University of Alberta

Implementation of the TIGUKAT Object Model

by

Boman B. Irani

Technical Report TR 93–10

June 1993

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

UNIVERSITY OF ALBERTA

Implementation of the TIGUKAT Object Model

BY

Boman B. Irani

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1993

Abstract

The object-oriented paradigm of computing has started to have a significant influence on many areas of information and data processing, including database systems. This thesis focuses on the various issues and aspects governing the implementation design and development of the object model for TIGUKAT[1], an object management system which is intended to be a full featured object-oriented database system on completion. The TIGUKAT object model [25] is *behaviorally* defined with a *uniform* object semantics. The model is *behavioral* in the sense that all access and manipulation of objects is restricted to the application of behaviors on objects, and it is *uniform* in that every entity within the model has the status of a *first-class object*. Various implementation design alternatives are discussed and the approaches that were chosen are justified. The ensuing implementation provides a robust kernel around which the rest of the system may be conveniently synthesized.

[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.

Acknowledgements

I would like to express my deepest gratitude and appreciation to Dr. M. Tamer Özsu, my supervisor, for granting me this wonderful opportunity to work under him. He has provided me with constant support throughout this research in the form of guidance, invaluable advice, encouragement and funds. It was a rewarding and pleasant experience to have worked with him and I could not have hoped for a better *guru*.

Dr. Duane Szafron provided stimulating ideas, suggestions and directions, for which I am sincerely grateful. Thanks are also due to my other thesis committee members, Dr. Jack Mowchenko and Dr. Peter van Beek, for their insightful comments.

Very many warm thanks to Ana, Adriana, Iqbal, Youping, Randy and Yuri. You guys made the DB lab a whole lot livelier.

Without the love and support from my parents, I would have never made it this far. Special thanks to my wonderful Memas for all her prayers and affection and also to my sister, Gover, for her loving encouragement.

And last, but definitely not the least, Shalini, you are *the* most beautiful person in my life.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

It is now commonly accepted that relational database management systems (RDBMSs) cannot support the needs of new applications. Many of these shortcomings stem directly from the *flat* nature of the relational data model. This inadequacy is evident when we attempt to gracefully accommodate complex data and its manipulation within the relational framework. Object-oriented database systems (OODBSs) primarily aim to alleviate this deficiency.

The TIGUKAT[1] project's primary objective is the design and implementation of a full fledged next-generation data management system which would efficiently and reliably handle complex data as found in many data-intensive application environments. These would include geographical information systems (GIS), CAD/CAM/CAE, software engineering, expert systems and office automation, to mention just a few.

The semantic richness of an object model offers enhanced power in the modeling of real world concepts as data while preserving a high level of abstraction and encapsulation, leading to a greater degree of modularization, reusability and interoperability. Researchers have proposed numerous extended relational, semantic and functional data models [14, 22, 31] which promise functionality far exceeding that of traditional relational systems. Many of these semantic data-modeling concepts have been incorporated into the TIGUKAT object model [24, 25].

1.2 Objectives and Overview

TIGUKAT is being designed as a full-featured OODBS supporting a query language, transactions, versioning, and view management for accessing, updating and reliably manipulating large quantities of arbitrarily complex, shared data. The object-oriented paradigm is supported by the integration of features such as the uniform encapsulation of all system entities as *first-class* objects with unique object identities (oids), behavioral and implementation inheritance via single or multiple subtyping, runtime binding of behaviors to functions and the potential for convenient expansion through incremental development and code reuse.

This thesis focuses on the implementation of the conceptual core object model and a proposed architecture for achieving persistence. TIGUKAT's primitive object system was

[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.

mapped into a feasible implementation design conserving the prime aspects of uniformity and extensibility as dictated by the conceptual model. Since TIGUKAT has a generic object model, its implementation is isolated from the type system of the language (C++) used to implement it. Issues governing the modeling of *meta* and *meta-meta* data within the core model, behavior application (method dispatch), the behavioral and implementation inheritance mechanisms and the handling of stored and computed functions proved to be the most significant. Some of the choices we made may have performance penalties. We make a note of these and suggest future remedial measures which ought to be taken to mitigate them.

We propose a suitable system design to achieve persistence employing the EXODUS storage manager (ESM) [8, 11, 34] to manage migration of system objects to and from disk. While the storage manager (SM) perceives objects as passive sequences of uninterpreted bytes of arbitrary size, determines the on-disk format of objects and performs any necessary translation between memory and disk formats, it is the object manager (OM) that implements object creation and deletion as instances of types, message passing between objects (behavior application) and fetching of persistent objects through the SM. All manipulation of objects is undertaken via the OM which also attaches object semantics to the underlying byte streams and represents them as *objects*. The OM makes decisions as to when objects are required to be read from or written to disk and it utilizes the SM's capabilities to perform these functions efficiently.

To date very limited information is available on a complete implementation of an object-oriented system although a number of such systems are in existence. We aim to present a demonstrable system implementation in this thesis stressing the mapping from the conceptual to the implementation level.

1.3 Organization of the Thesis

This thesis is divided into five chapters. In this introductory chapter we addressed the problem we attempt to tackle and our motivation for doing so. The TIGUKAT object model is introduced in sufficient detail in Chapter 2. We highlight those features which are of direct concern to the system implementor.

The system implementation design and development are fully discussed in Chapter 3. We scrutinize each option available for the decisions and justify the approach we have chosen. Chapter 4 focuses on our proposed design to achieve persistence in the system. We discuss ESM, emphasizing those constructs which are of particular relevance to our design of persistence in TIGUKAT. We also give a brief literature survey with the intention of taking a "where are we now" look at some existing storage managers.

Finally, we conclude with Chapter 5, giving directions for future research and suggesting some enhancements and improvements for the present system.

Chapter 2

The TIGUKAT System

During the last decade a significant amount of research effort has been directed towards the individual areas of OODBSs and object-oriented programming languages (OOPLs). It is only recently that both these areas seem to be converging with a common goal in sight: a universal, extensible object model. In this chapter we discuss the architecture of the entire system and examine the TIGUKAT object model in sufficient detail, highlighting those features which demanded crucial consideration in many of our implementation design decisions. For the complete and formal specifications of the model, including the structural counterpart, we refer the reader to [25].

2.1 System Architecture

Depicted in Figure 2.1 is a simplified schematic of the TIGUKAT system architecture. With reference to this figure, the TQL block represents the TIGUKAT Query Language which is presently being developed. This module will interact with the object model through the query optimizer. The actual query processing will involve a sequence of transformations (not shown in the diagram) including the translation from calculus to algebra, typechecking, algebraic optimizations and execution plan generation. Details may be found in [32]. The TIGUKAT Definition Language (TDL) is the language interface to be used by the type implementor and has constructs for defining and generating new types, classes, objects, behaviors, etc.

ESM, shown at the lowest end of the system, will be responsible for persistent storage on disk. ESM supports a client-server topology where ESM's client module is linked with the host application program and interacts with the ESM server which may be running on the same or a different machine. In our system, the TIGUKAT library will be linked with ESM's client module to form an executable module.

The TIGUKAT library comprises the complete primitive object system, including behaviors, functions and the macros for atomic object creation. Communication between this module and the server is transparent to the OM. All object manipulation is via defined behaviors. The OM encloses the core model and is responsible for interaction with the higher layers of the system (primarily TDL and the query optimizer). It is also responsible for maintaining the oids of persistent objects and providing a correspondence between oids and objects on disk.

The client module may request persistent objects from the server, passing it an oid, or write objects out to disk via the server and storing the oid returned. It is the OM which attaches TIGUKAT object semantics to the bytes and presents them as an "object" to the

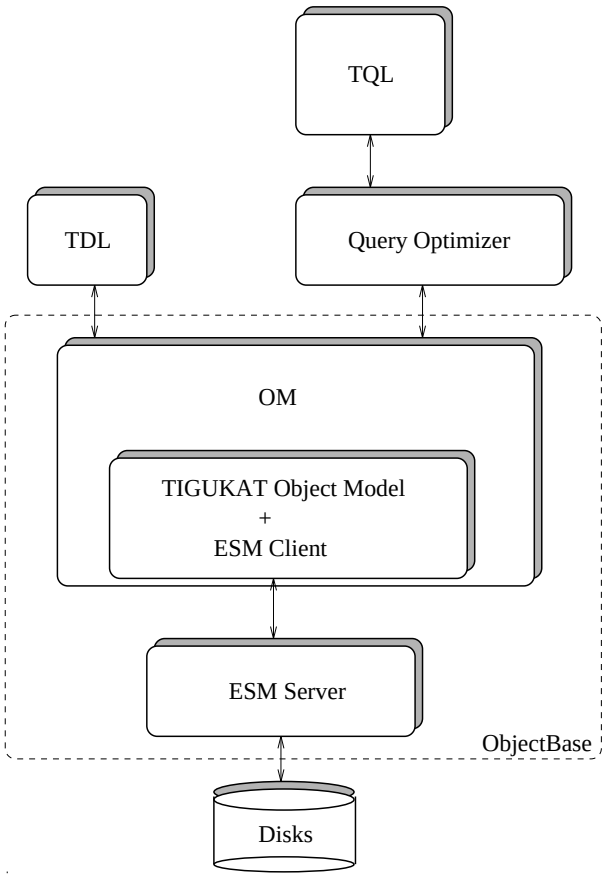


Figure 2.1: The TIGUKAT System Architecture

other modules. We discuss persistency issues in further detail in Chapter 4.

2.2 The Conceptual Object Model

The TIGUKAT object model is intended to serve as the central core of a system which may be conveniently extended[1] to support either an OODBSs or an OOPL or a fusion of the two. It is a *behavioral* model with a complete formal semantics attached and a seamless integration of behavioral and structural perspectives. *Uniformity* of all model abstractions is paramount and imparts a clean and precise semantics. Everything in the model (types, classes, etc.) is uniformly modeled as a *first-class object*[2] The *behavioral* nature ensues from the fact that all object access and manipulation is achieved by the application of *behaviors* (operations) on *objects*. An object will respond to the application of any behavior which is defined in its *type*'s interface.

TIGUKAT's uniformity aspects are similar to the approaches of the functional data models such as OODAPLEX [14] and FROOM [22]. The functional approach to defining

[1] *Extensibility* is the capability of defining new data types which are integrated into the existing type hierarchy such that no difference in usage exists between these user defined types and the system defined types [3].

[2] Database systems usually maintain a *schema* of the types defined in the database, shared across the system. This has been termed the *meta-information*, which is not treated as data [37]. The TIGUKAT model makes no differentiation between schema and data. Thus the concept of a *first-class object*.

behaviors has been borrowed from them and enhanced via integration with a structural counterpart. Several other experimental and commercial systems helped provide insights into the development of TIGUKAT. These include O_2 [5], Smalltalk [18] and EXODUS [11].

OODAPLEX is the object-oriented extension of the functional data model and the data language, DAPLEX [31], which de fines *entities* and *functions* as primitive modeling constructs. The properties of entities and the relationships among them are modeled as *functions*. OODAPLEX supports the concepts of encapsulation and permits recursive queries. TIGUKAT adopts a complete encapsulation of behaviors which uniformly accept objects as input and produce objects as results.

The FROOM (functional/relational object-oriented model) model is another functional model quite similar to OODAPLEX. FROOM perceives the uniformity of objects in definition and treatment allowing object access solely through behavior application. Behaviors are implemented via functions which are also objects [22]. TIGUKAT extends this perception of object s across every system entity.

The TIGUKAT model's groundwork is laid by a set of *primitive objects* which include: *atomic entities* such as *reals, integers, strings, characters, sets, bags, lists, etc.*; *types* for defining and structuring the information carried by common objects, including the operations which may be performed on them, within a centralized framework for these objects; *behaviors* for specifying the semantics of the operations which may be performed on objects; *functions* for specifying the implementations of behaviors over various types; *classes* for the automatic classification of objects related through their types; and *collections* for supporting possibly heterogeneous user-definable grouping of objects [25].

TIGUKAT objects are perceived as (*identity,state*) pairs where *identity* represents a unique, immutable, system managed identity implying the unique existence of every object and *state* represents the encapsulated information content contained by the object. An object identity (*oid*) is an internal reference to the object, not accessible to the user. This does not preclude extended environments from maintaining multiple *references* (or *denotations*) to objects which are not necessarily unique or which evolve depending on the scoping rules in effect as defined by the reference model. The state is the assortment of information content carried by the object and may be composed of references to other objects. Conceptually, every object is a *composite object* , which implies that every object has references (not necessarily implemented as pointers) to other objects. The identity of an object is distinct from the state and serves as an immutable internal identifier which is automatically maintained by the system without any user involvement. Nevertheless, user defined notions of *oids* are possible through application specific interpretations which may chose to recognize the result of a certain behavior application (e.g. *B_social_insurance_number*) as an oid for all objects of that type. TIGUKAT does not offer any exact semantics of object deletion or garbage collection of objects. These are issues presently being studied. Our implementation adheres strictly to the conceptual model and as such avoids any implementation details which lack a formal semantics.

TIGUKAT precisely delineates the means for defining an object's characteristics (i.e. a *type*) from the mechanism for grouping instances of that type (i.e. a *class*). The *type* serves as an *information repository* (*template*) which specifies object structure, behaviors and their implementations (*functions/methods*) for instances created using this type template. Types are organized in a lattice structure constrained by the subtyping notion which promotes software reuse and incremental extension of the lattice. If a type B is a *subtype* of type A, then B *must* inherit all the behaviors in type A's interface and may add additional behaviors

specific to type B. *Subtyping*[3] enforces an *isa* relationship between types (*substitutability*). TIGUKAT supports *multiple subtyping* where a type can defined as a direct subtype of multiple types. Since the semantics of behaviors is preserved across types, behavioral inheritance is no problem in the face of multiple subtyping, but inheriting implementations could be difficult to resolve. A conflict resolution policy is needed to decide (if possible) which implementation to inherit when more than one of the super types has semantically identical behaviors with disparate implementations. The naive approach is to always request user intervention in resolving conflicts. We discuss our implementation in Section 3.4. We opted for a partial solution attempting implementation inheritance but requesting intervention if it fails.

Every type that supports instantiation is paired with a corresponding *class* which serves to tie together that type and its object instances. This supplemental, but distinct, construct plays the role of instance manager for housing all instances of the type (the *extent* of the type). Object creation is via the unique corresponding class of the type whose instance is required. Creation is supported only if the type to be instantiated possesses an associated class. Thus, the model enforces a total mapping *classof* from objects into classes and a total, injective mapping *typeof* which maps each class to an unique type. Object creation occurs through a class, using the information content present in the corresponding type as a template and ensuring that all behaviors in the type's interface are applicable to these instances.

The model supports an additional object grouping construct, the *collection*, similar to a class but more general in certain respects. Unlike a class, a collection does not permit object creation, implying that only existing objects may be collected. An object may be a member of multiple collections, but class membership is restricted by the lattice structure on types. Classes are managed implicitly by the system whereas collections require user intervention in managing their extents. Finally, a collection may group objects of heterogeneous types while class membership is confined to only those objects which conform to its unique corresponding type.

The *behavior* objects define a semantics which describes their functionality. *Functions* implement this semantics, that is, they provide the *operational* semantics for a behavior. A behavior is applicable only to those objects which were created in accordance with a type whose interface incorporates that behavior. Although the implementation of a behavior, (i.e. the function it is associated with) may vary over the types which support it, implying that each type is free to provide its own implementation for that behavior, the semantics of that behavior remains constant and unique over all behaviors. Each behavior defined on a type must be associated with some implementation of the behavior for that type. No restriction is placed on what this implementation might be so long as it satisfies the behavior's semantics. If the same implementation is used by the subtypes, then the implementation is said to be *inherited*, but if it differs then we say that the implementation has been *redefined* or *overridden*. A behavior is applied on its first argument object, hereafter termed the *receiver*, whose type determines the appropriate function to be invoked. TIGUKAT has no concept of an *instance operation* or *class operation* as found in other systems such as Trellis/Owl [29] or of *instance variables* and *class variables* as in Smalltalk [18].

Figure 2.2 depicts the subtyping relationships of the *primitive type system*, τ. Each box represents a primitive type and the edges between boxes denote a subtyping relationship (left to right). Every type in τ is associated with a corresponding primitive class object and

[3]There is a distinction made in the model between *subtyping* and *specializing* [25]. A type is said to *specialize* some other type if it's behavioral specification subsumes that of the other type.

has primitive behaviors which are associated with some functions. The union of all the types in τ along with the set of primitive classes, behaviors, functions and other instance objects comprises the *primitive object system*, o. Since TIGUKAT supports multiple subtyping, the primitive type structure is potentially a directed acyclic graph, however, the addition of the base type, `T_null`, results in converting it to a lattice.

As can be seen in Figure 2.2, TIGUKAT is entirely self-contained. No external *meta-data* or *meta-meta-data* (traditionally known as the *schema* of the database) is required to support the core model. The uniformity aspect allows every entity to be managed as an object and the *higher level constructs* ensure this by encompassing the entire model within itself. These *higher level constructs* comprise the types `T_class_class`, `T_type_class`, `T_collection_class` and their corresponding classes (not shown in Figure 2.2), which are the basis of new class, type and collection generation, respectively [25]. We have also introduced the types `T_behavior_class` and `T_function_class` which offer the semantics for new behavior and function object creation respectively.

