recursive queries and additionally describing a companion algebra.

Iris [FBC+87, FAC+89, WLH90] is an object-oriented database management system which is founded on the functional data model of Daplex [Shi81]. The Iris model defines *objects*, *types* and *functions*. Objects are classified into the categories of *literal* (atomic) and *non-literal* (complex) objects. The literal objects are the directly system representable atomic building blocks of the non-literal objects. Iris completely encapsulates object properties into behaviors (i.e., functions or operations) which represent the only interface to objects. Thus, a high level of data abstraction and data independence is supported by the model. Operations take objects as arguments and produce objects as results. All objects are classified into types which define the operations applicable to objects in the extent of that type. Types may be structured into subtype/supertype relationships where multiple subtyping is supported. These "classes" of objects may overlap, meaning an object may belong to several heterogeneous types simultaneously unless there is an explicit declaration restricting classes to be disjoint (classes in subtype/supertype relationships must overlap). There is no support for separate user defined collections apart from classes. TIGUKAT adopts complete encapsulation of behaviors which uniformly accept objects as inputs and produce objects as results. The structural model refines this perspective by distinguishing between atomic, abstract and complex structured values. Our model supports heterogeneity through collections where classes are restricted collections of objects that are in subset relationships with one another.

The Functional/Relational Object-Oriented Model (FROOM) [MB90] is an object-oriented model with a goal of heterogeneous, persistent, distributed object management. The object model of FROOM is functional in nature; similar to DAPLEX and PROBE. FROOM has a uniform definition and treatment of objects and access to objects is gained only through behaviors which are implemented via functions that are also considered to be objects. This uniform perspective of objects is refined and formally incorporated into the definitions of TIGUKAT.

O_2 is a commercially available OODBMS [Deu90, Deu91]. It consists of a formal model definition based on the framework of a set-and-tuple data model [LRV88, BBB+88] which includes set, tuple and list constructors for modeling complex nested objects [LR89a]. The model supports subtyping based on the set inclusion semantics developed in [Car84] which establishes classes of objects. However, explicit user-defined collections are not supported. Furthermore, the system defines a language independent object-oriented database programming language (OODBPL) [LR89b], an ad-hoc query language [BCD89] and transaction management facilities [VBD89]. The query language is a subset of the OODBPL and thus does not suffer from an impedance mismatch problem. The success of O_2 as an implementation of an OODBMS, makes it a valuable guideline for developing an object model with practical applicability in industry.

Several other experimental systems have provided insights into the development of object-orientation and this proposal considers their contributions as well. Smalltalk [GR85] was one of the first commercially available software that incorporated features associated with object-orientation. However, Smalltalk is an object-oriented programming language which lacks the functionality of database systems. GEMSTONE [CM84, BMO+89] is a prototype system which added database features to Smalltalk making it one of the first OODBMSs. Other contributions come from ORION [BCG+87, KBC+89, KGBW90], EXODUS [CDF+88, CDV88], FAD [BBKV87], LOGRES/ALGRES [CCCR+90], CACTIS [Hud86], CLASSIC [BBMR89] and EMERALD [BHJ+87].

One unconventional approach which has generated ideas about object existence and reference is the formal model proposal by Wand [Wan89] which uses the philosophy of ontology [Bun77, Bun79] in defining the notion of an object. His technique introduces an intriguing philosophical perspective in defining the foundations of a formal object model. An ontological approach has applications in the design of object models because these models are expected to have high levels of abstraction;

and the more abstract we become in our definitions, the more we embrace the issues of philosophy.

5.2 Conformance to Manifestos

Following [MB90], we organize the discussion along the structure of [ABD+89] and refer to [SRL+90] periodically. The characteristics of an OODBMS are segmented in [ABD+89] into *mandatory* and *optional* parts. There are also a number of features which the authors were not able to agree on a classification at the time. Furthermore, they specify several open design decisions that they thought were best handled by the model designer because no consensus had been reached on them by the scientific community and it was uncertain at the time which of the alternatives were more or less object-oriented. We consider each of their issues in turn.