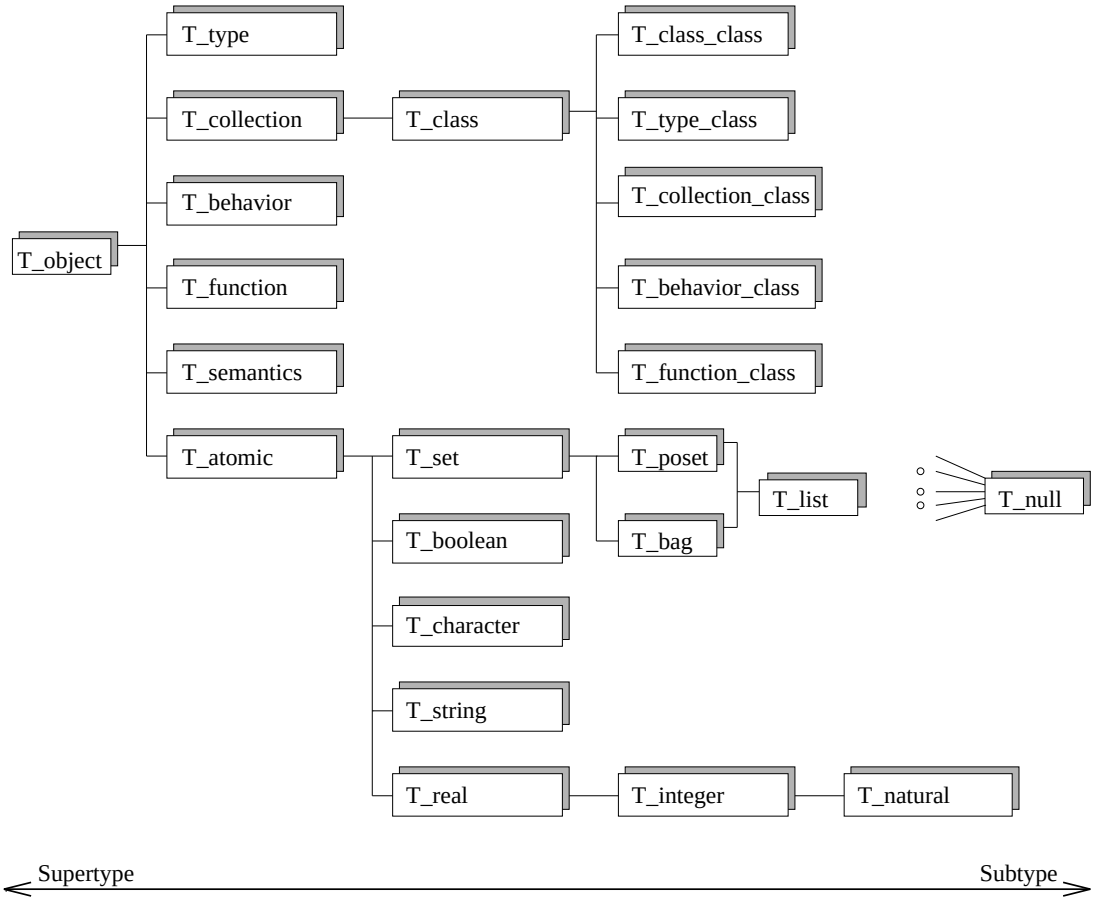


Figure 2.2: Primitive Type Lattice of TIGUKAT

A behavior may be associated with a function which has runtime calls to executable code (*computed* function) or with one that simply references another existing database object (*stored* function). TIGUKAT perceives behavior application as the invocation of some appropriate function regardless of the nature of the function. The distinction between stored and computed functions is of no consequence to the model. Whether a function be stored or computed, a semantic description (behavior) of that function always exists. The model supports any kind of function by abstracting the implementation details of the

function with the semantic consistency of behaviors. We discuss our present implementation approach in Section 3.3.

Sets, bags, strings, reals, etc., are system maintained immutable *atomic types*. From the user's perspective, it is expected that the entire domain of these types exists and may be manipulated using the predefined operations. Maintaining this conceptual abstraction is implementation dependent. It is not feasible to actually maintain every possible real in the extent of the type `T_real` . This is an infinite domain. The solution we have adopted is to automatically generate an atomic object when it is accessed for the first time. Each atomic type has an associated atomic class which groups the instances of that type. It is assumed that instances of atomic types serve as state, identity and reference simultaneously. For example the integer reference 5 would refer to the object 5 which is an instance of `T_integer` whose identity and state is the universally known abstraction of the integer 5. The system recognizes the existence of *one and only one* integer object 5. We discuss the approach we took in the implementation in Section 3.1.

A behavior is an object which when applied to another object performs some operation and produces a resultant object. The uniqueness of a behavior is determined by a semantic expression of its functionality and equality for behaviors is refined to incorporate equality of the semantic expression. There is a behavior, *B_semantics* defined on the type `T_behavior` which returns the semantics of the behavior object it is applied on. We introduced an additional type `T_semantics` whose instances correspond to the semantics of behavior objects. On application of the *B_semantics* behavior the resultant object is an instance of `T_semantics`. Presently, the semantics of a behavior is an aggregation of its name, argument types and return type but in subsequent research a more complete specification is being sought. Introduction of the `T_semantics` type offers a convenient mechanism to introduce as complex a behavioral semantics as desired into the system.

The only form of equality that the model recognizes at the conceptual level is that of *identity equality*. Two objects are considered to be equal if they are the same object, i.e. they have the same oid. There is no notion of *shallow* or *deep* equality as found in other models. These can be attained by customized interpretations through behavior application. These are design decisions which are strictly implementation dependent. However, the model does suggest a specialization of equality on behaviors to mean semantically identical and on each of the atomic types to mean *value equivalence*. The *B_equal* behavior is defined on the root type `T_object`. At this level it verifies identity equality only. Each of the atomic types redefine this *B_equal* to do a comparison of values which they represent. The type `T_behavior` redefines *B_equal* to verify that the two behavior objects being compared possess identical semantics.

Chapter 3

System Implementation

A true object-oriented system should efficiently support the key paradigms of polymorphism, abstraction, encapsulation, message passing, inheritance, instantiation, code reuse and the dynamic binding of functions (methods) to behaviors (messages) [3]. We strive to achieve all of these in our system and this chapter takes an implementor's view at them. The implementation task proved to be a considerably complex and challenging one due to the intrinsic richness and extensible nature of the object model. We describe and discuss the resolution of various design decisions and relative tradeoffs we faced and try to enumerate the alternative choices available at each of these decision points.

The prototype runs on the SunOS operating system on SparcStations and the coding style has been borrowed from Texas Instrument's C++ object-oriented library (COOL) [17]. It was from reading through some of COOL's working source code segments that our ideas for efficient use of C++ classes, polymorphism and inheritance in an application framework were founded. At times when we needed a particular class functionality we examined the COOL source code implementation for a similar class and then proceeded by significantly altering, modifying or decomposing the structure and implementation design to appropriately suit our needs. An outstanding feature of the COOL library is the clarity and simplicity of code accompanied by an abundance of informative comments.

The current size of the executeable code is approximately a Megabyte and the memory used by the primitive object system totals about seventy Kilobytes.

3.1 Data Structures and Mapping

We chose to implement the TIGUKAT object model using C++ [33] (actually GNU's C++ implementation called g++) mainly because of the power and structuring ability an object-oriented programming language offers in modeling real world applications. This being the first attempt at implementation, the prospect of numerous revisions and enhanced versions in the future is imminent. Maintainability of our code is thus of critical importance. An object-oriented language such as C++ offers a high degree of abstraction, information hiding and modularity all of which facilitate code readability and maintenance. Object-oriented design and implementation techniques provide a simple, incremental method of developing otherwise overly complex large software systems. We are of the opinion that a good object-oriented design invariably results in a robust and reliable implementation. However, the relationship between TIGUKAT and C++ ends at this.

The object models and primitive type systems of the two are only remotely compatible. This discrepancy stems from the distinguished manner in which the TIGUKAT model

concisely segregates *classes* from *types* and *behaviors* from *functions*[1]. TIGUKAT has what is called a *generic* object model. Its type system is not an extension of the C++ type system (unlike systems such as ObjectStore [21]). On the other hand, C++ possesses no notion of these distinct abstractions and the only template for object instantiation is that of the C++ class, which is a combination of our types and classes. TIGUKAT uniformly treats all entities in the model as *first class* objects (including types, classes, behaviors, functions and collections). In C++, the *class* template does not exist at run time. The TIGUKAT model perceives types and classes as two disparate entities. A type provides the behavioral template structure for maintaining the characteristics of all its instances (objects) and for behavioral inheritance via *subtyping* while a class serves as a repository for objects of its unique corresponding type and is used for the generation of instances of this type. Object instantiation may be done only through a class and the objects created will conform to the type associated with that class. Every type may be associated with *one* and only one distinct class. Semantic details concerning the heterogeneous grouping construct can be found in [24] under the discussion on *collections*.

As a direct consequence of the separation of behaviors from functions, TIGUKAT deals with behavioral and implementation inheritance as two relatively disjunct mechanisms. Conceptually the model itself fully supports multiple behavioral inheritance (*multiple subtyping*), while implementation inheritance is an entirely implementation dependent issue. The degree of implementation inheritance offered by any particular implementation of the system may vary from *nil* to *full*. Our present implementation provides an intermediate between the two, as discussed in Section 4.3.

The TIGUKAT object model is intended to be the lowest semantically complete level of the system. One may choose to build an OODBS or a OOPL around this kernel. The semantically rich functionality provided by the core model is powerful enough to support arbitrarily complex extensions as may be eventually desired.

We initially proposed to map the primitive type system of TIGUKAT directly into a semantically corresponding C++ inheritance hierarchy. This proposal was ultimately rejected due to its unwanted coherence to and dependence on the C++ object model and inheritance support mechanism. One of the prime features of TIGUKAT is that it is entirely self-definitive. It is not sustained by any form of external metadata or metametadata, like many other systems are. The higher level constructs such as `T_class_class`, `T_type_class` and `T_collection_class` ensure this independence [24]. By mapping TIGUKAT's types directly into C++ classes we stand to lose this prominent feature of the model. Embracement of this approach would have compelled us to be content with C++'s inheritance structure, which we found to be lacking in certain respects.

TIGUKAT strongly mandates that all types defined in the system have the semantics of first class objects. The technique we have employed to accomplish this is to first establish semantically different kinds of C++ object instances in the system viz. *type* objects, *class* objects, *object* objects, *behavior* objects, *function* objects, *collection* objects and *atomic* objects. These template instances differ in their structural contents. For example, a type object has a fixed number of slots dedicated for maintaining information such as its corresponding class (implemented as a reference to another C++ instance which is a *class* object), its subtypes set (reference to a C++ set instance), its supertypes, etc.

Once we had this comprehensive and well-defined semantic mapping established for the

[1] Each TIGUKAT function object is stored in the database itself along with references to its source code component and its compiled binary executable. The interested reader is referred to [24, 25] for complete formal details. In case of any source code modifications, recompilation is required.

complete gamut of different kinds of system objects we went ahead and synthesized the rest of TIGUKAT's primitive object system utilizing these basic components as building blocks.

There exists a single foundation C++ class, TgObject (Appendix A.1), which is the principal template for instantiation of all the other system objects. Every TIGUKAT object (*type* objects, *class* objects, *behavior* objects, *collection* objects, *function* objects, *object* objects, *atomic* objects and other primitive or user-defined objects) is an instance of this same fundamental C++ class. This approach ensures the uniform representation of all objects in the system since they may each be treated as an instance of TgObject. The TIGUKAT type, class, etc. semantics is buried within the TgObject structure. Following this approach, the TIGUKAT model could just as well have been implemented using any programming language that would suffice in building the foundation primitive system.

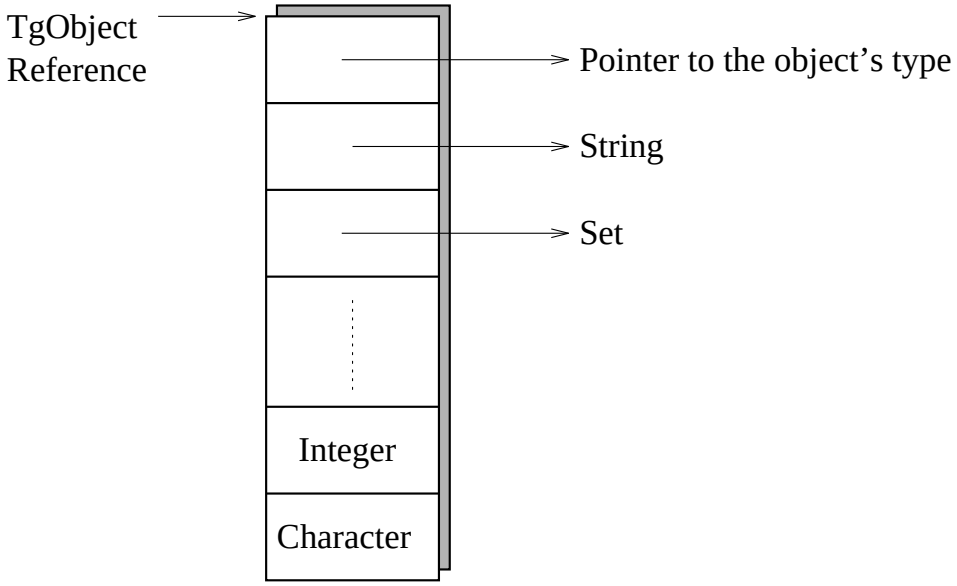


Figure 3.3: Representation of the Generic TgObject Structure

From the structural viewpoint every instance of TgObject comprises an array of records as depicted in Figure 3.3. These can be thought of as the *attributes* (data fields) of that particular instance. TgObject is a dynamic linear array each element of which is a pointer to an instance of the *AttrEntry* facilitator class. The AttrEntry class (Appendix A.1) provides instances each one of which is a four byte *void pointer* and may potentially reference any other object in the system. Member functions of this class provide functionality to set, get and compare instances. Integers and characters are stored in the element slot itself while all other objects, including the atomic objects such as reals, sets, strings, bags, lists and posets, have only references to them stored in the slots. This decision was made to ensure efficient use of memory. Presently we store pointers to floats but intend to use contiguous slots for storing these too. For any object, the first slot always contains a pointer to that object's type which was the template used for its creation. Thus, every object carries knowledge about its type.

Every *type* object, as shown in Figure 3.4, possesses a slot assigned to index into its exclusive row in the *dispatch cache* (Appendix A.2). This cache is implemented as a static matrix (two-dimensional array) of pointers to functions which accept a variable number of TgObject references as arguments and return a reference to a TgObject. The columns of the cache correspond to the unique *method selectors* (an integer mapping defined on each

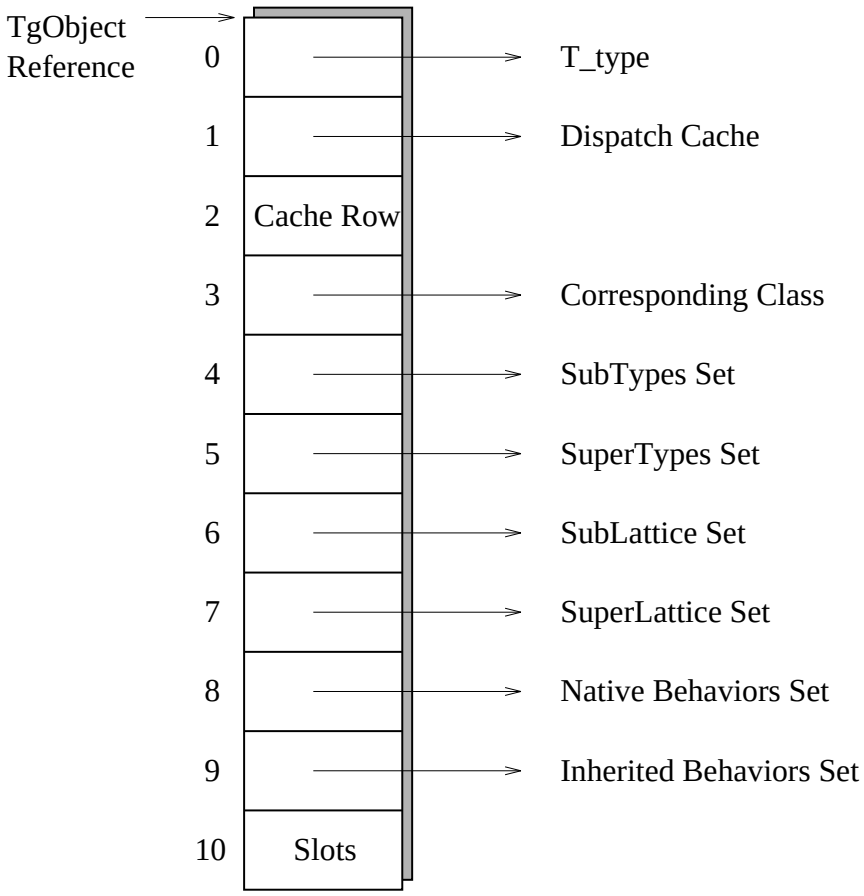


Figure 3.4: The Type Object's Structure

unique behavior object) and the rows correspond to all the types presently existing in the system. The creation of a new type results in the addition of a new row to the cache while new behavior creation adds a column. The schematic of the dispatch cache structure is as shown in Figure 3.5.

With reference to Appendix B.1, it is seen that all type objects (which are instances of the type `T_type` [25]) should support the behaviors in `T_type`'s interface. As shown in Figure 3.4, slots in the type object are either used to store references to the various C++ data structures that hold the necessary information or the required value is itself directly stored in the slot (only for integers and characters). The primitive system types are set up with appropriate contents in the slots so as to reflect a semantic correspondence to the primitive type system (shown in Figure 2.2) and to support the behaviors detailed in Appendix B. Any new types created will obtain their unique information from the argument list passed to the type creating *B_new* behavior defined on the type `T_type_class`.

Similar object structures exist for the class, behavior, function and collection abstractions as depicted in Figure 3.6, Figure 3.7, Figure 3.8 and Figure 3.9 respectively. With reference to Figure 3.7, the *defines set* is the set of all types which define that behavior object in their interface and the *functions list* holds the corresponding functions that implement the concerned behavior over the various types. This structure is maintained as a list because duplicate entries are permitted and there exists an ordering on its elements.

The TgObject class has been implemented as a dynamic data structure supporting the ability to append slots at the end and to insert or delete slots at any location within the

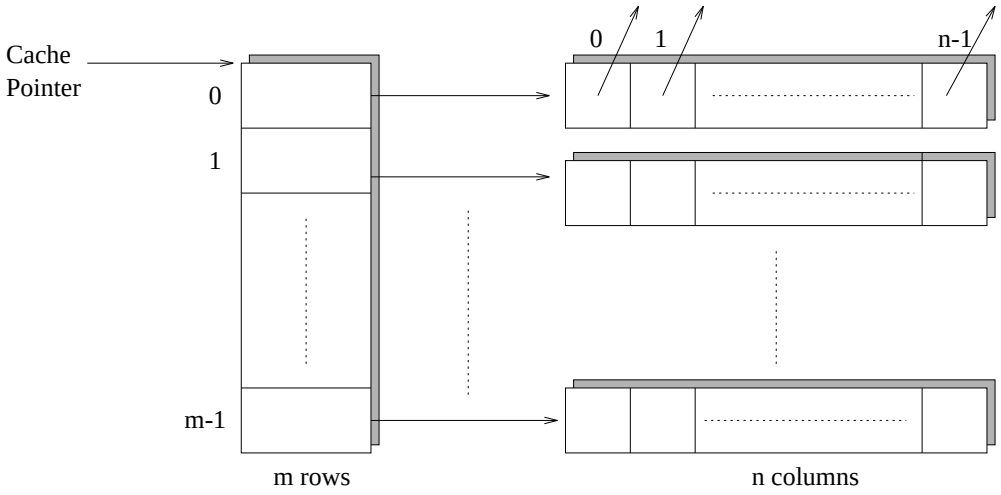


Figure 3.5: Schematic of the Dispatch Cache Structure

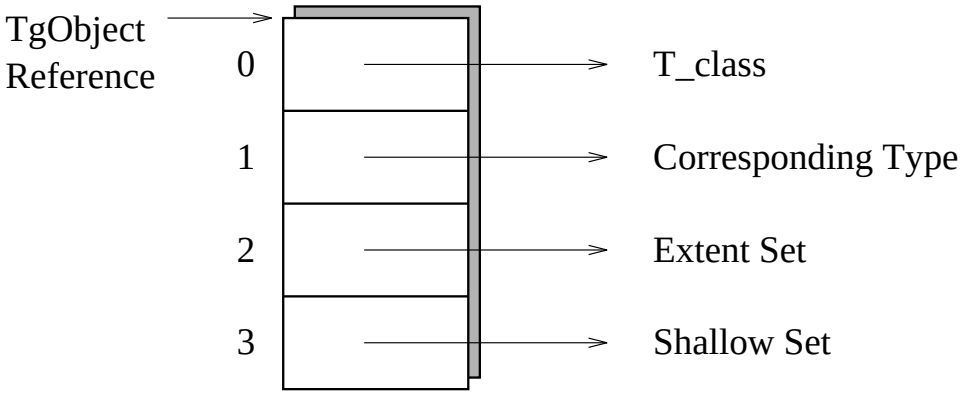


Figure 3.6: The Class Object's Structure

object. This functionality was necessary in order that we could support the reassociation of behaviors with stored or computed functions irrespective of any earlier association. We need to achieve this without incurring the overhead of carrying *empty* (unused) slots[2] in all the objects within the shallow extent[3] of the affected type. Dynamic type changes, updating and schema evolution will require that all system objects be dynamic (possess the ability to grow/shrink as required) in nature. ESM efficiently supports dynamic objects on disk.

TIGUKAT's atomic string, boolean, set, poset, bag and list abstractions have been implemented using the semantically corresponding low level C++ classes while the real, natural, integer and character abstractions borrow the C++ primitive types directly. The mapping though has been kept transparent to the type implementor or user of the system, who only deals with TIGUKAT entities. Other important structures used are the system caches (a dispatch cache, a cache for maintaining information about the nature of the functions that have been associated with the behaviors and a cache for pairing primitive set/get accessor functions), an internally used bag class (differs from the bag class used for the `T_bag` abstraction in that no count of occurrences of an element is maintained),

[2] An empty slot is generated if a behavior previously associated with a stored function is now reassociated with a computed function.

[3] The *shallow* extent of a type comprises those instances which were created using this type as a template.

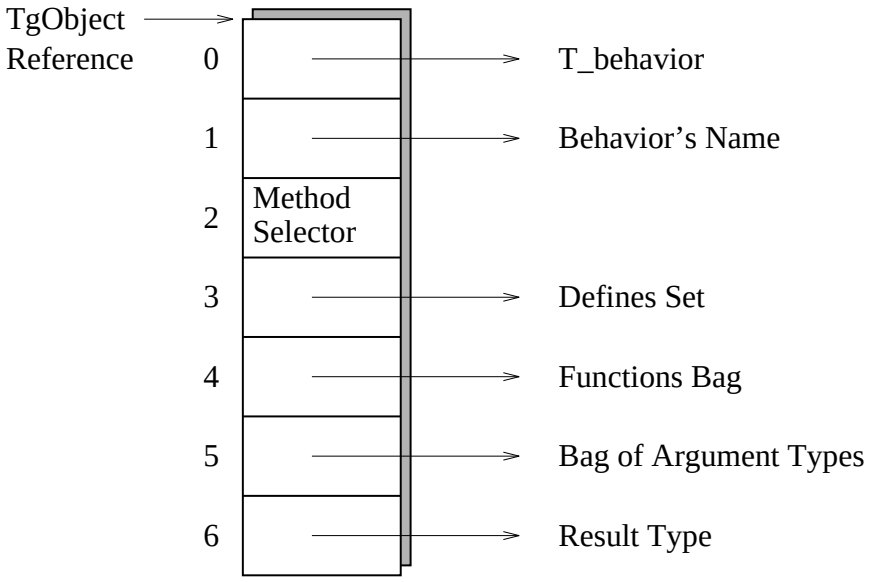


Figure 3.7: The Behavior Object's Structure

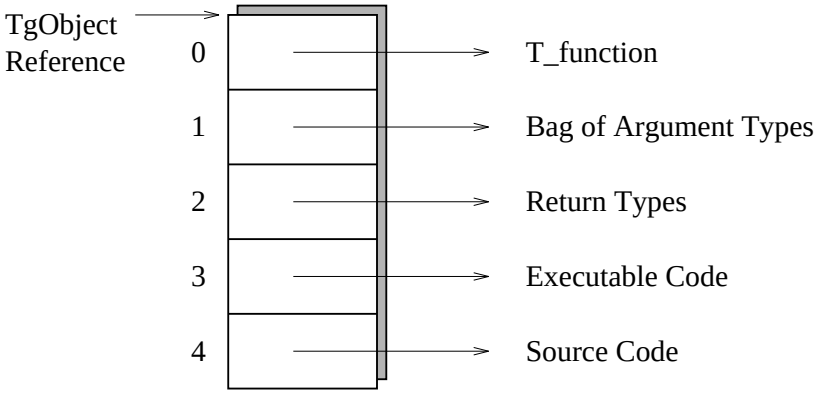


Figure 3.8: The Function Object's Structure

facilitator classes (to provide the slot element abstractions) for the elements of TgObject and the container classes (set, bag, etc.) and routines to generate objects of the atomic types (to be automatically invoked by the system when an atomic object is referenced for the first time). Source code abstracts for the set and bag abstractions are given in Appendix A.3.

Objects of all the atomic types too are uniformly handled as instances of TgObject. Although there is considerable overhead involved here, for lack of a better solution at the present time we decided that the uniform handling of only TgObject references being maintained across all types in the system offered the best initial compromise. The actual existence of these atomic TgObject instances is restricted to that period during which they are passed across other objects or behaviors. Once stored as a component of some other object they lose their typing information and exist as mere references to the underlying C++ structures or counterparts. The structure of an atomic object is shown in Figure 3.10.

Atomic object creation is not supported by the model [24]. Every atomic object ever needed is assumed to be in existence and always available when required. This is perfectly logical at the conceptual level but does not fit in gracefully at the implementation level.

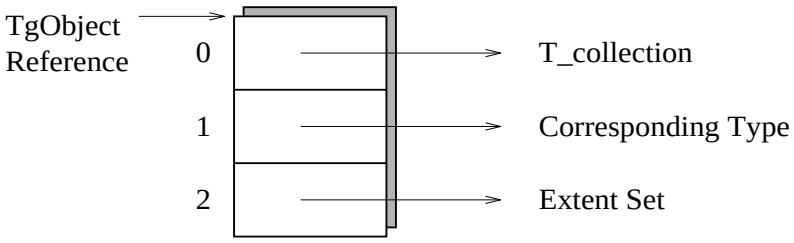


Figure 3.9: The Collection Object's Structure

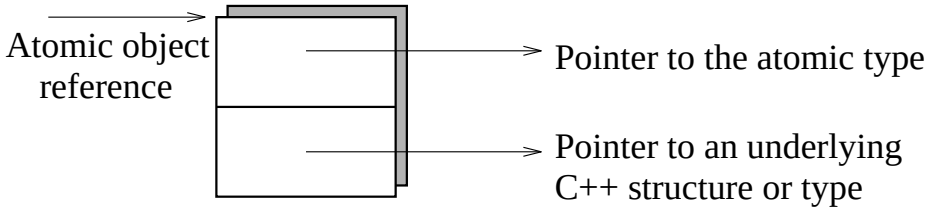


Figure 3.10: The Representation of an Atomic Object

We eventually handled atomic object creation by a special set of macros which are invoked when an atomic object is referenced for the first time. Contingent on which atomic type object is required, the appropriate macro is invoked, which generates the required object and returns a reference to it. A considerable amount of research is still required before we can propose a more elegant solution to this problem. This would involve the design of a convenient runtime environment other than that of C++. For the moment we are content with using this.

The `T_function` type object defines in its interface a behavior called *B_executable* which returns a reference to an executable piece of code. Type `T_function` also defines a behavior called *B_execute* whose operational functionality is to access the piece of code to be executed and to execute it using the list of arguments supplied. All arguments required must be passed to these behaviors. In the implementation we use a number of functions which accept a variable number of arguments to accomplish this. Thus we have a pretty straightforward implementation of functions as objects.

The instantiation of a new object summons an updating of all extents in the object's type's superlattice. One way to accomplish this is to simply mark all the supertypes whose sublattices and class extents are presently not current and need updating in order that they accurately reflect the instantiation of the new object. The actual update is deferred to a later time, maybe at the end of the current session or when the system is not in use. We proposed this mechanism since it would not inhibit system speed during object creation. It is trading the efficiency with which types are created to obtain efficiency in object instantiation since type creation is expected to occur less frequently than general object instantiation.

Another alternative here is maintaining only the *shallow extents* and when the *deep extent*[4] is required it is computed as an union of the shallow extents of all the subtypes. This proposal too would have a slowing effect, especially during querying. Presently we adopt the approach of performing all the computation immediately during object creation

[4]The *shallow extent* of a class includes only those objects instantiated using this class' type as a template while the *deep extent* comprises all the objects in the shallow extents of the class as well as those in each of its subclasses.

and storing the deep extent as a stored value.

3.2 Behavior Application

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

Dispatching may be considered as a special case of what is called *resolution* [37]. *Resolution* has been defined as a runtime interpretation process that selects a particular value from a possibly ambiguous set of values. Method dispatch (behavior application), hence, seeks to select an appropriate function object (method) whose code needs to be executed, from a set of function objects each of which implement the same named behavior object over different types. In order to correctly make this decision some additional information (actual type of the receiver and the method selector) relevant to the context is required.

Any efficient implementation of an object-oriented system based on the message passing paradigm needs to give due consideration to the time and space tradeoffs inherent in a method dispatcher. We have opted for a relatively simple but fast mechanism at the cost of bearing the consequential memory overhead. The system maintains a *dispatch cache* which consists of a slot for each behavior-type pair that exists in the system. This cache is a statically allocated volatile structure which needs to be reinitialized on program startup. The size of this lookup table is accordingly proportional to the total number of unique behaviors in the system and the total number of types in existence. We sacrifice memory usage for quick response time during execution, but as proposed in [2, 15], an incremental coloring algorithm would help reduce this excessive memory consumption drastically. We have not implemented this optimization in this version.

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

Behaviors may be reassociated with functions at any time (redefinition of behaviors) and this makes it imperative that we support the dynamic binding of behaviors and perform dispatch *on the fly*. Although it is evident that *static* (compile time) dispatching is more efficient [12], this will seldom be possible in our system. The reference to an object of a particular type may potentially be referencing an object of any of this type's subtypes. The ambiguity about which function should be invoked can only be resolved at runtime when knowledge about which type's instance is being referenced becomes available. Thus, the actual type of a receiver object needs to be identified prior to function execution. We

note that although dynamic binding might render static type checking difficult it does not entirely preclude it.

To elucidate this further consider the following. The system is *statically typed*[5], which implies that all typechecking is performed at compile time. In most cases the compiler (or preprocessor) will be capable of statically determining which function needs to be executed for the application of a certain behavior on a particular object. This, however, will not always be possible. With reference to Figure 3.11 and Example 3.1, consider an object of some type, say `T_mango`, which may either be accessed through a pointer of type `T_mango` (pterM) or a pointer of one of its supertypes, say `T_fruit` (pterF), depending on the outcome of some particular condition.

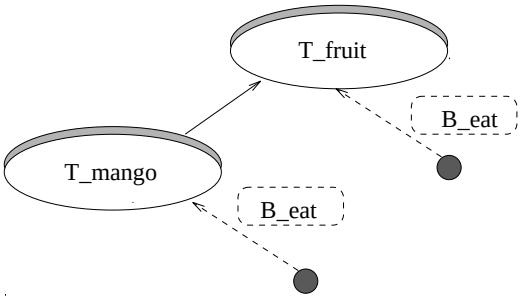


Figure 3.11: Casting References

Example 3.1

```
T_mango* pterM = B_new(C_mango);
if (someCondition) then
    T_fruit* pterF = B_new(C_fruit);
else
    T_fruit* pterF = (pterF*) pterM;
 B_eat(pterF);
```

This ambiguous situation is arrived at because we permit the *casting* of a reference to some `T_mango` type object into a reference to a `T_fruit` type object[6]. It is now impossible for the compiler to statically determine which row of the cache needs to be accessed to jump to the appropriate executable code when the behavior *B_eat* is applied to the object presently being referenced via the `T_fruit` pointer. This has to be done at run time following a check on the receiver's *actual* type.

Let us take a cursory look at how method dispatch is implemented in the C++ and Smalltalk-80 systems. In C++ terminology, a *virtual function* is a member function which is defined in some base class but may potentially be replaced in each of the derived classes by any alternative function which has a matching semantics [16]. Virtual functions may be defined for any class and the C++ system implementation maintains a *virtual table* for every class that has at least one virtual function. This table is a direct jump table and selectors are specific to the objects for which that table applies. Each object of a class that has virtual functions needs to maintain a pointer to that class' virtual table. To handle multiple inheritance in the presence of virtual functions the tables are segmented with

[5]Static typing should not be confused with *strong typing*, which we interpret as the requirement that every single system object has information about its type.

[6]The analogy here is to permit viewing a mango as a fruit, since all mangoes are after all fruits.

each segment corresponding to one of the superclasses. The proper offset into the table is computed at runtime during function dispatch, to get to the address of the appropriate code to be executed. This offset depends on the class of the receiver object [33].

The Smalltalk-80 system maintains a structure called the *method dictionary* for each class. When an object receives a message the dispatcher checks the method dictionary of the receiver's class for the presence of the method selector, visiting the dictionaries of the superclasses of that class, each time the check fails [18]. This process will terminate with failure if the root class' method dictionary is finally searched and the required method is not found. It is interesting to note here that Smalltalk-80 is a language with untyped variables and object references and inferring types at compile time is difficult. This makes runtime typechecking mandatory.