5.2.1 Mandatory requirements

Complex objects. Our model supports complex objects. TIGUKAT is functional in that objects (and their properties) are only accessible through the applications of behaviors. The model is uniform in that everything is an object including behaviors and their implementations. Since behaviors are mappings from objects into other objects, every object may be considered as a complex object. Our model does not explicitly incorporate the notion of constructors. Instead, we uniformly define a type which exhibits the behavior of a desired constructor. For example, our model defines an atomic *set* type whose instances are sets and whose behaviors are operations on sets. We have similarly defined atomic types for *bags, posets* and *lists*. *Tuples* in TIGUKAT are cast into the notion of behaviors on types which adheres to the uniformity of the model. These implicit "constructors" may be uniformly applied to build structures of any object in the system.

Object identity. Our model supports strong object identity, meaning objects have a unique, immutable, system managed identity. This contrasts [SRL+90] which emphasizes the importance of user-specified identities. The notion of user identities are always supportable through behaviors which are defined and managed by the user, regardless of whether system identities are defined or not.

Encapsulation. Our model fully encapsulates the state of objects whose only access is through a set of public behaviors defined on its type. Objects may be viewed as instances of abstract data types.

Types and Classes. We separate the notions of type and class and attach a different semantics to each one. We define a type as a specification tool (template) for objects and a class as a grouping mechanism for instances of a type. A class has a number of restrictions defined on it which impose a structure on the groupings of objects. We also define a collection in our model, not mentioned in [ABD+89], that serves as a more general user-specified grouping mechanism.

Class or Type Hierarchies. Our model defines two categories of "inheritance hierarchies." The first refers to the inheritance of behavioral specifications on types (called *behavioral inheritance*) which is defined by specialize and subtyping relationships on types. The second is an inheritance mechanism for the methods (functions in our model) which implement behaviors

(called *implementation inheritance*). We are careful to attach individual semantics to each one.

Overriding, overloading and late binding. These notions are supported in TIGUKAT through the separation of the *behavioral* and *implementation* inheritance hierarchies. We consider the semantics of behaviors to be separate from the functions which provide those semantics. This means that behaviors may be defined on many types (*overloading*) and that the implementation of the behavior may be different (redefined) for each type (*overriding*). *Late binding* is more a language support issue and is not part of the formal model definition. However, it does mean that an applications language for our model must facilitate the late binding property.

Computational completeness. Since our model is functionally based and uniform, any computable function can be defined and attached to any behavior of a type in the system. Furthermore, the query model will define a set of operations on the model which will be computationally complete. This we feel is in support of the computational completeness requirement.

Extensibility. Our model is fully extensible through the operations provided on the higher level constructs as shown in Section 3.6. The additional benefit is that these operations are uniformly provided as behaviors on primitive types.

Persistence. Our model maintains orthogonality between persistent and transient objects; persistence is orthogonal to type. The manner in which objects can be made persistent or transient is a language issue which will be considered as part of the database language methodology. The different storage and management requirements of persistent and transient objects is an implementation issue which is outside the object model considerations.

Secondary storage management. This is an implementation design issue and is not part of the object model specification. [SRL+90] explicitly states that these kinds of issues should not be addressed in the data model and we refrain from doing so.

Concurrency and recovery. This is a consideration for an object transaction model which is not part of this proposal.

Ad hoc query facility. One of our most prominent directions is looking at propagating the definition of a query model into the structure of our object model. In doing so, we wish to extend the query facility to a full object calculus and algebra which is computationally complete.

5.2.2 Optional Features

Multiple inheritance. Our model provides multiple inheritance as explained in the manifesto papers through what we prefer to call multiple subtyping. We attach a different meanings to the term *inheritance* which refers to the reuse of behaviors and implementations. The general consensus nowadays is that multiple subtyping is a mandatory feature of an OODBMS.

Type checking and type inferencing It has already been proven [SÖ90a] that much of the type checking involved in query processing can be performed at compile time. Our subsequent query model definition will also support type inferencing and dynamic schema which will assist in deriving type information for queries which return objects of heterogeneous types.

Distribution Distribution is an issue related to the implementation of the model and should be transparent within the model definition itself. We are not currently considering the problems associated with distributing our system.

Design transactions Design transactions are part of a transaction model for the system which is not considered in this proposal.