In our system types exist as objects and every object has knowledge about its type. Behavior application on any object is via the type of this receiver object. The type has knowledge (a pointer) about its corresponding row in the dispatch cache. Our inheritance structure is different from that used by C++. The reasoning behind this is that since behaviors in the system are the equivalent of C++'s virtual member functions we would need to maintain a virtual table for each and every type in the system. That would be extremely inefficient in memory usage and needlessly convoluted. The reason the C++ implementation can afford to do this is that in any one application they expect the percentage of types which have virtual functions to be negligible. In this respect C++ is not considered to be a *pure* object-oriented language in comparison to a language like Smalltalk-80 since all functions *can not* be redefined down a type hierarchy.

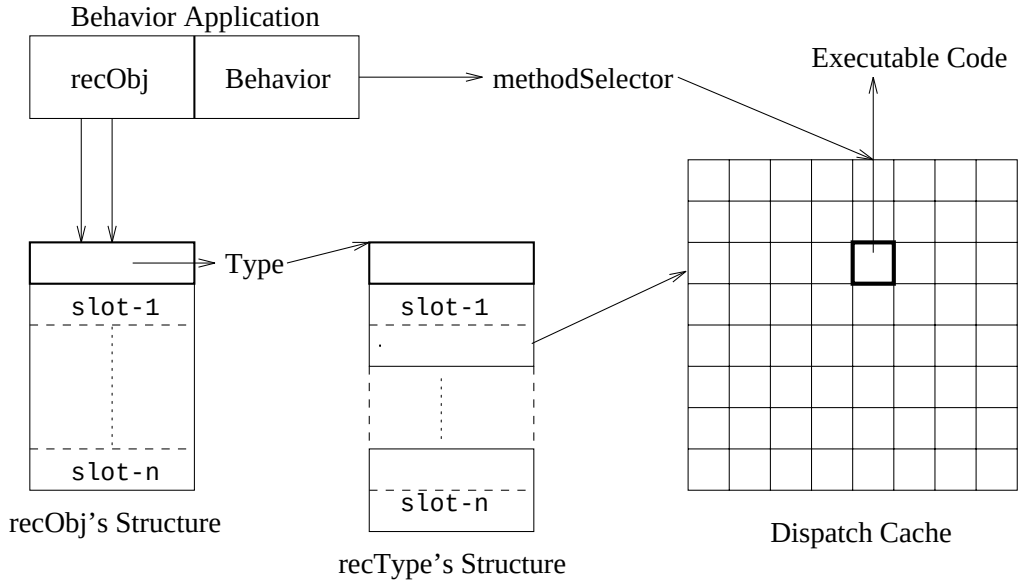


Figure 3.12: The Behavior Application Process

The behavior application process in TIGUKAT involves the following procedure. With reference to Figure 3.12, given an object, *recObj*, the receiver of a particular message, we extract its type (the *B_mapsto* behavior defined on `T_object`), *recType*, which is readily available since every object knows its type. All types have knowledge of the unique cache row that they correspond to. From the applied behavior object we extract the method selector, *methodSelector*. This integer value indexes into a unique column in the dispatch cache. Having arrived at the desired slot, the address of the executable code stored in it,

which serves as a pointer to the entry point of the function, may now be accessed and executed. The list of arguments passed to the behavior is carried over to function execution after relevant typechecking is done[7]. Behavior application is conveniently reduced to the execution of a single line of code:

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

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

The only behaviors applicable on an object are those which are defined in the object's type's interface set. The behaviors in this set might have either been inherited from the supertypes or passed as an argument set to the type creating *B_new* behavior defined on the type `T_type_class` (refer to the Appendix B.1). Irrespective of how membership in the interface set was attained, it is mandatory that each of the behaviors in this set be associated with some function object before class creation. We permit the reassociation of behaviors after class creation and this does not cause any complications so long as the behavior is being reassociated with a *computed* function. If the desired reassociation is with a *stored* function, then that function would expect a slot in all the objects in the extent of the types for which it has been defined. In this implementation we do not support the reassociation of a behavior with a stored function once the type has a corresponding class associated with it and thus possesses the potential for object creation. We have proposed a suitable algorithm for achieving this (Section 4.3) and intend supporting this feature in a later version.

Functions are treated as first class objects being instances of the type `T_function`. Although the only difference between two stored functions which implement the same behavior down a type hierarchy may be the slot in the object to be accessed, the function is still considered to be redefined. This redefinition is transparent to the type implementor and does not require any intervention on his part. The system will automatically handle this change in slot access during class creation.

3.3 Stored and Computed Functions

The TIGUKAT object model is a behavioral model and has no notion of attributes. All object creation, access and manipulation is achieved purely by the application of behaviors on objects. No concept of an *attribute* exists. Consider that some behavior is associated with a *stored function*. On invocation, that function will require to access a memory location (data field or slot) within the physical structure of the object it was invoked on. The function either places an *attribute*[8] into this slot or retrieves one from it. The stored function accesses the concerned memory location via primitive system provided *set* and *get* accessor functions. For accessing any particular slot a set-get pair is created. This pairing is applicable on a type basis and is maintained in an auxiliary cache structure (SC2 in Figure 3.17).

[7] Although the association of a behavior object for a particular type is with an instance of `T_function`, the actual address of the executable code is placed into the appropriate cache slot by the behavior association process. This decision seeks to eliminate the extra level of indirection (pointer chasing) via the associated function object.

[8] Since every object in the system is potentially a *composite* object, the term *attribute* implies a reference to some other system object. Contained objects are accessed via these stored functions.

Although the object model does not distinguish between stored and computed functions at the conceptual level this distinction needs to be addressed in the implementation. The uniformity of functions is maintained at the upper levels but not when viewed by the type implementor. This concept of distinguishing the stored and computed functions is a purely implementation dependent issue and is concealed at the implementation level. Only stored functions will require the existence of physical slots (attributes) in all instances of those types which support behaviors associated with a stored function. All objects are instances of some type that possesses one or more behaviors in its interface. Those behaviors which have been associated with stored functions will require as many slots in their physical structure as there are *stored behaviors*[9] in the type's interface (in addition to the single slot that references the object's type, which is a universal requirement for all system objects). The information about the total number of slots required in each of the type's instances is maintained in the *slots* field of that type object and is accessed during application of the object creating *B_new* behavior defined on `T_class`.

An alternative for overcoming the slot access problem is to maintain "named" attributes in a property list and to perform a search on this list, with the attribute *name* being the key, whenever access is desired. The model supports no concept of attributes and since *named* attributes would violate the behavioral nature of the model, this is not an acceptable option.

The approach we have taken is as follows. When a new type is created its inherited behavior set contains those behaviors which are inherited from its specified direct supertypes. Information about which of these behaviors are associated with stored functions in the supertypes is maintained in an additional data structure we call the *supplementary cache*. This is a matrix similar to the dispatch cache (SC1 in Figure 3.17). For every entry in the dispatch cache there is a corresponding entry in the supplementary cache used to indicate the stored or computed nature of the associated function. The supplementary cache makes its information content available to the type creation process. The implementation inheritance mechanism (part of the type creating process) extracts all the relevant information from the supplementary cache and attempts to resolve which of the inherited behaviors should be associated with stored and which with computed functions in the new type. Reassociation of these behaviors is permitted (irrespective of whether it is with a stored or computed function) until class creation time, following which associations with stored functions is prohibited.

An alternative to maintaining a separate cache is to mask the least significant bit of the function pointers stored in the dispatch cache and use it as an escape bit to indicate the nature of the function it addresses. Since there is a possibility that pointers might be corrupted, we have not implemented this optimization in this version.

The *native* behavior set consists of those behaviors which are passed in the argument set of behaviors required by the type creating *B_new* behavior defined on `T_behavior_class`. These might later be associated with stored or computed functions (application of the *B_associate* behavior defined on `T_behavior`). The nature of the function that every behavior is presently associated with is maintained in the supplementary cache. A union of the *interface* set of behaviors of the type's direct supertypes is inherited into the *inherited* Set. Only the stored functions require slots to be created in each of the instances of the new type. It is possible that some behaviors which were previously associated with stored functions in one or more of the direct supertypes are now to be reassociated with a computed function. If this information about the switch in association is available prior to object instantiation for the new type, no overhead is incurred since the need to carry any empty

[9]Simply, any behavior that has presently been associated with a stored function.

slots is eliminated.

When a behavior is associated with a function for a particular type (*B_associate*) the behavior association process detects whether it is a stored (set or get) or computed function and places the relevant indication in the corresponding supplementary cache entry for that behavior.

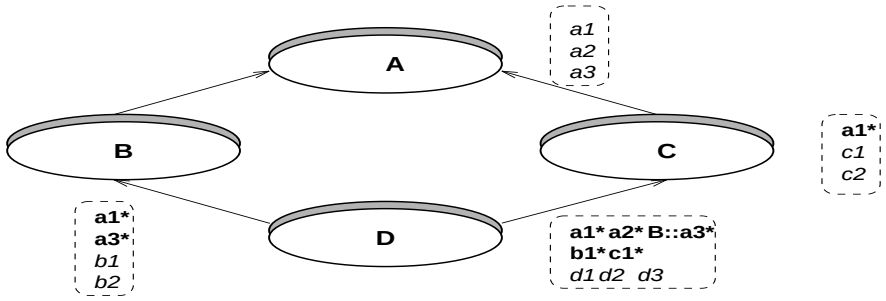
Now, if we were to permit the reassociation of a behavior with a stored function for a type which already has a class (and possibly some instances) we are faced with the problem of determining which slot should that function be accessing. The association of behaviors with computed functions is permitted and easily accommodated because the structure of the instances of that type will not be affected. But, if the behavior is associated with a stored function, the existing object structures do not possess that extra slot, as would be expected by that function. This is the reason why we prohibit reassociation of a behavior with stored functions once the type has a class and as a consequence it now possesses the potential for object creation (and in fact may already have some instances).

For this first implementation we have decided to carry the overhead of an *empty* slot if a behavior initially associated with a stored function is reassociated with a computed function. In the future, however, we will permit any kind of reassociation: switch from computed to stored or from stored to computed. The computational overhead involved will be an iteration through all objects in that type's shallow extent. The present implementation has provided for this enhancement to be easily incorporated chiefly by providing dynamic slot allocation and deallocation capabilities in the fundamental TgObject class. The proposed mechanism will iterate through all the object's in the shallow extent of that type's class and for each object in it allocate an extra slot. This may be appended to the end of the object since particular location is of no significance until a class is created. *Marked* slots could be kept around. A slot is said to be *marked* if a behavior which was previously associated with a stored function is reassociated with a computed one, making that slot redundant. The marked slot may then be reused to accommodate more recent reassociations of behaviors with stored functions. In case the instances of that type already possess a marked empty slot, then that slot may be conveniently and safely reused. Presently, although a reassociation is permitted from stored to computed, all objects which may be instantiated after the reassociation will still carry the extra empty slots. This is due to the type's knowledge of slots which may not be conveniently modified.

The example depicted in Figure 3.13 shows an inheritance graph with multiple subtyping, the resulting object structure with stored and computed functions accounted for and the dispatch cache as it would appear for this simple system. Behaviors marked with an asterix are those which have been reassociated or redefined down the type hierarchy. The behaviors *a1*, *b1*, *c1* and *d1* are associated with stored functions.

With reference to Figure 3.13, the types B and C each have the type A as their single supertype, while the type D inherits from multiple supertypes B and C. The object instance structures show the appropriate number of slots required for access by the stored functions associated with some of the behaviors in their respective types.

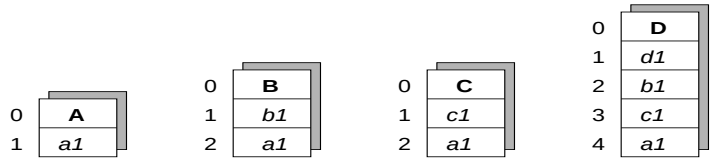
An X in the dispatch cache indicates a unused entry, an integer value represents the slot number to be accessed by the stored function while a behavior name represents the address of the executable code. The behavior *a1* when inherited by types B and C is considered to be redefined (shown marked with an asterix) because the slot access number is different from that used in the type A (as can be seen from the instance structures). Behaviors *b1* and *c1* in type D are marked with an asterix for the same reason, but behavior *a3* is shown marked in type B because its association is with a different function than it was associated



Inheritance Graph

Behs / Type	a1 1	a2 2	a3 3	b1 4	b2 5	c1 6	c2 7	d1 8	d2 9	d3 10
A 1	1	a2	a3	X	X	X	X	X	X	X
B 2	2	a2	**a3***	1	b2	X	X	X	X	X
C 3	2	a2	a3	X	X	1	c2	X	X	X
D 4	4	**a2***	**a3***	2	b2	3	c2	1	d2	d3

Dispatch Cache



Object Instances

Figure 3.13: Stored and Computed Functions

with in type A. Finally, implementation inheritance in type D required intervention from the type implementor to resolve the conflict for the behavior *a3*. The choice of code to be used is that from the type B (depicted as B::a3∗).

As can be seen in the figure, it is possible that more than half the cache may be empty. This results in a sheer waste of space. We intend to alleviate this waste by using an incremental coloring scheme that allows rows of the cache to be shared by multiple types in effect reducing the overall cache size [2, 15]. Cache slot assignments will be made using a coloring algorithm which searches for an optimal assignment scheme. Since the graph coloring problem has an exponential complexity, the resultant cache will seldom be the optimal one. However, the application of heuristics help obtain a quick solution which is satisfactorily close to the optimal one. Space reduction estimates of up to sixty percent are deemed possible. This reduction is dependent on the number of types whose interfaces have a partial overlap due to subtyping or specialization.

We presently prohibit subtyping any type which still has behaviors which have not yet been associated with any function for that type. We also considered the following implementation alternative. For a function object three kinds of implementations could be possible, stored, computed and *don't know*. During type creation the requisite, possibly empty, set of behaviors passed as arguments to the *B_new* behavior for type creation, are by default *don't knows*, that is, they are not associated with any implementation as yet, stored or computed.

These behaviors *must* be associated with a particular implementation (function) before the type is considered to be *fully* defined. Any type which still has a *don't know* is said to be *functionally incomplete*. It is behaviorally defined, but functionally undefined. Class creation is disallowed for such an incomplete type and thus no objects may ever be instantiated of this type till it has been entirely defined.

However, one may choose to subtype this partially defined type, inheriting the *don't knows*, but also inheriting the non-instantiation feature. All *don't knows* for the subtype may then be resolved via association with implementations and we may create objects. The supertype might still possess the *don't know* behaviors and continue to be regarded as an incomplete type. This is analogous to the notion of *abstract super types*, as defined in other models including Smalltalk. The type exists solely for achieving behavioral inheritance, but no objects may ever be created of that type.

All slot accesses (the offset number of the field which is to be accessed in the object structure) are reassigned during the creation of a class for a new type. A single slot number is assigned to each matched pair of the set-get primitive accessor functions associated with a corresponding pair of stored behaviors defined on that type. Class creation (Figure 3.14) marks the *deadline* for association of any of the type's behaviors with stored functions. The class creation process iterates through all behaviors associated with stored functions and for every pair of accessor primitives discovered it allocates a unique slot which they will access in the objects that may now be instantiated for this new type.

Recapitulating briefly, class creation implies the potential for instantiation, which in turn requires structural knowledge about the objects to be instantiated. Therefore, since reassociation of a behavior with a stored function might result in structural modifications to existing objects it is not sanctioned once a class exists for that particular type. In later versions we will permit the dynamic reassociation of behaviors with stored or computed functions at any time. This is presently being studied in conjunction with schema evolution and update semantics.

3.4 Behavioral and Implementation Inheritance

Two kinds of inheritance are supported by our system, *behavioral* and *implementation* inheritance. Behavioral inheritance (subtyping) is the simpler and more intuitive of the two. This has been precisely conceived at the conceptual model level. The implementation strategy involves taking the union of the interface sets of all the types declared as immediate supertypes of the new type being created . This set forms the contents of the new type's *inherited* set and comprises the minimum set of behaviors that all objects of this type should conform to. The nature of the functions that these behaviors have been associated with is of no consequence to the behavioral inheritance mechanism. The algorithm implemented (shown in Figure 3.15) iterates through the relevant interfaces and selects all the behaviors with unique signatures as candidates for insertion into the new type's inherited set . This is a relatively straightforward technique.

Implementation inheritance is an entirely implementation dependent feature which facilitates code resusability by ensuring that all code be at a level at which the maximum number of types can share it [3]. It can get arbitrarily complex depending on the degree of behavioral conflict which occurs among the declared direct supertypes. If only single inheritance is present the inherited set of the new type is precisely the contents of the interface set of its sole supertype. No conflict resolution is necessary and all entries in the dispatch cache and the supplementary cache are merely duplicated in the row allocated for the new type

Algorithm 5.1 *(Class Creation Algorithm)*

```
begin
  input T : the type for which a class is to be created;
        DC : dispatch cache;
        SC1 : supplementary cache for stored and computed information;
        SC2 : supplementary cache for pairing the primitive accessors;
  var interfaceSet : set of behaviors defined on the type;
      slotCount : integer count of slot numbers;
      methSelector : integer index into DC's columns;
      cacheRow : integer index into DC's rows;
      pair : integer value which indicates accessor's counterpart function;
  interfaceSet ← B_interface(T);
  slotCount ← 1;
  cacheRow ← row in DC for T;
  for each interfaceSet_i ∈ interfaceSet do
    begin
      methSelector ← column in DC for interfaceSet_i;
      if SC1(cacheRow,methSelector) ≠ stored ∧
        SC1(cacheRow,methSelector) ≠ computed then
          begin
            class creation failure;
            incomplete behaviors in type T;
          end
      else if SC1(cacheRow,methSelector) = stored then
          begin
            pair ← SC2(cacheRow,methSelector);
            if DC(cacheRow,methSelector) = NULL then
              begin
                DC(cacheRow,methSelector) ← slotCount;
                DC(cacheRow,pair) ← slotCount;
                slotCount ← slotCount + 1;
              end
            endif
          end
      endif
    end
  endfor
  update slots field in T;
end
```

Figure 3.14: The Class Creation Algorithm

Algorithm 5.2 *(Behavioral Inheritance Algorithm)*

```
begin
  input ST : set of direct superTypes;
  var candidateSet : interface set defined on type ST_i;
      inherSet : inherited behaviors set for new type;
      addBehavior : boolean
  candidateSet ← φ;                                                        (1)
  inherSet ← φ;                                                            (2)
  addBehavior ← TRUE;                                                      (3)
  for each ST_i ∈ ST do
    begin
      candidateSet ← B_interface(ST_i);                                    (4)
      for each candidateSet_i ∈ candidateSet do                            (5)
        begin
          for each inherSet_i ∈ inherSet do                                (6)
            begin
              if B_semantics(candidateSet_i) = B_semantics(inherSet_i) then (7)
                begin
                  notify about possible Conflict Resolution;               (8)
                  addBehavior ← FALSE;                                     (9)
                  next candidateSet_i;                                     (10)
                end
              else
                  next inherSet;                                           (11)
              endif
            end
          endfor
          if addBehavior = TRUE then                                       (12)
            inherSet ← inherSet ∪ candidateSet_i;                          (13)
          endif
        end
      endfor
    end
  endfor
end
```

Figure 3.15: The Behavioral Inheritance Algorithm

for the complete set of inherited behaviors, as shown in Figure 3.16. Each of the inherited behavior's internal information structures (the *defines* set and the *functions* bag) are also updated appropriately to reflect the current state. This implies that all implementations for the inherited set of behaviors are inherited too. However, the type-implementor has the liberty to reassociate any or all of these inherited behaviors.

The purpose of an implementation inheritance mechanism is to make things as convenient for the type implementor as possible. In the worst case absolutely no implementation inheritance may be possible by the system. This situation occurs when the new type's specified multiple direct supertypes have interfaces which happen to be perfectly identical but all of the behaviors conflict with respect to their associations with functions. We note that such an occurrence is in fact rare and would reflect a poor design on the part of the type specifier[10]. It will now be the type implementor's responsibility to resolve all conflicts and reassociate behaviors. A system generated message requesting intervention is displayed and the new type is considered to be functionally incomplete until bindings are reestablished.

Figure 3.17 depicts an inheritance graph with multiple subtyping. The arrows indicate a subtyping relationship between the types shown by ovals and the dotted arrow indicates an instance of the type. The dashed boxes contain the interface sets of the corresponding types while the matrices shown as DC, SC1 (maintains the stored or computed nature of the associated functions) and SC2 (contains the pairing information for all accessor function pairs) are the dispatch cache and the two auxiliary caches, respectively.

This inheritance structure has a clash in behaviors that the system is unable to resolve automatically and requires the type implementor's intervention. The Conflict resolution policy fails because the behaviors *B_setName* and *B_getName* are defined in the interfaces of both the direct supertypes (`T_person` and `T_student` are immediate supertypes of `T_ta` and have conflicting implementations associated in each of these types, being computed in `T_student` but stored in `T_person` (indicated by the bracketed *s*'s or *c*'s). We have assumed that the type implementor opted for the stored implementations to be inherited and therefore each instance of `T_ta` requires a total of three slots.

Consider the scenario when multiple direct supertypes (multiple subtyping) have been specified. The inheritance algorithm employed is shown in Figure 3.18. We iterate through each of the behavior objects in the new inherited set which has been generated during behavioral inheritance. If that particular behavior exists in no more than one of the supertype's interfaces that would imply a conflict-free condition and thus no necessity of performing any conflict resolution. The implementation for that behavior may be safely inherited together with the property of its associated function being either stored or computed. The appropriate entry in the supplementary cache, indicating a stored or computed association, is inserted. If the association is with a computed function then the address of that function is also inserted into the dispatch cache. All the stored functions will possess a NULL entry in the dispatch cache until class creation time. At that time slots will be assigned to all the stored functions, one slot per pair of set-get accessors (Figure 3.14). Note that although the function may have been accessing a certain slot in the supertype that slot access value will no longer be valid in the new type. Reallocation of these slot accesses is entirely system managed.

For each clash in behavior (which occurs as a result of some particular behavior object

[10]By *type specifier* we mean the person who designs the inheritance hierarchy for the user application. The *type implementor* is the one who actually implements this required hierarchy using TDL. He is the one who has authority to create new types, classes, functions, behaviors, etc. The *user* is the end user who may query the existing system and instantiate new objects, but may not be authorized to modify the existing type structure.

Algorithm 5.3 *(Implementation Inheritance Algorithm)*

```
begin
  input ST : set of direct superTypes;
        IS : inherited set of behaviors in the new type;
        DC : dispatch cache;
        SC1 : supplementary cache for stored and computed information;
        SC2 : supplementary cache for pairing the primitive accessors;
  var candidateType : one of the types from the direct superTypes set ST;
  candidateType ← φ;                                                    (1)
  if cardinality(ST) ≥ 2 then                                           (2)
    begin
      multiple inheritance algorithm;                                   (3)
    end
  else begin
      candidateType ← ST_i;                                             (4)
      for each IS_i ∈ IS do
        begin
          if SC1(candidateType, IS_i) = computed then                   (5)
            begin
              SC1(newType, IS_i) ← computed;                            (6)
              DC(newType, IS_i) ← DC(candidateType, IS_i);              (7)
            end
          else if SC1(candidateType, ISi) = stored then                 (8)
            begin
              SC1(newType, IS_i) ← stored;                              (9)
              DC(newType, IS_i) ← NULL;                                (10)
              SC2(newType, ISi) ← SC2(candidateType, IS_i);            (11)
          else begin
              IS_i has an incomplete implementation;                   (12)
              Inheritance failure;                                     (13)
            end
          endif
        end
      endfor
    end
  endif
end
```

Figure 3.16: The Implementation Inheritance Algorithm

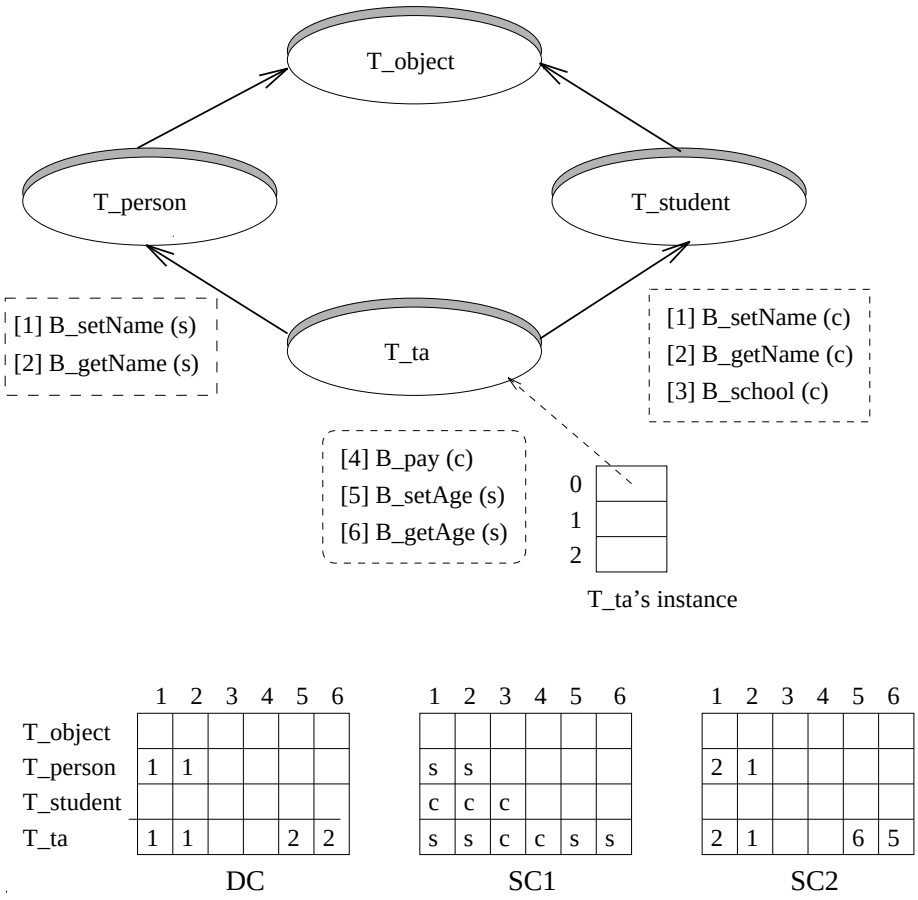


Figure 3.17: Implementation Inheritance Requiring Conflict Resolution

having a signature identical to that of a behavior in the interface of one or more of the other types in the new type's specified direct supertypes) the conflict resolution policy has to be applied. The supplementary cache values for that behavior are examined. If they happen to indicate a consistent *computed* for all the concerned supertypes, the values of the addresses of the functions from the dispatch cache are then examined. If these too are identical for each of the types in the supertypes set, then this behavior's implementation is safely inherited and the corresponding address (will be identical in all the entries) is inserted into the dispatch cache. A *computed* indication is placed in the supplementary cache.

In case it has been found that there is a consistent *stored* in all the entries for the supertypes, the corresponding value of `T_function` is examined for each type and if these too match perfectly, then a *stored* indicator is placed in the supplementary cache and a NULL is entered into the dispatch cache. Note that for all the stored functions, the dispatch cache will hold the corresponding slot number to access (an identical value for each paired set-get) instead of containing the address of the executable code. These slot numbers will only be inferred and allocated during class creation, at which time we will be in a position to determine the total number of all the associated stored functions, both inherited and natively defined for that type (Figure 3.14).

In the event that an inherited behavior is associated with a stored function in one of the supertypes and a computed function in another, or there is mismatch in the values of function pointers then no conflict resolution is possible by the system and a NULL is entered in both caches. It is the type implementor's responsibility to associate this behavior

Algorithm 5.4 *(Implementation Inheritance for Multiple Supertypes)*

```
begin
  input ST : set of direct supertypes;
        IS : inherited set of behaviors in the new type;
        DC : dispatch cache;
        SC1 : supplementary cache for stored and computed information;
        SC2 : supplementary cache for pairing the primitive accessors;
  var confTypeSet : set of supertypes in which a behavior conflicts;
  confTypeSet ← φ;                                                          (1)
  for each IS_i ∈ IS do                                                     (2)
    begin
      for each ST_i ∈ ST do                                                 (3)
        begin
          if IS_i ∈ B_interface(ST_i) then                                  (4)
            confTypeSet ← confTypeSet ∪ ST_i;                               (5)
          endif
        end
      endfor
      if cardinality(confTypeSet) = 1 then                                  (6)
        similar to single inheritance                                       (7)
      else begin
        if ∀ confTypeSet_i ∈ confTypeSet                                    (8)
          SC1(confTypeSet_i, IS_i) = computed ∧                             (9)
          B_impln(confTypeSet_i,IS_i) = B_impln(confTypeSet_{i+1}, IS_i) then
            begin
              SC1(newType,IS_i) ← computed;                                 (10)
              DC(newType,IS_i) ← DC(confTypeSet_i,IS_i);                    (11)
            end
        else if ∀ confTypeSet_i ∈ confTypeSet                               (12)
          SC1(confTypeSet_i, IS_i) = stored ∧                               (13)
          B_impln(confTypeSet_i,IS_i) = B_impln(confTypeSet_{i+1}, IS_i) then
            begin
              SC1(newType,IS_i) ← stored;                                   (14)
              DC(newType,IS_i) ← NULL;                                      (15)
              SC2(newType,IS_i) ← SC2(confTypeSet_i,IS_i);                  (16)
            end
        else begin
              Conflict Resolution Failure;                                  (17)
              Type Implementor's Responsibility;
            end
        endif
      endif
    end
  endfor
end
```

Figure 3.18: Implementation Inheritance for Multiple Supertypes

with an appropriate implementation of his choice or to specify which of the supertype's implementations is to be inherited. A message requesting intervention will be displayed. The cache values for this behavior must be filled in (i.e. each behavior must be associated with some function) before class creation in order that the newly established type be considered functionally complete.

All internal information maintaining structures of the new type as well as those of the inherited behaviors need to be updated to reflect an accurate status of the system. These include the *defines* set and corresponding *functions* bag of the behavior objects.

Chapter 4

Design for Persistence

In the previous chapter the main memory implementation of TIGUKAT is described. In this chapter an extension of this implementation to provide persistence is described. It should be noted that this is one possible design which has not been implemented.

The storage manager (SM) is that module of an OODBS that lies at the lowest end of the system. Persistent data needs to be stored on permanent disk storage devices to ensure its survival across application program executions. The SM stores and retrieves these data chunks as uninterpreted, untyped, undifferentiated, passive sequences of bytes, determines the on-disk format of objects and performs any necessary translation between main memory and disk formats. It does not attach any semantics to this data and consequently does not see them as "objects" at this level. It is the responsibility of the layer built on top of the SM to attach the necessary *object* semantics and represent the uninterpreted sequence of bytes as an "object". This upper layer has been termed the OM. It is the OM that implements object creation and deletion as instances of types, handles behavior application and fetching/storing of persistent objects through relevant calls to the SM. All manipulation of objects is undertaken via the OM. The OM determines when objects are required to be read in from or written to disk and it utilizes the SM's functionality to perform these tasks efficiently. The procedural interface through which the rest of the system interacts with the OM is called the object manager interface (OMI).

A SM should handle the concept of a unique object identifier (oid) which may be either a *physical* (they contain the actual physical address of the object) or a *logical* (they are mapped through an indexing scheme to obtain the object) oid. If this low level layer has appropriate information about the semantic details of objects there might exist a greater potential for various sorts of optimization (querying, clustering and pre-fetching).

Most of the current RDBMS's storage systems employ a similar technique for handling large amounts of data by splitting data bytes over the smaller pieces of a B-tree-like indexed structure [12]. This strategy has been carried over to the presently available SMs for OODBSs, offering a mere transitory solution to the problem. The development of specialized object storage management facilities is as yet a wide open research problem and needs to be examined in more detail in conjunction with the development of a specialized programming environment that supports the object-oriented paradigm more fully.

In the remainder of this chapter we examine the requirements of persistent data management in an object-oriented environment and propose a suitable design for achieving persistence in the TIGUKAT system. We present a model of persistence based on explicit object level persistence in Section 4.1. In Section 4.2 we discuss ESM in considerable detail and then in Section 4.3 we propose a possible integration methodology between TIGUKAT

and ESM. In Section 4.4 we take a look at some additional storage managers.

4.1 The Model of Persistence for TIGUKAT

A fundamental decision governing the implementation of an object-oriented database system is the strategy to be employed for managing the persistence of system objects. *Persistence* can be defined as the ability of an object to survive across multiple application program executions and a *persistent object* is one which has been endowed with this property. A database system requires that persistence be transparent to the user. The user should not be required to perform any explicit I/O requests or open and close any files. A declarative specification indicating that a particular object be made to persist across sessions should suffice. It will then be the responsibility of the OM to achieve this. The OM could either act immediately or at the end of the current session, by issuing the appropriate commands to the low level SM. The SM (ESM in our case) will write the object out to disk and return a handle (ESM oid) to the OM. The oid serves as a guarantee that at any time in the future, given the oid as input, the SM will extract and return the required object. It is the OM's responsibility to track the handle for gaining access to the object during some future session. As far as the manipulation of all system objects are concerned there should be no distinction between *transient*[1] and persistent objects. Uniformity of access and manipulation of every object should be provided. Ideally, the same piece of compiled code should be able to process all objects irrespective of whether they are transient or persistent and every object should possess an equal right to persistence [4]. In other words, persistence should be a characteristic of an object independent of its type or any of its other characteristics.

As enumerated in [37] there are five basic approaches to persistence prevalent in present systems. The first strategy suggests that the decision about persistence be made prior to object creation. Depending upon whether one wants a persistent or transient object the appropriate object creation routine will be invoked. The second approach is what has been termed *reachability* based persistence. This methodology, incorporated in systems such as O_2 [5], requires that for an object to persist it must be *hung* off some persistent root object via direct or indirect references. In this scheme an object can be made to persist at any stage during its lifetime and may later be made transient. Garbage collection occurs when no references remain to that object. The third predominant approach to persistence is *allocation* based. It restricts the persistence of an object by requiring it to be allocated within some persistent container (collection) during object creation. This requires the existence of persistent storage space with variables naming locations within that space. Objects which are written into persistent variables are guaranteed to be persistent so long as they are maintained in the persistent variable. Systems like ObjectStore [21] take this approach although it renders garbage collection difficult due to the dangling references problem.