Versions We have not defined a version support mechanism for our model. However, the uniformity aspects of the model may provide some assistance in developing one.

5.2.3 Undetermined Mandatory or Optional

View definition and derived data. Views are part of the future work of this research. A view mechanism with update semantics will be defined for our object model.

Database administration utilities. This is an implementation consideration and is not part of the core model definition. However, any computable function can be defined as a behavior on objects in the system. Thus, required database administration utilities may be supplied as behaviors on the primitive types or the type system may be extended to include objects which facilitate these utilities.

Integrity constraints. We have not included integrity constraints in our model definition. Again, it is questionable if these should be part of the core model definition. However, our model has the notion of predicates defined on collections. These may be helpful in easily supporting certain integrity constraints (e.g., "the salary of all employees in this collection should be under $75,000"). Nevertheless, these predicates are not sufficiently powerful to specify constraints over multiple collections (e.g., referential integrity). Furthermore, using the functional nature of our model, behaviors may be defined for automatically maintaining the integrity of objects. That is, the type implementor defines an update interface of behaviors which must be used to modify objects and which maintains the integrity of the objects.

Schema Evolution. The type system of our model is fully extensible through primitive behaviors defined on the primitive types. The query model specification of our model will include dynamic schema evolution policies to assist in automatically deriving type information for queries which return objects of heterogeneous types.

5.2.4 Open Choices

Programming paradigm. Our model separates behavioral specifications from their implementations. This provides implementation independence. Since functions are a separate primitive in the model, their implementation may be specified in practically any language. The only requirement is that they must adhere to the semantics defined by their associated behavior.

Representation system. Our model supports a basic set of types which include the functionality to uniformly extend all parts of the type system, including the atomic types.

Type system. As indicated in the point above, we provide a basic type system which is fully extensible.

Uniformity Our model uniformly treats all entities as objects. This includes all the primitives such as *object, type, class, collection* and *function*. We feel this is an important feature in several respects. From the modeling perspective, we have a clean, self-contained description of the model with no dependence on external meta information. From the language point of view, we have a single uniform approach in accessing and manipulating all information in the system. In the query model this means the efficient query operators may uniformly be applied to the modeling primitives, thereby providing a powerful, ad-hoc access mechanism to what is essentially meta-information.

5.3 Conformance to OODB Task Group Recommendations

Many of the notions covered by the manifestos are repeated in the ODM reference model [FKMT91]. For this reason, we only point out those recommendations which differ from the manifestos and which are applicable to the object model component of an OODBMS.

- We use the "classical or messaging object model" paradigm where the recipient of a behavior is always explicit.
- We define exactly the notion of identity given in the report and use object references as the "logical identifiers" of objects.
- We define a much clearer separation of type and class than given.
- As a consequence of the previous point, our definitions of subtyping, behavioral inheritance and implementation inheritance have a much cleaner separation and semantics.
- We use the notion of "literals" to refer to atomic objects which encapsulate reference, identity and state.
- We support the argument that the only equality needed in a model definition is that of "identity equal."

The other components of the ODM reference model comply with those covered in Section 5.2 or are related to non-data model issues such as storage management, query models, transaction management and programming languages.

Chapter 6

Conclusions and Future Work

The initial goals of this research were to identify and formalize a set of object-oriented characteristics into an object model definition which exhibits sufficient power and expressibility for supporting the functionality of advanced applications such as geographic information systems, engineering databases, office information systems, knowledge base systems, and multi-media databases. These applications are regarded as managers of complex data types with complex relationships. User access to such systems are characterized by long-running, interactive transactions that involve large and semantically diverse units of data. Thus, the functionality required of OODBMSs subsumes the functionality of their predecessors. Yet, fundamental research on some of the common database features such as query models, query processing, view mechanisms, schema evolution and transaction management are quite sparse. Part of the reason is the lack of a universally accepted, sufficiently powerful and formal object model. This proposal defines a formal behavioral object model and integrates it with a formal structural counterpart. The coupling of these two components is an essential first step towards meeting the advanced requirements demanded by OODBMSs. The completion of this initial task establishes a theoretical foundation which facilitates a systematic investigation of various other open issues concerning object database management.