The design approach we resolved upon supports the application of an explicit *B_persistent*[2] behavior on an object of any type. This behavior coerces the receiver object to be persistent. Thus, we intend supporting explicit object level persistence. Finally, earlier suggestions for persistence were *type* based, but this was far too restrictive an approach. It was requisite to designate certain types as persistent and all objects instantiated of these types are automat-

[1] *Transient* objects are those objects whose lifetimes span only the current application execution. They cease to exist on program completion and may not be referenced in any successive session.

[2] The primitive behaviors *B_persistent* and *B_transient* are defined on the TIGUKAT type lattice's root type, `T_object`.

ically made to persist. The E language [28, 30, 26] uses a similar approach and maintains a parallel hierarchy of persistent and corresponding non-persistent (db) types. An illusion of orthogonal persistence may be achieved by programming exclusively using db types.

Many researchers now agree that persistence should be a characteristic of objects entirely orthogonal to their type [4]. We opted for this approach since it best maintains the uniformity of object access in the system and does not inhibit the universal use of all types as might be required by an application. Any object existing in the transient system may be explicitly made to persist at any stage of its transient life. Thus, all TIGUKAT objects should be *potentially persistable* while being *inherently transient*. The default is that all objects instantiated during a particular application session are transient unless explicitly made persistent by application of the *B_persistent* behavior. On the other hand, applying the *B_transient* behavior on a previously persistent object will render it transient and that object should cease to exist when the present session terminates. This object though, should continue to be in scope until the end of the current session[3]. We use the terms *activation* and *passivation* for the mechanisms of bringing in a persistent object from disk into main memory and writing out an object from main memory onto disk, respectively. These terms have been borrowed from [1].

The primary grouping construct over which all querying will be done is the collection. We should permit the creation of persistent as well as transient collections. Other considerations would be permitting transient objects to be allocated within persistent collections (these should cease to exist at the end of that particular session), persistent objects allocated within transient collections (these objects should continue to exist in their respective class extents after the session, although the collection itself would be volatile) and finally, objects and the containing collection are either both transient or both persistent.

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

Figure 4.19: The Persistency Matrix

Coercing an object into persistence might result in what we term *persistency side-effects*. We interpret the *persistency matrix* shown in Figure 4.19 as follows. The matrix depicts the various alternative strategies involved in making a TIGUKAT object persist (application of the *B_persistent* behavior on an object). A + entry indicates some persistency side-effect while a − entry indicates side-effect free persistence. If a *type* object is to be made persistent, it is effect-free. This implies that its corresponding class and instances, if any exist, are not required to be made persistent. In the second case, if a *class* object is required to be persistent, the corresponding type must be made persistent but the instances, if any, are not necessarily made to persist. The final case is when a particular instance is made to persist. This has the effect of making both its type and class (excluding other instances)

[3] If it requires to be deleted then a *B_delete* behavior could be applied immediately following the application of *B_transient*. Update and deletion semantics have not as yet been specified in the TIGUKAT model.

persistent. This protects against the object being passivated as an instance of one type and sometime later being erroneously activated as an instance of some other type. The type and class of all the type and class objects in the system form part of the primitive type system. Since these primitive types are by default persistent, the question of making them persistent does not arise.

All system types (whether primitive or user defined) are instances of the type `T_type` and are maintained in the extent of the class **C_type**. All user defined classes are instances of the type `T_class` and are maintained in the extent of the class **C_class** [25]. Persistent type and class objects should all be allocated within a pair of default ESM file objects. A persistent object will be contained in the extent of its type's corresponding class. Each TIGUKAT class should be mapped into an ESM file object. At the termination of the current active session all objects marked as persistent should be passivated by the OM via ESM. This will result in the return of an ESM physical oid (handle) which is to be tracked by the OM.

There exists the compliment of the persistency matrix, which we term the *transiency matrix* (not shown). This matrix derives the repercussions of making presently persistent objects transient during a session (application of the *B_transient* behavior on an object). The effects induced will be precisely the opposite of those seen in the persistency matrix (i.e. making an instance transient will not effect its type or class, making a class transient does not effect its corresponding type but all its instances will be made transient too and making a type transient will have the effect of making its corresponding class and all its instances transient). This model of persistence is a fairly low level and basic one, leaving the responsibility for referential integrity on the application programmer. However, it may be easily extended by taking into consideration other concepts such as the transitive closure of persistency effects. This would ensure that no information is lost when a certain object is asserted to persist but other objects that it references are transient. The system could automatically compute the closure and make persistent every effected object, preserving the referential integrity.

4.2 The EXODUS Storage Manager

It is after examining some of the other storage systems such as Mneme [23], ObServer [20], WiSS [13], EOS [6] and that we are in a position to propose using the EXODUS storage manager [7, 8, 10, 28] to satisfy our persistence requirements. The features ESM has to offer seem to suite our purpose rather aptly. We have been fairly impressed with the performance of an earlier prototype of this system, WiSS [13], as used by the O_2 [35, 5] OODBS. Besides, the E persistent programming language extension of EXODUS [26] is very similar to what we have in mind. Concrete opinions about using ESM in large implementation efforts, such as in [19], help strengthen our proposal.

To summarize the basic requirements of a SM: it should efficiently and reliably support the storage and retrieval of arbitrarily sized objects (a few bytes to a few gigabytes). It should not discriminate between small and large objects and access should be uniform over all objects. The SM must be able to handle changes in an object's size by permitting it to grow or shrink when needed. One should be able to add or delete a specific range of bytes at any location within an object. An SM should provide reasonable random access performance to enable efficient handling of these byte range operations. It should also either directly support oids or provide the functionality needed to support this. An SM should provide for distribution of data and remote access. A client-server configuration is usually

sufficient. Some basic forms of transaction management, recovery and clustering must be offered. Versioning and garbage collection would be plus points. Internal fragmentation should be kept at acceptable levels giving a high disk utilization. In addition, an SM should provide a clear and simple, yet powerful, interface to it's users.

The EXODUS[7, 8, 10, 28] extensible database system project undertaken at the University of Wisconsin offers as its disk manager an efficient, multi-user object storage system [8, 7, 34]. ESM supports most of the required traditional database functionality such as transactions, concurrency control, versioning, indexing and recovery, though at a rudimentary level [9]. ESM is an enhanced version of WiSS (Wisconsin Storage System) [13] with the major added functionality being its uniform treatment of *small* and *large* objects as far as the applications built on top of it are concerned. Internally ESM differentiates between small and large objects but this distinction is transparent to application programs.

ESM has been written in a combination of C and C++ and offers a procedural interface to applications. It can be thought of as an *hybrid* between a SM and an OM, since it manages persistent objects both on disk and in main memory. ESM's primary construct is the *storage object* which consists of an uninterpreted, dynamic sequence of bytes of practically any size. Related storage objects are stored within a *file object* which supports indexes for efficient object access[4]. Presently supported indexes are B^+-tree and linear hashing. For complete internal and operational details the reader is referred to [8, 9, 27, 34].

4.2.1 Architecture

ESM supports the client-server configuration. The client and server may be running on different machines and while the server presently supports multiple clients, a client may not connect to more than one server at a time[5]. The term *client* refers to the application program linked with ESM's code and data structures to form a library, the *client module*[6]. All data is stored by the server on volumes which might either be Unix files or raw disk partitions.

Concurrency control is achieved at either the page or file level and employs the standard 2PL (2 phase locking) protocol. Not only may an entire object be locked, but subportions (pages) of it may be locked if so desired. Six modes of locking are supported and although file objects may be locked in any of these, page level locking supports only the *shared* or *exclusive* mode. System recovery is attained using a fairly traditional form of WAL (write ahead logging) [34]. The transaction manager uses the notion of basic, atomic, *serializable* transactions. All operations involving files or objects *must* be bracketed within a *begin* and *commit*. Nested transactions are not yet supported. An *abort* may be initiated either by the client or the server.

The basic system architecture of ESM is as shown in Figure 4.20. The client(s) and server communicate via RPCs (remote procedure call) through reliable TCP (transmission control protocol) sockets. Object and index operations are performed on the client but operations on file objects are performed on the server to enable efficient recovery techniques to be applied.

If an object is requested by the client, referencing it via its oid, and it is not found to be presently *pinned* in the client's buffer pool, then a request is made to the server to fetch the

[4] It is mandatory that *every* storage object reside within some file object. If no file object is specified then ESM assumes a default file.

[5] The recent release, Version 3.0, now supports distributed transactions.

[6] The TIGUKAT object model and the OMI together can be considered to be the *client* in our system implementation.

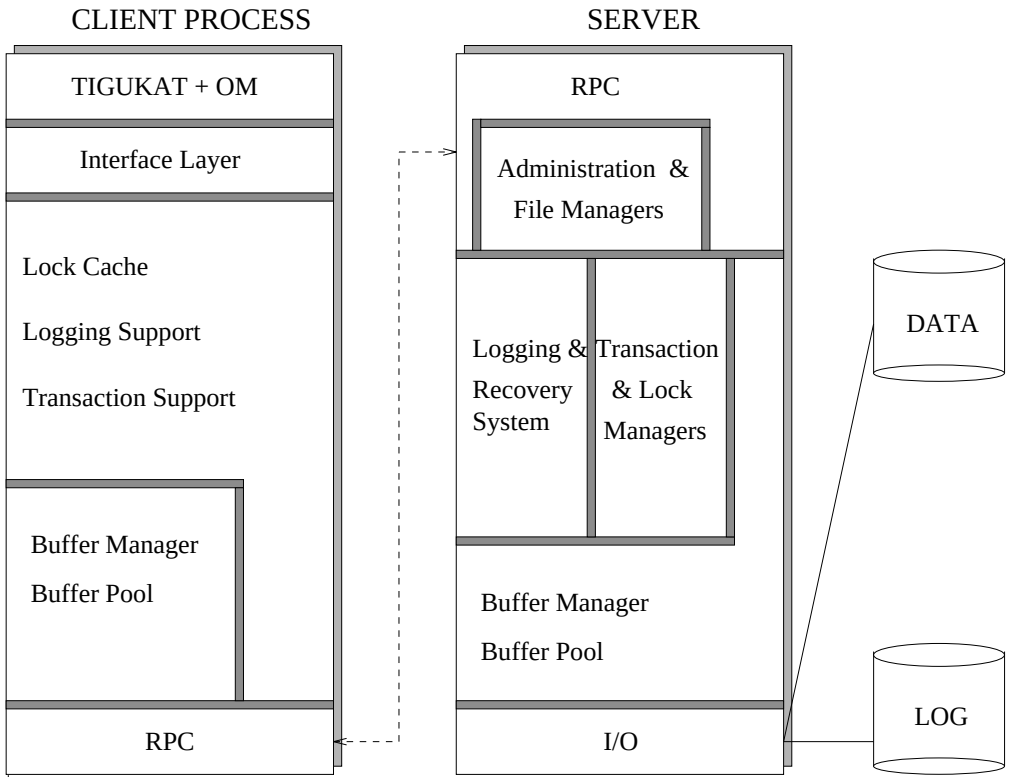


Figure 4.20: Architecture of the EXODUS Storage Manager

object. The server inspects its buffer pool and if it too does not find the requested object there then it reads in the object from disk and sends the required portion of it to the client.

4.2.2 Objects, Oids and Files

ESM internally distinguishes between two types of storage objects, *small* and *large*. The *small* objects are those whose size is limited to a maximum of one data page (4 Kbytes) while *large* objects may be of virtually any size, exceeding a page. A small object may dynamically *grow*, due to appending or inserting bytes into it. If its size exceeds a page ESM then treats it as a large object. A new *large object header*[7] is left in place of the *old* object and the actual data bytes are split across multiple pages of a B^+-tree-like structure. The procedural interface of ESM does not however differentiate between the two and all object creation and manipulation is independent of the size of the object. Small objects may share the page they reside on with other small objects as well as large object headers. The header of a large object is the root of a B^+-tree-like structure and the actual portions of the object are split across multiple pages, each of which is reserved for the exclusive use of that object and sharing of these pages is prohibited with other small objects or headers.

All objects are accessed via *object identifiers* (oids). The oid of a small object points directly to the physical location of that object on disk while the oid of a large object points to its header. ESM employs *structured physical oids*, as opposed to *logical* oids mainly for efficiency reasons [9, 12]. An ESM oid is a twelve byte entity which is structured as shown

[7] Each ESM object has a special piece of data prepended to it, called the the *object header*. The contents of this header include object size, certain properties pertaining to versions and two bytes, the *tag* field, which applications are free to use in an arbitrary manner.

in Figure 4.21.

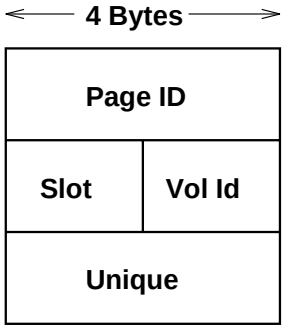


Figure 4.21: The ESM OID Structure

An oid is the *permanent* address of the object on disk and is generated by the server module of the system upon object creation. The *unique* field is a 32-bit value determined by a counter at object creation time and its purpose is to ensure detection of dangling or corrupted references. This field ensures that it is practically impossible for any two objects to have the same oid [34]. Subportions of an object may be addressed by combining a 4-byte offset from the start of the object with its 12-byte oid. This gives us a 16-byte *pointer* to the portion of an object which resides on disk.

All ESM objects are created within some *file object*. A *file object* is the grouping construct for objects which might be related in some way. ESM provides a mechanism for iterating through the objects within a file sequentially. The objects in a file are unordered and every file object has zero or one distinguished *root* object which may be used to attach a name string to the file.

When ESM brings an object (or portion of it) into the main memory buffers, it returns a pointer to a *user descriptor* (UD) in its user descriptor pool. The total size of a UD is thirty two bytes. The object is accessed by dereferencing the *base pointer* portion of that UD. The structure of a user descriptor is as depicted in Figure 4.22.

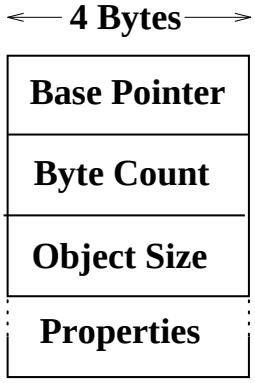


Figure 4.22: The Structure of a User Descriptor

The *base pointer* serves as the 4-byte in-memory pointer to the start of data. The *byte count* contains the size in bytes of the data presently available to the application and the *object size* contains the total size of the object a portion of which is currently *fixed*. The *properties* field holds certain information flags obtained from the object header as well as some user flags which may be employed arbitrarily for application specific purposes such

as maintaining the *type* of the object. Besides these, the UD also maintains the oid of the object that it currently points to (in a slot in the properties field).

The UD serves as a *handle* on the memory resident object which is now said to be *pinned* (fixed). ESM reserves the right to relocate the object in its buffer subsequently reflecting this change in the UD. *Pinning* is basically a two-way contract between ESM and the application program with ESM guaranteeing that the object will be available in main memory so long as it is not explicitly released by the application.

We note that an oid is a permanent valid specifier for an object during its entire lifetime and across all applications that have access to the volume on which the object resides, while the UD serves as its handle only for that duration of pinning in main memory. ESM does not support explicit I/O requests from its clients and reserves the right to schedule actual disk I/O at a convenient time. The OM might call ESM to release a pinned object, but this does not compel ESM to perform any disk I/O. It might just mark that object as *unpinned* and will eventually write it out.

4.2.3 Interface Routines

In this subsection we examine some of the routines provided by the ESM interface which are of prime consequence to our proposed implementation design for persistence. Routines are provided for creation and destruction of objects, dynamic growth or shrinking of existing objects and sequencing through all the storage objects within a file object. We are particularly concerned with those routines which are provided to create, read and write persistent objects.

The routine *sm_ReadObject* brings into memory the requested portion of an object, given its oid and an offset plus length. If the object is not already pinned in the buffer, it is read in and a pointer to a UD is returned to the requesting application. It is to be noted that an object should not be accessed directly while it is resident. Object access is always via dereferencing of the UD's base pointer. The pinned object will remain so until explicitly *unpinned* either by using the *sm_ReleaseObject* routine or by using the option to release the object consequent to an update on it. *Sm_ReleaseObject* informs ESM that the application has finished with the object which may now be safely written out to disk and its UD returned to the pool.

To update an object, once it has been brought into main memory, the routine used is *sm_WriteObject*, which needs to be provided with the UD returned by *sm_ReadObject*. We also need to specify the range of data to be updated and a pointer to the data to be added. For reasons pertaining to recovery it is not advisable to ever write an object directly [34].

Initialization and shutdown routines need to be used at the start and end of an application program. A new persistent object is created using the routine *sm_CreateObject* which expects as its arguments details about which *buffer group*[8] to use, the file in which the object is to be created, the initial value of its data and possibly a pointer to some object header whose tag field is copied into the header for the new object. The application can also specify some storage hints for physical placement of (basic clustering) the new object, although ESM might choose to ignore these. Successful creation of the new object is indicated by the return of a pointer to the oid of the object, which is then used for all future references to that object.

It is also possible to read only the object header portion of any object, and this is useful

[8]ESM implements the concept of a buffer group which is a collection of pages in the buffer pool that may be physically contiguous and have an attached replacement policy.

when it is desired only to know details about a particular object without having to perform actual I/O and have the data read in, especially in the case of large objects.

File objects are handled via similar create and destroy routines besides others which may be used to scan through all objects currently placed in that file. Routines are provided for the B^+-tree and linear hashing indexes currently supported by ESM. A complete discussion of ESM's procedural interface can be found in [34].

4.3 Integrating TIGUKAT with EXODUS

Two main approaches to object activation/passivation can be identified: the *object-faulting* model [30] and the *load-store* model [28]. We discuss both but emphasize on the load-store model and propose the adoption of a similar strategy.

In TIGUKAT, a *class* manages all instances created of its corresponding type. We propose using ESM's file object facility to map each persistent class (which will hold persistent object instances) onto a corresponding ESM file object. ESM requires that every storage object should be allocated in some file object. This fits the TIGUKAT concept of a class well. Each ESM file can be modeled as an instance of a C++ *file* class. All applicable operations (as supported by ESM) should be defined by that C++ class' member functions (implemented as ESM calls). Other collections (classes are also collections) within which these objects are allocated will have relevant references (oids) stored in them.

The object-faulting model aims to provide virtual memory for memory resident objects. Mechanisms similar to demand paging are used to fault a referenced object into the storage manager's buffer. The object is then copied into the virtual memory space and subsequently released from the buffer. All further access to the object is directly via main memory pointers. This model of persistence employs some form of pointer *swizzling Swizzling* is the process of converting persistent database objects between an external form (oid) and an internal form (direct memory pointers). Since swizzling would require an entire object be faulted into virtual memory, we foresee some problems associated with the swizzling of large objects.

Our implementation design for persistence proposes the access of activated (memory resident persistent objects) in the ESM client's[9] buffer *in-place* via relevant calls to the procedural interface provided. We considered this conventional access approach since we do not currently have a programming language interface that can take advantage of swizzled pointers. Furthermore, in-place access makes sharing of objects easier. Although the client buffer is allocated within the applications's private data space objects should never be accessed directly since recovery will be hampered. Object access and manipulation is enclosed within a *begin-commit* of an ESM supported transaction. Object creation will be via a call to the relevant ESM function and will result in the return of an ESM oid. It is then the OM's responsibility to keep track of this handle. When a TIGUKAT application needs to read a persistent object, the OM first obtains the handle on this object, i.e. the oid. A call to ESM's *sm_readObject* function is issued, passing it the oid. If the object is not present in the client buffer then the page containing the object is requested from the ESM server (possibly resulting in some disk I/O if that object is not presently buffered by the server) [36].

ESM will then pin the requested object in the client's buffer and return a pointer to a UD in the pool of UDs that it maintains (Figure 4.23). Dereferencing the *base pointer* portion of the UD will give access to the resident object itself. The object, once pinned

[9] The client would comprise the TIGUKAT library linked with ESM's client interface module.

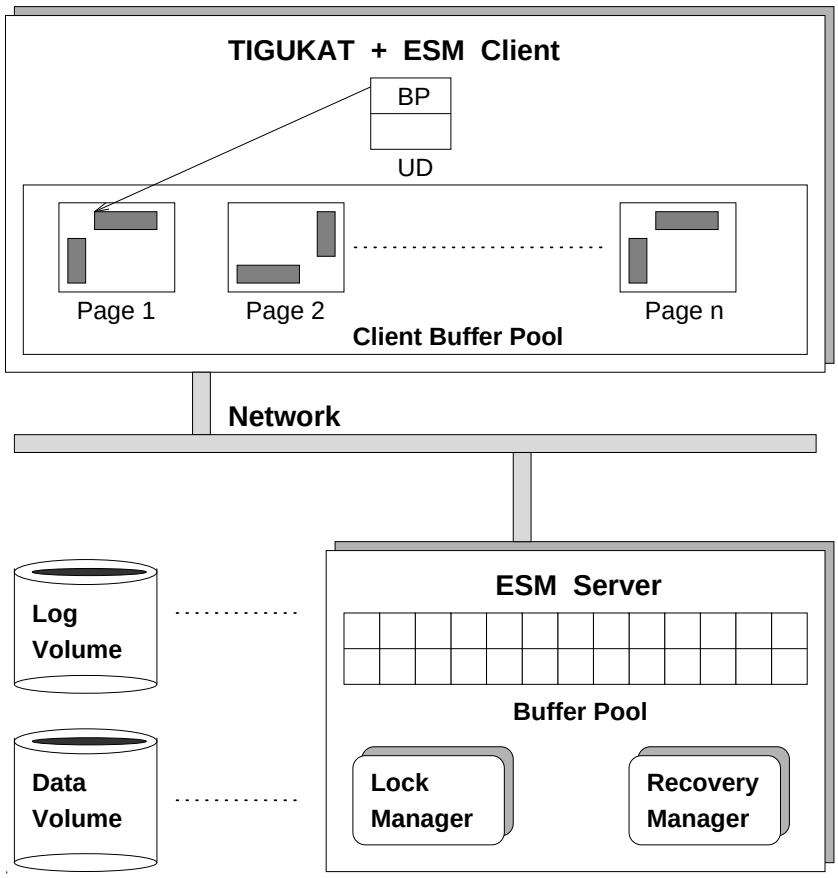


Figure 4.23: Persistent Object Access in TIGUKAT

in the client buffer pool, may be updated with a call to ESM's *sm_writeObject* function. Arguments to this call include the UD and a pointer to the data value that is to be written into the object. ESM will update the object in the client buffer and generate a log record (for recovery purposes).

Once the TIGUKAT application has finished with the object, the OM must call *sm_releaseObject* to unpin the object (maybe tagged onto the *sm_writeObject* routine). If all the objects on the page are unpinned at this time, then the client buffer manager marks it as a candidate for replacement and it will be shipped out to the server when the buffer space is full.

ESM now supports callbacks from server to client and therefore inter-transaction caching of data pages is permitted. This eliminates the disadvantage of the procedural persistence mechanism which was that it might lead to redundant fixing of data since the same object maybe pinned/unpinned multiple times by the application.

The format of an object in main memory will be different from its format on disk. Consider the case where two objects are in main memory and have inter-object references. These references will be normal C++ pointers (4 bytes) in main memory but need to be replaced with appropriate ESM oids (12 bytes) when passivated. The reverse holds through when an object is activated. The OM needs to maintain information about which portions of a object holds references in order to do this.

One approach would involve maintaining a *type table* containing a segment for each defined type. This table would hold information as to which slots within the instances of various types hold references. Note that this structure would hold similar information

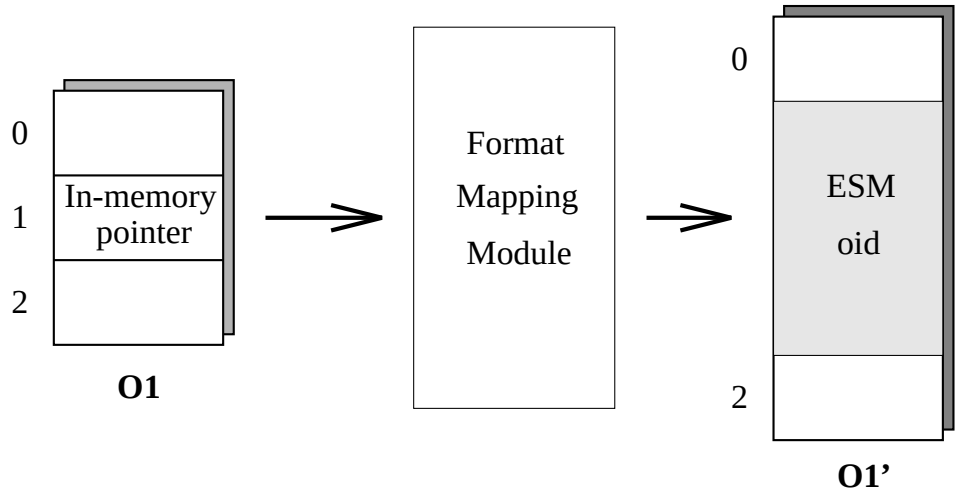


Figure 4.24: Format Mapping during the Passivation Process

as the SC1 supplementary cache described in Chapter 3. In fact, the information in SC1 can be incorporated into the new table. When an object is marked for passivation, the format mapping module of the OM will iterate through the reference holding slots and determine whether the referenced objects are also marked for persistence. Those that are will need to have their main memory pointers replaced by appropriate ESM oids. This process is as depicted in Figure 4.24, where *O1* is the object format prior to passivation and *O1′* is the passivated form. It is required that these referenced objects be passivated prior to the passivation of the referencing object. ESM is then called to store them and an oid is returned for each object made persistent. This oid is inserted into the referencing object in place of the earlier in-memory pointer. A passivated object with references to other peristent objects thus grows in size proportional to the number of persistent objects it references. Note that determining the exhaustive chain of persistency side effects would require a transitive closure of the object references. In case the referenced object is transient, then at the end of the current session a *null* indicator is inserted into the in-memory pointer slot. This implies that we would be losing some information during passivation.

On activation of a persistent object, the OM iterates through those portions which hold oids of other persistent objects (determined from the *type table*). On need for access to any of these objects it consults another table (called the *resident object table* or ROT) to infer whether they have already been pinned in the buffer. The ROT could be a hashed dictionary of oid-UD pairs. If no entry for the oid is present in the ROT, ESM is invoked to bring that object in. The object is then referenced via its present UD.

Another issue which needs to be addressed is how to determine whether a slot in an object contains an oid or a direct in-memory pointer. A runtime software check will be required. We propose masking the first bit of each referential slot in every object. This is used to indicate whether the reference stored there is either a direct memory pointer or a pointer to an oid structure. This will effectively reduce the addressable space by half. The four byte slots will thus suffice in holding references to either transient or persistent objects. If the object being referenced is a persistent object, its oid is obtained from the oid struct, ESM is called and the object is pinned in the buffer. The four byte slot can now be made to point to the UD of the pinned object. Note that the UD also contains the oid of the object it currently references. The oid pointer has to be restored in the slot when the object is passivated.

Our current research has not touched upon some other relevant issues such as *garbage collecting*, *clustering* and *pre-fetching* of persistent objects. These are still open research topics and will be examined at some other stage.

4.4 A Review of Other Storage Managers

A number of other storage managers have been specifically built or adapted to meet the requirements of an object-oriented system. In this section we briefly discuss some of these *viz.* WiSS [13], ObServer [20], Mneme [23] and EOS [6].

4.4.1 The Wisconsin Storage System

The Wisconsin storage system (WiSS) [13] was originally designed as a relational storage manager at the University of Wisconsin. It supports sequential and indexed access to data on disk through a low level manipulation language, providing full control of the physical location of pages on disk. It was designed for operation with a raw disk device and uses record-structured sequential files, unstructured files, B^+-trees and supports the concept of *long data items* of variable, arbitrary size. All these are eventually mapped into the basic unit of persistence: pages. The WiSS interface differentiates between small and long data items. WiSS is restricted to a single thread of execution and this can have serious performance implications due to excessive cpu idle times while WiSS is performing some I/O operation. WiSS does support a basic form of concurrency control but the level of granularity is the entire file and no recovery is possible.

In spite of these shortcomings WiSS has been successfully employed by the O_2 [5] OODBS as the disk manager, providing persistence, disk management and concurrency control for "flat" records of data, with an O_2 transaction being mapped directly into a WiSS transaction. We suggest the use of ESM as our storage manager because it is an enhanced, upward compatible version of WiSS.

4.4.2 The ObServer Object Server

ObServer (object server) [20] is designed to be used as a typeless backend of an object-oriented system, responsible for managing the persistent object store on disk. It reads and writes chunks (contiguous stream of bytes) of memory from disk storage as requested by higher level modules. ObServer possesses a primitive notion of transactions supporting object-level locking. All objects reside on a central server which manages them and controls access to them.

A higher type level is responsible to deal with the semantics of data chunks as objects, interpreted through the associated type definitions. This layer acts as a "client" to the ObServer backend and has to be bound with the client module of the server. Clients request objects from the server, make any required modifications and commit changes back to the server. This is very similar to ESM. When a request is issued to the server the client is not suspended while awaiting a response but continues execution. When ObServer does reply the client may choose to ignore it.

Each chunk of memory stored has an associated handle called the UID (unique identifier). A single UID is associated with each object and when the UID is dereferenced it leads to a header block for that object (similar to ESM). The server maintains the correspondence between UIDs and chunks. This concept is identical to the concept of oids in ESM. ObServer uses a sequential approach to storing data on disks. The objects are stored in

segments on disk and each object may be contained within more than one segment. Each segment in turn is stored in consecutive pages. A hash table is used to store the logical correspondence between UIDs and objects. This involves some overhead which is eliminated by using physical oids, as in ESM.

4.4.3 The Mneme Persistent Object Store

The Mneme (Greek word for *memory*) project stemmed from an effort to integrate programming language and database features for providing better support for cooperative, information-intensive tasks such as those found in software engineering environments [23]. The aim is to provide an efficient persistence mechanism for object-oriented programming languages giving an illusion of a large, shared heap of "objects" which would be directly accessible from the programming language (the ESM interface restricts direct access to objects). The Mneme system seems to be tuned for better performance and less overhead for small objects rather than for large objects. It is more suited for use with a persistent programming language (short oids, an orientation towards automatic storage management, focusing on large scale distribution) than with a OODBS.

A Mneme object is conceived as a contiguous chunk of fields with an associated *logical* (not linked to precise physical locations) oid (4-bytes), as opposed from the physical oids (12-bytes) used by ESM. The oid scheme used is a novel idea based on locality of reference and operating system files. A file is basically a disk file containing a collection of physical segments each of which contains several logical segments. Oids thus have three fields of ten bits each, one field indicating the file, one specifying the logical segment and one referencing the object. Given an oid, objects are accessed via a sequence of table lookups. These "objects" possess no notion of types, classes, behaviors or inheritance, that is the responsibility of upper layers. The configuration of the system is client-server, with the application interfacing with the client interface (Mneme client library), and the client and server possibly running on different machines.

Mneme also supports the notion of transactions within which all object manipulation occurs and store consistency is guaranteed. Mneme provides adequately for a garbage collection algorithm to be implemented within it (ESM is not concerned with garbage collection).

4.4.4 EOS

EOS [6] is a SM for experimental database implementation whose key difference from ESM is that its large object support mechanism eliminates the fixed segment size constraint imposed by ESM. Instead of a large object being stored as a sequence of *fixed* sized segments consisting of physically contiguous disk blocks, as in ESM, EOS stores its large objects as a sequence of variable-sized segments. It offers superior storage utilization by disallowing holes in a segment. Thus, the B-tree-like structure used to store the large objects is slightly different from that used by ESM and therefore EOS's insert, delete and append algorithms are significantly different.

Biliris [6] justifies the need for variable-sized segments to improve ESM's performance for large object reads.

Chapter 5

Conclusion

The implementation design and development of the TIGUKAT object model proved to be an interesting and difficult problem. The objective of the research presented in this thesis was the demonstration of the feasibility of implementing an extensible and uniform core object system. This is precisely what we have achieved. We have affirmed through implemented evidence that the TIGUKAT object model can be successfully employed to satisfy the needs of object data management. We have striven to maintain the uniformity and extensibility of the object model in order that the prototype facilitate modular growth of the system while preserving its modeling power. The type system should be easily modifiable as the system evolves through imminent extensions without unduly affecting the existing logical structure. This requirement dictated that the architecture be modular and structured in layers.

On the basis of difficulties encountered while investigating the mapping from the conceptual model to a feasible implementation design numerous changes managed to filter their way up to the object model (an updated version of the TIGUKAT object model is presently being researched). During the course of this research potential model imperfections were exposed and possible solutions to these were discussed. Some of these might never have surfaced without this attempt at system implementation.

5.1 Future Research

There are several areas which require further work. In this section we propose some guidelines for future research which would help enhance the TIGUKAT system.

- In the current implementation, the system dispatch cache provides an extremely fast dispatch mechanism but suffers a poor space utilization. This excessive memory consumption can be alleviated by using one of the coloring schemes proposed in [15, 2]. The technique involves assigning colors to method selectors with the possibility of multiple selectors being assigned an identical color. This would in effect reduce the number of columns required in the cache. Also, instead of maintaining two supplementary *byte* caches, as we presently do, they could be combined into a single structure which maintains both pieces of information: the nature of the associated functions and accessor function pairing.

- The reassociation of a behavior with either a stored or a computed function, irrespective of any earlier association or of existing instances, needs to be supported. This

should be studied in conjunction with a precise specification of schema evolution, update and deletion semantics.

- The system's performance appears to be deficient during the type creation process. There is quite a bit of iteration involved through the interface sets of the specified direct supertypes. This could cause a significant delay if the number of supertypes is large, each having a considerable interface.

- Investigation of a runtime environment that would directly support TIGUKAT's notion of atomic types is needed. Our present approach of generating atomic types when they are first referenced seems to be a bottleneck to efficiency. This integrated environment should manage its own memory instead of using the memory management provided by the programming language (C++ at present).