The complete behavioral/structural object model forms a solid foundation for prototype development of an OODBMS. The work on this prototype is currently underway. The structural model is the interface between the high level conceptual behavioral model and a low level storage management subsystem. The prototype will initially concentrate on the implementation of the higher level object model aspects rather than on low level object storage management. Therefore, we plan to couple our prototype with an existing object storage manager subsystem such as the Wisconsin Storage System (WiSS) [CDKK85] or the O_2Engine of the O_2 OODBMS [Deu91, VBD89]. This approach will allow us to focus on the implementation of an object query language, object views and object database programming language support. The issues related to object storage management are quite complex and require a significant amount of research, especially in the forum of distributed systems.

The fundamental object modeling contributions of this research are the following:

- a formal specification of both the behavioral and structural aspects of an object model with sufficient power for handling advanced database functionality,
- a formal integration of the behavioral and structural components,

- a clean separation and a formal definition of many object model features which usually are only intuitively defined in other studies, and
- a uniform approach to objects which includes meta-information as primitive objects in the model, a consistent object reference access method, and a homogeneous treatment of object behaviors as functions.

The remaining major issues of our research not addressed in this report include the definition of an object query model and query formalism with a complete object calculus and algebra specification that take into account update semantics. As mentioned, we will revisit the query model definitions of [Str91] and [SÖ90a] to enhance them in accordance with the object model extensions introduced in this report. Two issues that demand specific attention are the proper handling of quantifiers in both the calculus and algebra (requiring additional operators), and the extension of the object algebra by the inclusion of "object creating" operators. Currently, the algebra only supports "object preserving" operators [SS90][1]. These additions require the services of a dynamic schema evolution policy which will be incorporated into the object model at this time.

Query models offer a significant advantage over conventional programming languages in that queries support efficient associative access to data. Using a declarative calculus, one formulates queries by focusing on "what" information is required rather than "how" to go about retrieving the information; accessing the information efficiently is the responsibility of a procedural algebra. An essential component contributing to the success of this approach is a calculus to algebra equivalence proof which mechanizes the transformation from a declarative query into a procedural one that can be optimized by applying a series of equivalence rewrite rules. When developing the query model, it will be foremost to insure closure, finiteness and type consistency of the definitions.

The definition of an object query model will provide the necessary efficient associative access to objects. However, this model will initially only work on the physical database as defined by its *conceptual schema*. A conceptual schema can have various meanings to different applications, meaning there can be several *external schemas* for a single *conceptual schema*. These various perspectives should be specifiable in a model without having to redefine the physical or conceptual schema for each application. The definition and management of external schemas will be specified as part of a view management facility which includes update semantics in order to provide efficient associative access and manipulation of objects in the database. Following the uniformity of the model, views would be first class objects. Their precise semantics within this context and their management strategies will be developed.

The object-oriented approach is seen as a suitable candidate for facilitating an integration between the data abstraction and computation model of object-oriented programming languages with the performance and consistency of an object query model. Traditionally, these two areas have developed orthogonally to one another. An integration would alleviate many problems associated with embedded languages in use today. We plan to investigate and characterize the properties of such an integration in order to develop a methodology for specifying a seamless interface between these two disciplines.

The realizable contributions of the subsequent research include:

- an object query model definition with complete object calculus and algebra specifications including an equivalence proof,

[1] Object preserving algebras are restricted in that they cannot create new objects.

- a uniform definition of object views with a consistent semantics for view updates and view management,
- a type consistent dynamic schema evolution policy, and
- a methodology for integrating the object query model with object programming languages.

Acknowledgements

The authors would like to express a sincere thank you to all those who contributed to the development of the TIGUKAT model. We especially thank the other members of the Object Database Group which include Boman Irani, Anna Lipka, Adriana Muñoz, Ana Domínguez and Youping Niu for their numerous readings of earlier drafts of this work and for their many helpful comments.

Bibliography

[AB84] S. Abiteboul and N. Bidoit. Non First Normal Form Relations to Represent Hierarchically Organized Data. In *Proc. of the 3rd ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 191–200, Waterloo, Ontario, April 1984.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K.Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. 1st Int'l. Conf. on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, Japan, 1989.

[AC86] G. Ariav and J. Clifford. Database Research and Systems: Key Issues in Perspective. In G. Ariav and J. Clifford, editors, *New Directions for Database Systems*, pages 1–9. Ablex Pub. Corp., 1986.

[AH84] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. In *Proc. of the 3rd ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 119–132, Waterloo, Ontario, April 1984.

[All86] L. Allison. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, 1986.

[Aro89] S. Aronoff. *Geographic Information Systems: A Management Perspective*. WDL Publications, 1989.

[Bar81] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1981.

[BBB+88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez. The Design and Implementation of O_2: An Object-Oriented Database System. In *Proc. of the 2nd Int'l Workshop on Object-Oriented Database Systems*, pages 1–22. Springer Verlag, September 1988.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a Powerful and Simple Database Language. In *Proc. 13th Int'l Conf. on Very Large Databases*, pages 97–105, Brighton, England, September 1987.

[BBMR89] A. Borgida, R.J. Brachman, D.L. McGuinness, and L.A. Resnick. CLASSIC: A Structural Data Model for Objects. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, Portland, Oregon, June 1989.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for the O_2 Object-Oriented Database System. In *Proc. 2nd Int'l Workshop on Database Programming Languages*, pages 122–138, Gleneden Beach, Oregon, June 1989.

[BCG+87] J. Banerjee, H.T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, and H.J. Kim. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[Bee90] C. Beeri. A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering*, 5:353–382, 1990.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[BK90] F. Bancilhon and W. Kim. Object-Oriented Database Systems: In Transition. *ACM SIGMOD Record*, 19(4):49–53, 1990.

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

[BMO+89] R. Brentl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, 1989.

[Bun77] M. Bunge. *Treatise on Basic Philosophy: Vol 3: Ontology I: The Furniture of the World*. Reidel, Boston, 1977.

[Bun79] M. Bunge. *Treatise on Basic Philosophy: Vol 4: Ontology II: A World of Systems*. Reidel, Boston, 1979.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In *Int'l Symposium on Semantics of Data Types*, pages 51–67, Sophia-Antipolis, France, June 1984.

[Car86] L. Cardelli. A Polymorphic λ-calculus with Type:Type. Research Report 10, DEC Systems Research Center, 130 Lytton Ave. Palo Alto, California 94301, May 1986.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 225–236, Atlantic City, N.J., June 1990.

[CDF+88] M. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita. The Architecture of the EXODUS Extensible DBMS. In M. Stonebraker, editor, *Readings in Database Systems*, pages 488–501. Morgan Kaufmann Publishers, 1988.

[CDKK85] H-T. Chou, D. Dewitt, R.H. Katz, and A.C. Klug. Design and Implementation of the Wisconsin Storage System. *Software – Practice & Experience*, 15(10):943–962, October 1985.

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

[Che76] P.P.S. Chen. The Entity–Relationship Model: Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[CM84] G. Copeland and D. Maier. Making Smalltalk a Database System. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 316–325, Boston, Mass., June 1984.

[Cod70] E.F. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod79] E.F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4), December 1979.

[CP89] W. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications*, pages 433–443, New Orleans, Louisiana, October 1989.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Day89] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proc. 2nd Int'l Workshop on Database Programming Languages*, pages 80–102, Gleneden Beach, Oregon, June 1989.

[Deu90] O. Deux, *et. al.* The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[Deu91] O. Deux, *et. al.* The O_2 System. *Communications of the ACM*, 34(10):34–48, October 1991.

[DKA+86] P. Dadam, K Kuespert, F. Anderson, H. Blanken, R. erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch. A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 356–367, Washington, DC, May 1986.

[FAC+89] D.H. Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, and W.K. Wilkinson. Overview of the Iris DBMS. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, 1989.

[FBC+87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[FKMT91] E. Fong, W. Kent, K. Moore, and C. Thompson. X3/SPARC/DBSSG/OODBTG Final Report. Technical report, NIST, September 1991.

[GR85] A. Goldberg and D. Robson. *SMALLTALK-80: The Language and its Implementation*. Addison-Wesley, 1985.