- The design for persistence that we have suggested is via ESM's functional interface. This might be a little inefficient since persistent object access is indirect. A considerable amount of non-trivial research is needed to study an elegant pointer swizzling technique where persistent objects might be dynamically faulted into the system's virtual memory on dereference.

- The consequences of accessing multiple databases needs to be studied and issues pertaining to clustering and garbage collecting persistent objects should be investigated in relation to persistence.

Bibliography

[1] R. Agrawal, S. Dar, and N. H. Gehani. *The O++ Database Programming Language: Implementation and Experience*. AT&T Bell Labs, 1990.

[2] P. André and J. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 110–123, September 1992.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings 1st International Conference on Deductive and Object-Oriented Databases*, pages 40–57, 1989.

[4] M. P. Atkinson and O. P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[5] F. Bancilhon, C. Delobel, and O. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers, 1992.

[6] A. Biliris. The Performance of Three Database Storage Structures for Managing Large Objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 276–285, 1992.

[7] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita, and M. Muralikrishna. The Architecture of the EXODUS Extensible DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, 1986.

[8] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*. Morgan-Kaufman, 1990.

[9] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 91–100, August 1986.

[10] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Storage Management for Objects in Exodus. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley Publishing Co., 1989.

[11] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proceedings of the ACM SIGMOD Conference*, pages 413–423, June 1988.

[12] R. G. Cattell. *Object Data Management*. Addison-Wesley Publishing Co., 1991.

[13] H. Chou, D. DeWitt, and S. L. Vandenberg. Design and Implementation of the Wisconsin Storage System. *Software - Practice and Experience*, 15(10):943–962, October 1985.

[14] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proceedings 2nd International Workshop on Database Programming Languages*, pages 80–102, 1989.

[15] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 211–214, October 1989.

[16] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Co., second edition, 1991.

[17] M. Fontana and M. Neath. Checked Out and Long Overdue: Experiences in the Design of a C++ Class Library. Technical report, Texas Instruments Incorporated, Information Technology Group, 1990.

[18] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Publishing Co., 1983.

[19] E. Hanson, T. Harvey, and M. Roth. Experiences in DBMS Implementation Using an Object-Oriented Persistent Programming Language and a Database Toolkit. In *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 314–328, October 1991.

[20] M. Hornick and S. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1):70–95, January 1987.

[21] C. Lamb, G. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

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

[23] J. E. B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Office Information Systems*, 8(2):103–137, April 1990.

[24] R. Peters. *TIGUKAT: A Uniform Behavioral Object Model, Query Model and View Manager for Object Management Systems.* PhD thesis, University of Alberta, 1993.

[25] R. J. Peters, M. T. Özsu, and D. Szafron. TIGUKAT: An Object Model for Query and View Support in Object Database Systems. Technical Report TR-92-14, University of Alberta, October 1992.

[26] J. Richardson, M. Carey, and D. Schuh. The Design of the E Programming Language. Technical Report 824, University of Wisconsin, February 1989.

[27] J. E. Richardson and M. J. Carey. Programming constructs for Database System Implementation in EXODUS. In *Proceedings of the ACM SIGMOD Conference*, pages 208–219, June 1987.

[28] J. E. Richardson and M. J. Carey. Persistence in the E Language: Issues and Implementation. *Software - Practice and Experience*, 19(12):1115–1150, December 1989.

[29] C. Schaffert, T. Cooper, B. Bullis, M. Killian, and C. Wilpolt. An Introduction to Trellis/Owl. In *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, September 1986.

[30] D. T. Schuh, M. J. Carey, and D. J. DeWitt. Persistence in E Revisited - Implementation Experiences. In *Proceedings 4th International Workshop on Persistent Object Systems*, pages 345–359, September 1990.

[31] D. W. Shipman. The Functional Model and the Data Language DAPLEX. In *ACM Transactions on Database Systems*, pages 388–404, 1981.

[32] D. D. Straube and M. T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Office Information Systems*, 8(4):387–430, October 1990.

[33] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., second edition, 1991.

[34] University of Wisconsin. *Using the EXODUS Storage Manager V2.2*, November 1992.

[35] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager: An Overview. In *Proceedings 15th International Conference on Very Large Data Bases*, August 1989.

[36] S. J. White and D. J. DeWitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 419–431, August 1992.

[37] S. Zdonik and D. Maier. Fundamentals of Object-Oriented Databases. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, pages 1–36. Morgan-Kaufman, 1990.

A

C++ Class Declarations

This chapter contains source code excerpts from the class declarations of some of the important support structures.

A.1 The TgObject Class

```
// The class declaration for TgObject

class TgObject {

protected:
  AttrEntry* elem;      // Pointer to allocated storage
  int size;      // Size of allocated storage
  int numberElements;      // Number of elements in TgObject
  int curPos;      // Keeps current position
  void remove ();      // Remove element at current position
  Boolean push (const AttrEntry&);      // Append slot to TgObject

public:
  TgObject ();      // TgObject v;
  TgObject (unsigned int);      // TgObject v(10);
  TgObject (const TgObject&);      // TgObject v = y;
  ~TgObject ();      // Destructor

  Boolean isEmpty();      // Any elements in TgObject?
  int length () const;      // Number of elements
  int capacity () const;      // Maximum number of elements

  void setType(TgObject&);      // Set 1st attr
  TgObject& getType() const;      // Returns 1st attr - always the type
  pter getAttrAt(int) const;      // return attribute at that slot
  TgObject& getTgObj(int) const;      // return attribute at that slot
  Set& getSetAt(int) const;      // return attribute at that slot
  Bag& getBagAt(int) const;      // return attribute at that slot
  TgBag& getTgBagAt(int) const;      // return attribute at that slot
  TgString& getStringAt(int) const;      // return attribute at that slot
```

```
  Cache& getCache() const;       // return Cache
  int getCacheRow() const;       // return Cache row index
  int getIntAt(int) const;       // return attribute at that slot
  double& getDoubleAt(int) const;       // return attribute at that slot
  char getCharAt(int) const;       // return attribute at that slot

  Boolean putAttrAt(int,pter);       // insert attribute at that slot
  Boolean putAttrAt(int,char);       // insert attribute at that slot
  Boolean putAttrAt(int,TgString&);       // insert attribute at that slot
  Boolean putAttrAt(int,char*);       // insert attribute at that slot
  Boolean putAttrAt(int,TgObject&);       // insert attribute at that slot
  Boolean putAttrAt(int,Cache&);       // insert attribute at that slot
  Boolean putAttrAt(int,Set&);       // insert attribute at that slot
  Boolean putAttrAt(int,Bag&);       // insert attribute at that slot
  Boolean putAttrAt(int,TgBag&);       // insert attribute at that slot
  Boolean putAttrAt(int,int);       // insert attribute at that slot
  Boolean putAttrAt(int,double&);       // insert attribute at that slot

  Boolean addEntry (pter);       // Append entry at end of array
  Boolean addEntry (TgString&);       // Append entry at end of array
  Boolean addEntry (Cache&);       // Append entry at end of array
  Boolean addEntry (char*);       // Append entry at end of array
  Boolean addEntry (Set&);       // Append entry at end of array
  Boolean addEntry (Bag&);       // Append entry at end of array
  Boolean addEntry (TgBag&);       // Append entry at end of array
  Boolean addEntry (TgObject&);       // Append entry at end of array
  Boolean addEntry (int);       // Append entry at end of array
  Boolean addEntry (double&);       // Append entry at end of array
  Boolean addEntry (char);       // Append entry at end of array
  Boolean removeEntry (int);       // delete this slot

  TgObject& operator= (const TgObject&);       // v = vec1;
  Boolean operator== (const TgObject&) const;       // Comparison
  Boolean operator≠ (const TgObject&) const;       // Comparison
};

// Declaration for the class AttrEntry (each element of TgObject)

class AttrEntry {
private:
   pter pf;       // A void pointer - may point to any type of object

public:

   AttrEntry();       // Default Constructor
   AttrEntry(pter);       // constructor expecting void*
   ~AttrEntry();       // destructor

   pter getPter() const;       // Get the pointer - return it
```

```
    void setPter(pter);       // Set the pointer

    Boolean operator== (const AttrEntry&) const;       // Comparison
    AttrEntry& operator= (const AttrEntry& );       // Assignment
};
```

A.2 The Cache Class

```
typedef TgObject* (*pfun)(...);
// A pointer to a function that returns a pointer to a TgObject

// Class declaration for class Cache

class Cache {
private:
  pfun** data;      // Pointer to the Cache
  int numRows;      // Number of rows
  int numCols;      // Number of columns


public:
  Cache (unsigned int, unsigned int);      // Cache C(r,c);
  Cache (const Cache&);      // C1 = C2 Copy Constructor;
  ~Cache();      // Destructor

  void put (unsigned int, unsigned int, pfun);      // Assign value
  pfun get (unsigned int, unsigned int) const;      // Get value

  void fill (const pfun&);      // Set elements to value

  int rows () const;      // Return number of rows
  int columns () const;      // Return number of columns
};
```

A.3 The Set and Bag Classes

```
// Class declaration for the purely virtual BaseSet class

class BaseSet {
public:
  virtual int length() const = 0;      // Virtual - defined in subclasses
  virtual TgObject& get(int) = 0;      // Virtual - defined in subclasses
  virtual Boolean addEntry(TgObject&) = 0;
  virtual Boolean removeEntry(const TgObject&) = 0;
  virtual Boolean find (const TgObject&, unsigned int start = 0) = 0;
};


// Class declaration for the Set class

class Set : public BaseSet {

protected:
  SetEntry* elem;      // Pointer to allocated storage
  int size;      // Size of allocated storage
  int numberElements;      // Number of elements in Set
  int curPos;      // Keeps current position

  void remove ();      // Remove element at current position
  Boolean push (const SetEntry&);      // Add TgObject to end of Set

public:
  Set ();      // Set s - Default constructor;
  Set (unsigned int);      // Set s(10);
  Set (const Set&);      // Set s1 = s2 - Copy Constructor;
  virtual ~Set ();      // Destructor

  Boolean find (const TgObject&, unsigned int start = 0);
    // Set current positi on
  Boolean isEmpty();      // Any elements in Set?
  int length () const;      // Number of elements
  int capacity () const;      // Maximum number of elements

  TgObject& get(int);      // Return element, set curpos

  Set& operator= (const Set&);      // s = s1;

  virtual Boolean addEntry(TgObject&);      // Add this element to the set
  Boolean removeEntry(const TgObject&);      // Remove this element from the set

  virtual Boolean operator== (const Set&) const;      // Compare 2 Sets
  Boolean operator≠ (const Set&) const;      // Compare 2 Sets
```

```
};

// Class declaration for the Bag class

class Bag : public Set {

public:
  Bag ();      // Bag b;
  Bag (unsigned int);      // Bag b(10);
  Bag (const Bag&);      // Bag b1 = b2;
  virtual ~Bag();      // Destructor

  Boolean put (TgObject&,int);      // Put element into Bag at given slot
  Boolean addEntry(TgObject&);      // Add element to Bag
  Boolean operator== (const Bag&) const;      // Compare 2 Bags
};

// Declarations for the TgBag Class

class TgBag : public BaseSet {

protected:
  BagElem* elem;      // Pointer to allocated storage
  int size;      // Size of allocated storage
  int numberElements;      // Number of elements in TgBag
  int curPos;      // Keeps current position

  void remove ();      // Remove element at current position
  Boolean push (TgObject&);      // add TgObject to end of TgBag

public:
  TgBag ();      // TgBag v - Default constructor;
  TgBag (unsigned int);      // TgBag v(10);
  TgBag (const TgBag&);      // TgBag v = y - Copy Constructor;
  virtual ~TgBag ();      // Destructor

  Boolean find(const TgObject&, unsigned int start = 0);
  Boolean isEmpty();      // Any elements in TgBag?
  int length () const;      // Number of elements
  int capacity () const;      // Maximum number of elements
  int occurences(const TgObject&) ;      // number of occurences of the object
  int card() const;      // Total value of all counts

  TgObject& get(int);      // Return TgObject
  BagElem& getElem(int);      // Return element

  TgBag& operator= (const TgBag&);      // tBag1 = tBag2;

  Boolean addEntry(TgObject&);      // add this element to the Bag
```

```
  Boolean removeEntry(const TgObject&);      // remove this element from the Bag

  Boolean operator== (const TgBag&) const;      // Compare 2 TgBags
  Boolean operator≠ (const TgBag&) const;      // Compare 2 TgBags

};

// Declarations for the BagElem class

class BagElem {
    int count;
    TgObject* elem;

   public:
      BagElem();      // Default constructor
      BagElem(TgObject&) ;      // Constructor
      ~BagElem();      // destructor

      int getCount() const;      // Returns occurences of element
      void setCount(int) ;      // Sets occurences of element
      void incrCount();      // Increments occurences of element
      void decrCount();      // Decrements occurences of element
      TgObject& getElem();      // Returns reference element object
      void setElem(TgObject&);      // Inserts reference to object into bag
      Boolean operator== (const BagElem&) const;      // Compare 2 elems
      BagElem& operator= (const BagElem&);      // Assignment operator
};
```

B

Behavioral Specifications

This chapter contains a comprehensive behavioral summary of all TIGUKAT's primitive system types implemented as required in [24]. In addition we have added the types `T_behavior_class`, `T_function_class` and `T_semantics`.

Section 1 contains the summary of the non-atomic types and section 2 that for the atomic types.

B.1 Non-atomic Types

Type	Signatures	
T_object	*B_self*:	T_object
	B_mapsTo:	T_type
	B_conformsTo:	T_type → T_boolean
	B_equal:	T_object → T_boolean
	B_notequal:	T_object → T_boolean
	B_persistent:	T_object
	B_transient:	T_object
T_type	*B_interface*:	T_set⟨T_behavior⟩
	B_native:	T_set⟨T_behavior⟩
	B_inherited:	T_set⟨T_behavior⟩
	B_specialize:	T_type → T_boolean
	B_subType:	T_type → T_boolean
	B_subTypes:	T_set⟨T_type⟩
	B_superTypes:	T_set⟨T_type⟩
	B_subLattice:	T_poset⟨T_type⟩
	B_superLattice:	T_poset⟨T_type⟩
	B_classof:	T_class
T_behavior	*B_name*:	T_string
	B_argTypes:	T_list⟨T_type⟩
	B_resultType:	T_type
	B_semantics:	T_semantics
	B_associate:	T_type → T_function → T_behavior
	B_implementation:	T_type → T_function
	B_apply:	T_object → T_list → T_object
	B_defines:	T_set⟨T_type⟩
T_function	*B_argTypes*:	T_list⟨T_type⟩
	B_resultType:	T_type
	B_source:	T_object
	B_execute:	T_list → T_object
	B_executable:	T_object

Type	Signatures	
T_collection	*B_typeOf*:	T_type
	B_extent:	T_set
T_class	*B_shallow*:	T_set
	B_new:	T_object
T_class_class	*B_new*:	T_type → T_class
T_type_class	*B_new*:	T_set⟨T_type⟩ → T_set⟨T_behavior⟩ → T_type
T_collection_class	*B_new*:	T_type → T_collection
T_behavior_class	*B_new*:	T_string → T_list⟨T_type⟩ → T_type → T_behavior
T_function_class	*B_new*:	T_list⟨T_type⟩ → T_type → T_function
T_semantics	*B_name*:	T_string
	B_argTypes:	T_list⟨T_type⟩
	B_resultType:	T_type

B.2 Atomic Types

Type	Signatures	
T_atomic		
T_boolean	*B_not*:	T_boolean
	B_or:	T_boolean → T_boolean
	B_if:	T_object → T_object → T_object
	B_and:	T_boolean → T_boolean
	B_xor:	T_boolean → T_boolean
T_character	*B_ord*:	T_natural
T_string	*B_car*:	T_character
	B_cdr:	T_string
	B_concat:	T_string → T_string
T_real	*B_add*:	T_real → T_real
	B_subtract:	T_real → T_real
	B_multiply:	T_real → T_real
	B_divide:	T_real → T_real
	B_trunc:	T_integer
	B_round:	T_integer
	B_lessThan:	T_real → T_boolean
	B_lessThanEQ:	T_real → T_boolean
	B_greaterThan:	T_real → T_boolean
	B_greaterThanEQ:	T_real → T_boolean
T_integer	Behaviors from T_real refined to work on integers	
T_naturals	Behaviors from T_integer refined to work on naturals	
T_set	*B_mbrType*:	T_type
	B_union:	T_set → T_set
	B_diff:	T_set → T_set
	B_intersect:	T_set → T_set
	B_forEach:	T_behavior → T_list → T_bag
	B_containedBy:	T_set → T_boolean
	B_cardinality:	T_natural
	B_elementOf:	T_object → T_boolean

Type	Signatures
`T_bag`	*B_occurrences*: `T_object` → `T_natural` *B_count*: `T_natural` Behaviors from `T_set` refined to preserve duplicates
`T_poset`	*B_ordered*: `T_object` → `T_object` → `T_boolean` *B_ordering*: `T_behavior` Behaviors from `T_set` refined to preserve ordering
`T_list`	Behaviors from `T_bag` and `T_poset` refined to preserve duplication and ordering