[Haa90] L.M. Haas, *et. al.* Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HM78] M. Hammer and D. McLeod. The Semantic Data Model: A Modeling Mechanism for Database Applications. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 26–36, Austin, Texas, May-June 1978.

[HM81] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[Hud86] S.E. Hudson. CACTIS: A Database System for Specifying Functionally-Defined Data. In *1986 Int'l Workshop on Object-Oriented Database Systems*, pages 26–37, Pacific Grove, California, September 1986.

[JGF+88] D. Jagannathan, R.L. Guck, B.L. Fritchman, J.P. Thompson, and D.M. Tolbert. SIM: A Database System Based on the Semantic Data Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 46–55, Chicago, Illinois, September 1988.

[JS82] G. Jaeschke and H. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proc. of the 1st ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 124–138, Los Angeles, California, March 1982.

[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, Portland, Oregon, September 1986.

[Ken79] W. Kent. Limitations of Record-Based Information Models. *ACM Transactions on Database Systems*, 4(1):107–131, March 1979.

[Ken90a] W. Kent. A Framework for Object Concepts. Technical Report HPL-90-30, Hewlett Packard Labs, April 1990.

[Ken90b] W. Kent. A Rigorous Model of Object Reference, Identity and Existence. Technical Report HPL-90-31, Hewlett Packard Labs, April 1990.

[KGBW90] W. Kim, J.F. Garza, N. Ballou, and D. Wolek. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[Kim90a] W. Kim. Object-Oriented Databases: Definition and Research Directions. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):327–341, September 1990.

[Kim90b] W. Kim. Research Directions in Object-Oriented Databases. In *Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–15, Nashville, Tenn., April 1990.

[Kin89] R. King. My Cat is Object-Oriented. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, 1989.

[KW89] M. Kifer and J. Wu. A Logic for Object-Oriented Programming (Maier's O-Logic Revisited). In *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, Philadelphia, Penn., March 1989.

[LR89a] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 360–368, Philadelphia, Penn., March 1989.

[LR89b] C. Lécluse and P. Richard. The O_2 Database Programming Language. In *Proc. 15th Int'l Conf. on Very Large Databases*, Amsterdam, August 1989.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 424–433, Chicago, Illinois, September 1988.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[Mai89] D. Maier. Why isn't there an Object-Oriented Data Model. In *Proceedings of IFIP* 11^{th} *World Computer Conference*, San Francisco, CA, August-September 1989.

[MB90] F. Manola and A.P. Buchmann. A Functional/Relational Object-Oriented Model for Distributed Object Management. Technical Memorandum TM-0331-11-90-165, GTE Laboratories Incorporated, Waltham, MA, December 1990.

[MD86] F. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In K.R. Dittrich and U. Dayal, editors, *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 18–25. IEEE Computer Science Press, 1986.

[MZO89] D. Maier, J. Zhu, and H. Ohkawa. Features of the TEDM Object Model. In *Proc. 1st Int'l. Conf. on Deductive and Object-Oriented Databases*, pages 476–495, Kyoto, Japan, 1989.

[Nie89] O. Nierstrasz. A Survey of Object-Oriented Concepts. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, 1989.

[OH86] S. Osborn and T.E. Heaven. The Design of a Relational Database System with Abstract Types for Domains. *ACM Transactions on Database Systems*, 11(3):357–373, 1986.

[ÖS91] M.T. Özsu and D.D. Straube. Issues in Query Model Design in Object-Oriented Database System. *Computer Standards & Interfaces*, 13:157–167, 1991.

[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.

[ÖSP92] 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, *Advances in Databases and Artificial Intelligence*. JAI Press, 1992. In press.

[OY87] Z.M. Ozsoyoglu and L.Y. Yuan. A New Normal Form for Nested Relations. *ACM Transactions on Database Systems*, 12(1):111–136, March 1987.

[Pie88] B.C. Pierce. A Taste of Category Theory for Computer Scientists. Technical Report CMU-CS-88-203, Carnegie Mellon University, 1988.

[PM88] J. Peckham and F. Maryanski. Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189, September 1988.

[Rev89] G.E. Revesz. *Lambda-Calculus, Combinators, and Functional Programming.* Cambridge University Press, 1989.

[RK87] M.A. Roth and H.F. Korth. The Design of ¬1NF Relational Databases into Nested Normal Form. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 143–159, San Francisco, May 1987.

[RK89] N. Roussopoulos and H.S. Kim. ROOST: A Relational Object Oriented System. In *Proc. 3rd Int'l Conf. on Foundations of Data Organization and Algorithms*, pages 404–420, Paris, France, June 1989.

[RS87] L.A. Rowe and M.R. Stonebraker. The POSTGRES Data Model. In *Proc. 13th Int'l Conf. on Very Large Databases*, pages 83–96, Brighton, England, September 1987.

[SB85] M. Stefik and D. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, pages 40–62, 1985.

[Sch85] H. Schek. Toward a Basic Relational NF^2 Algebra Processor. In *Proc. Int'l Conf. on Foundations of Data Organization*, pages 173–182, Kyoto, Japan, May 1985.

[Sch88] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Wm. C. Brown Publishers, 1988.

[Shi81] D.W. Shipman. The Functional Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), March 1981.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.

[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 *ECOOP/OOPSLA '90 Proceedings*, pages 224–233, Ottawa, Canada, October 1990.

[SÖ91] D.D. Straube and M.T. Özsu. Execution Plan Generation for an Object-Oriented Data Model. In *Proc. 2nd Int'l. Conf. on Deductive and Object-Oriented Databases*, pages 43–67, Munich, Germany, December 1991.

[SR86] M. Stonebraker and L.A. Rowe. The Design of POSTGRES. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 340–355, Washington, DC, May 1986.

[SRH90] M. Stonebraker, L.A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[SRL+90] M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P.Bernstein, and D. Beech. Third-Generation Data Base System Manifesto. *ACM SIGMOD Record*, 19(3):31–44, September 1990.

[SS77] J.M. Smith and C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, June 1977.

[SS86] H. Schek and M. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2):137–147, 1986.

[SS90] M. Scholl and H. Schek. A Relational Object Model. In *Third Int'l Conf. on Database Theory*, pages 89–105, Paris, France, December 1990.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* M.I.T. Press, 1977.

[Sto88] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In M. Stonebraker, editor, *Readings in Database Systems*, pages 480–487. Morgan Kaufmann Publishers, 1988.

[Str90] D.D. Straube. An Introduction to Object-Oriented Databases. In 15th *Simposium Internacional de Sistemas Computacionale*, Monterrey, Mexico, March 1990.

[Str91] D.D. Straube. *Queries and Query Processing in Object-Oriented Database Systems.* PhD thesis, University of Alberta, 1991.

[SZ89] G. Shaw and S. Zdonik. An Object-Oriented Query Algebra. In *Proc. 2nd Int'l Workshop on Database Programming Languages*, pages 103–112, Gleneden Beach, Oregon, June 1989.

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

[Tom90] C.D. Tomlin. *Geographic Information Systems and Cartographic Modeling.* Prentice-Hall, 1990.

[Ull87] Jeffrey D. Ullman. Database Theory: Past and Future. In *Proc. of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 1–10. ACM Press, March 1987.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager: An Overview. In *Proc. 15th Int'l Conf. on Very Large Databases*, Amsterdam, August 1989.

[Wan89] Y. Wand. A Proposal for a Formal Model of Objects. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications.* Addison Wesley, 1989.

[Weg87] P. Wegner. Dimensions of Object-Based Language Design. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications*, pages 168–182, Orlando, Florida, October 1987.

[Weg90] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, March 1990.

[WSSH88] P.F. Wilms, P.M. Schwarz, H.J. Schek, and L.M. Haas. Incorporating Data Types in an Extensible Database Architecture. In *Proc. 3rd Int'l Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 180–192, Jerusalem, Israel, June 1988.

[Yan90] M. Yannakakis. Graph-Theoretic Methods in Database Theory. In *Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 230–242, Nashville, Tenn., April 1990.

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

[ZM90] Stanley Zdonik and David Maier. Fundamentals of Object-Oriented Databases. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 1–36. Morgan Kaufmann Publishers, 1990.